Reconfigurable computing platform for small-scale resource-constrained robot

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Byung Hwa Kim

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Tryphon T. Georgiou, Thomas A. Posbergh

January 2010

Acknowledgements

First of all, I would like to express my great appreciation to my advisors, Professor Tryphon Georgiou and Professor Thomas Posbergh for their endless encouragement, valuable guidance, and financial support throughout my Ph.D. study in the University of Minnesota. I would like to thank Professor Richard Voyles for his advice, inspiration, and financial support during my academic period. I also express my gratitude to my Ph.D. advisor, the late Professor E. Bruce Lee. He supported my research efforts with both facilities and funds when I first came to Minnesota for my Ph.D. study.

I am also grateful to Professor Gerald Sobelman and Professor Stergios Roumeliotis for serving as the committee members in my final oral examination. Their comments and suggestions helped me to improve the quality of my thesis.

I would like to thank current and former members of our research group. Particularly, Jaewook Bae, Colin D'souza, Amy Larson, Fu Lin, Binh Lieu, Xinhua Jiang, Rashad Moarref for our numerous discussions on various research topics.

I would like to give special thanks to my parents, my parents-in-law, my sister, and my brother for their love, encouragement, and constant support to continue my studies.

Finally, my deepest appreciation goes to my wife, Hyun Su Park, and my two sons, Junhee Kim and Hwanhee Kim, who always supported me with their love.

Dedication

This dissertation is dedicated to my beloved wife, Hyun Su Park, who has believed me and supported me all the way since the beginning of my studies.

Abstract

Specific applications often require robots of small size for reasons such as costs, access, and stealth. Small-scale robots impose constraints on resources such as power or space for modules, but they still require great functionality to do challenging tasks such as surveillance, urban search and rescue, application-specific sensing, robotic assembly, etc.

This thesis develops a reconfigurable computing platform for small-scale resource-constrained robots that allows rapid deployment of available hardware and software for a specific task. Resource-adaptive control is introduced where control parameters can be changed with respect to the resource usage such as power consumption, area, or execution speed, as well as plant change. The use of a Field Programmable Gate Array (FPGA) is essential in providing the flexibility in hardware for both sensor interfacing and hardware-accelerated computation. In this study, reconfiguration is achieved by two steps; static reconfiguration and dynamic reconfiguration.

This thesis utilizes reconfiguration technology in order to solve issues on resources and functionality. Prior to executing a task, a robot needs to be equipped with necessary sensors and actuators. This thesis introduces a new scheme of configuring a robot system before deploying a robot into a field, which is called static reconfiguration. Static reconfigurability of the hardware manifests itself in the form of a “morphing bus” architecture that permits the modular connection of various sensors. It is a novel sensor bus in the fact that no bus interface circuitry is required on a sensor side – the bus “morphs” to accommodate the signals of the sensor.

Dynamic reconfiguration or run-time reconfiguration is performed in order to maximize the resource utilization in terms of power, area and speed while the robot is executing tasks. A software architecture for hardware/software dynamic reconfigurability is proposed and it provides for the reallocation of hardware and software resources at run time as the mobile, resource-constrained robots encounter unknown environmental conditions that render various sensors ineffective. A novel strategy to search a configuration tree is presented and metrics for cost functions in the tree are introduced. Resource-adaptive controller can modify control parameters, or change the order of a plant model, or even choose a different control algorithm by examining resource utilization during dynamic reconfiguration.

Table of Contents

List of Tables

List of Figures

I. Introduction

1.1 The resource-constrained robot

There are many robots in the field with different size and power (see Figure 1). They execute different tasks with their own capabilities.

In [1], field robots are defined as robots that extend human's sensing and/or manipulating capability to a remote location. In my thesis, field robots are referred as robots which can operate in difficult-to-access areas, and they are often required to do multiple tasks. Examples of field robots would be a robot in nuclear plant, plume tracking [2], or urban search and rescue.

For specific applications, field robots often require small size for cost, access, stealth or other reasons. Small-scale robots typically have significant constraints such as limited size or low power, but they still require great functionality to do challenging tasks such as surveillance, urban search and rescue. Also it requires adaptation capability for different environments.

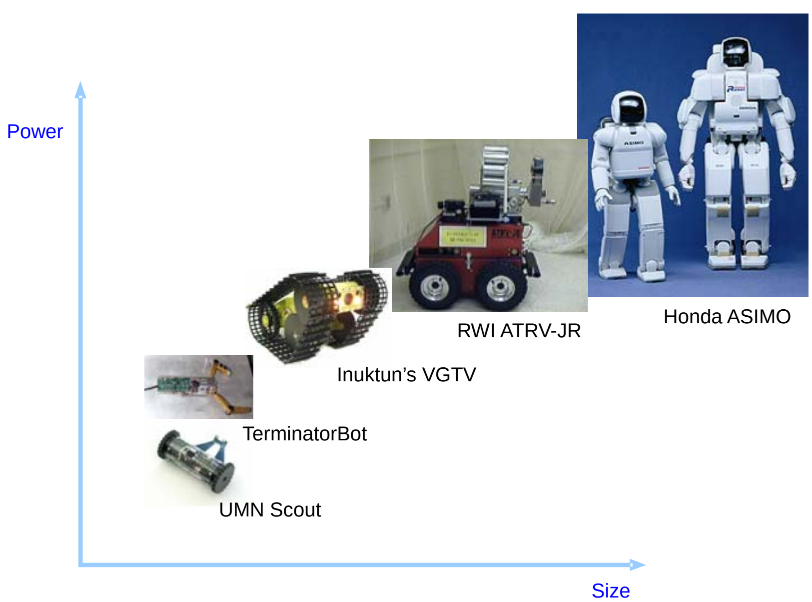
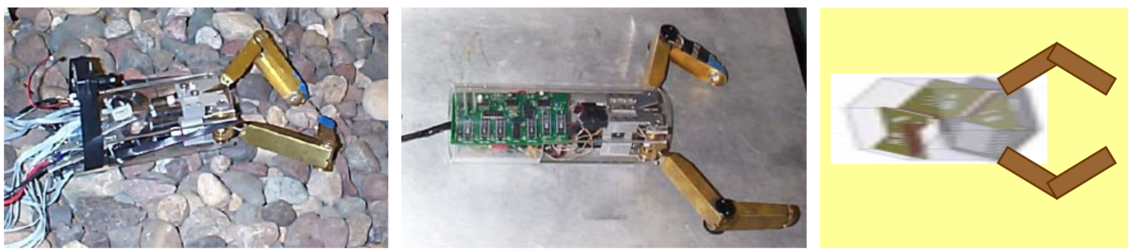


Figure 1. Various robots with different size and power.

In this thesis, two examples of resource-constrained robots are introduced; TerminatorBot [3] and a two-wheeled balancing robot.

In earlier work, a small-scale niche robot called TerminatorBot was developed for urban search and rescue purpose. My contribution to this work was to find and design a powerful new central processing board equipped with reconfigurable feature to replace an existing 8-bit microprocessor. Different versions of TerminatorBot are shown in Figure 2.



(a) Initial version

(b) Core-bored search and rescue version

(c) FPGA-based reconfigurable computing platform

Figure 2. Different versions of TerminatorBot.

TerminatorBot was a typical example of a field robot which requires reconfiguration feature because of the following reasons.

- TerminatorBot has a small form factor for core-bored search and rescue operation.
- It cannot carry all the sensors available, but it can only be equipped with partial sensors because of its small size.
- During run time, it can encounter unknown situation such as a broken sensor, an obstacle, or a foggy view.

For the initial version of TerminatorBot, it had a large bulky tether connected to two external PCs for motion control and vision processing. An Atmega128 microprocessor [4] was used to communicate with the external PC and drive motors. Atmega128 is an 8-bit microcontroller with 128K Bytes in-system programmable flash memory, PWM channels, and 8 channel A/D converters. The second version of the robot was aimed for core-bored search and rescue. This version had a thinner tether for power, video, and command signals. The Atmega128 microprocessor was adapted for motion control of 6 actuators. The external PC was still needed for vision processing and Personal Digital Assistant (PDA) was used to provide kinematics and user interface to command robot.

For the next generation of robot, both vision processing and motion control are integrated into single chip by virtue of higher processing power of FPGA compared to the 8-bit microcontroller. This version would be tetherless with battery power and wireless communication capability. It still requires high performance to adapt different environment. However, heavy computational tasks such as vision, motion control, etc often require more computational power than conventional microcontrollers provide. An alternative to power-hungry, full-fledged CPUs for small-scale robots to achieve such heavy-duty tasks is the relative power efficiency of hardware acceleration. One way of providing this is to use Application Specific Integrated Circuits (ASIC) which are custom-designed for a specific task. These, however, are often too constraining for a general purpose robot. Each operation potentially requires a different suite of sensors and/or actuators. Due to size and power constraints it is often not possible (or even necessary) to carry all the sensors and actuators with it for all possible tasks.

Another example of resource-constrained robots is a two-wheeled balancing robot. The balancing robot has dynamics similar to an inverted pendulum and is intrinsically unstable, and it requires adaptive control to adapt the robot a changing environment such as plant parameter variation. As a preliminary version of balancing robot the following figure shows one example LEGO robot which was developed in our laboratory using LEGO Mindstorms NXT [5, 6].

The self-balancing robot has similarity with rockets or space shuttles in terms of stability condition. Rockets are also inherently unstable and not easy to control a moving direction since they generate propulsion from the bottom of them, which is pretty similar to the situation of balancing a long stick with human's hand. The dynamics of a rocket change even more significantly when it is released at launch. One noticeable point about Rocket is that its mass is changing as it consumes fuel, which means that plant parameters are time-varying and thus it require certain kind of adaptive controller.

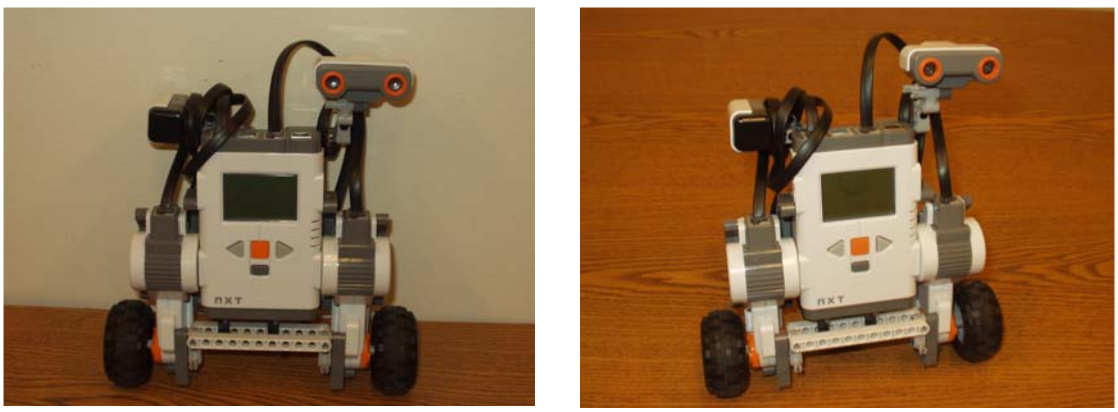


Figure 3. Photographs of LEGO balancing robot.

1.2 Reconfigurable computing

Reconfigurable computing was originally introduced in the late 1960s by Estrin et al. at UCLA [7]. There has not been rigorous study because of lack of realizable reconfigurable hardware. Reconfigurable chips such as Field Programmable Gate Arrays (FPGAs) have only been around since the early 1990s for this kind of a computing platform. Papers with good introductions and overviews of reconfigurable computing include [8-13]. They explain about the trend, history, different methods, or tools in reconfigurable computing.

Reconfigurable computing has several advantages. First, it allows one to execute and test more hardware than a physical logic gate limit and consequently it yields to conduct more tasks. It also helps to save hardware resources or power consumption; otherwise additional hardware will be needed to perform different tasks.

Another benefit is that it permits convenience in firmware update. With reconfigurable computing, the firmware of a product can be updated later or certain feature can be added after the product has been shipped.

1.3 FPGAs

A Field Programmable Gate Array (FPGA) is a type of user programmable digital logic devices called Programmable Logic Devices (PLDs). A PLD is an integrated circuit that user can program digital logic and various digital functions repeatedly. FPGA typically consists of programmable logic blocks, interconnecting routing channels, and input/output pads as shown below [14].

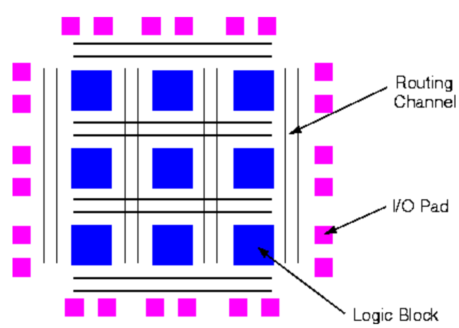


Figure 4. FPGA structure.

The FPGA logic block can be realized in many ways. A basic logic block may consist of a 4-input look-up table (LUT), and a D flip flop [14]. There is one output as shown below. The logic block has four inputs and a clock input.

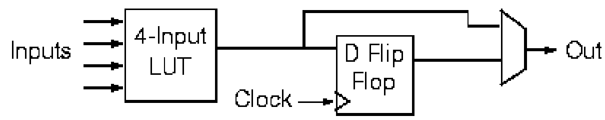


Figure 5. Logic block structure.

Control Algorithms were traditionally implemented using software platforms such as microcontrollers or Digital Signal Processors (DSPs). In these software platforms control scheme stored in memory is sequentially executed. In comparison with microcontrollers, FPGAs provide parallel processing of modules to implement controllers and thus yield better performance than microcontrollers with hardware-accelerated computation. DSPs are designed to implement complex algorithms, but they are expensive compared to microcontrollers. In this study, FPGA is used to implement digital controller to resolve both performance and cost issues.

1.4 Resource-adaptive control

Adaptive control has been historically used in process control dating back to 1951 when it was proposed by Draper and Li [15]. They introduced a control system to maximize the performance of an internal combustion engine in the presence of uncertainties.

Whitaker et al. [16] in 1958 researched adaptive aircraft flight control systems introducing model reference adaptive control (MRAC) systems. A self-oscillating adaptive system was considered by Li and van der Velde [17] in 1960. Astrom and Wittenmark [18] developed self-tuning regulators (STR), which can be easily implemented using microprocessor and this control scheme was widely applied in many control areas.

Zhang and Jiang [19] give an overview of adaptive control and reconfigurable fault-tolerant control system and different mathematical tools are shown in the paper. Adaptive control is a control technique to adjust controller parameters to adapt environmental changes such as plant change or sensor uncertainties.

Compared to traditional adaptive control, resource-adaptive control is a new scheme to change control parameters at run time in an FPGA. Approach in this thesis is to change the hardware architecture of adaptive controller through FPGA reconfiguration. Resource-adaptive control is proposed where control parameters are changed with respect to the resource usage such as power consumption, area, or execution speed, as well as plant change. This strategy is applied in the study of two-wheeled balancing robot. In resouce-adaptive control parameters are changed at run time. Linear Quadratic Regulator (LQR) control was chosen to stabilize the robot, since it is a full-state feedback controller and suitable to work with multiple-input multiple-output (MIMO) system. Two feedback signals of the robot, position and tilt, are used for the LQR controller. When there is any change in plant parameters which is causing instability of the robot and which means change in robustness of the system, different Q or R values of the LQR controller are chosen in order to meet design specifications.

The architecture of proposed resource-adaptive controller is shown below. A conventional control system would be composed of a plant, actuators, sensors, interface logic and controller. In addition to these classical blocks, resource-adaptive controller has the following functional blocks.

- 'Environmental change detection': While monitoring the system, this module detects any change in plant, actuators, and sensors, for instance, plant parameter change, sensor failure, or malfunctioning actuator.
- 'Reconfiguration mechanism': From the result of environmental change detection, this

module will try to change control parameters, interface logic, or even order of plant model.

- 'Reconfigurable controller': It can change control parameters by way of reconfiguration in order to meet performance requirements.

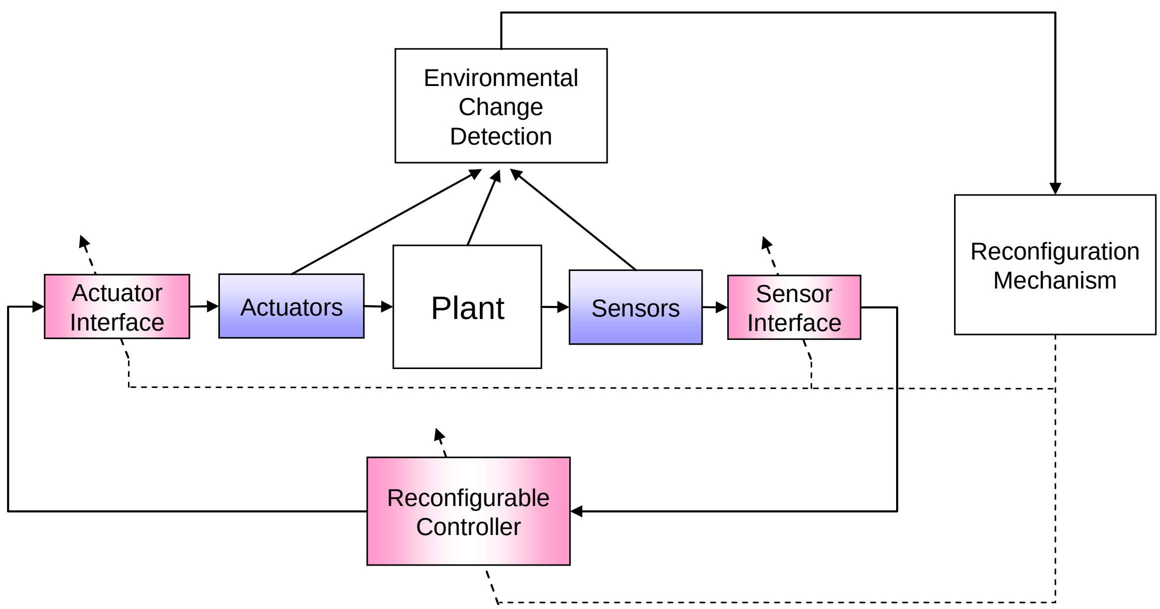


Figure 6. Architecture of resource-adaptive controller.

Digital controllers are traditionally implemented by software running on microcontrollers or DSPs. These processors execute operations serially in order to run algorithms. Compared to these, FPGA provides concurrent processing of modules to implement controllers and will potentially yield better performance than microcontrollers, even with hardware-accelerated computation. FPGAs can also provide more flexibility than Application Specific Integrated Circuits (ASIC), since ASIC are custom-designed for specific task. Reconfigurable computing fits between the microprocessor and the ASIC, i.e. accelerating algorithms by implementing them in hardware while achieving the flexibility to change the algorithm when required (see Figure 7). Recent FPGA has processor core in it as well as logic blocks, for example, Virtex series FPGA from Xilinx. Virtex-4 FX FPGAs could comprise up to two embedded PowerPC 405 cores. The IBM PowerPC 405 core is a 32-bit RISC CPU for use in custom logic applications [20].

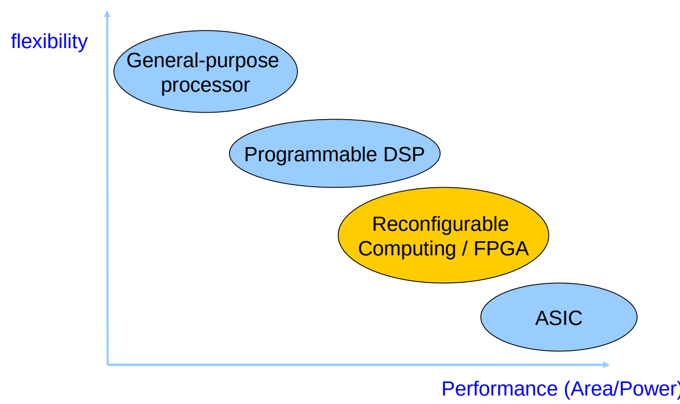


Figure 7. Tradeoffs between different paradigms of computation.

1.5 Problem description

This thesis develops a reconfigurable computing platform for small-scale resource-constrained robots that allow rapid deployment of statically configured hardware and software for a specific task.

Reconfiguration is essential for the following reasons: First, the robot has limited resources such as battery power, small space, restricted RAM size, etc. Since the robot has battery power, we may save power usage through reconfiguration if we can choose power-efficient modules and eliminate useless modules which consumed considerable power. The second reason is that the robot needs to perform various tasks such as motion control, vision processing, sensor fusion, or search and rescue operation. By way of reconfiguration, it could choose suitable configuration for different tasks. For example, a robot might be used for searching collapsed building at one moment and then for rescuing survivors the next. The other reason is that quality of service can be improved such as task success rate or execution speed through reconfiguration.

Another issue regarding reconfiguration is the question of ‘When is reconfiguration necessary?’. When task is changed, the robot may need to reconfigure to achieve the specific task. Also, when the environment is changed, reconfiguration is required. For instance, when a robot is moving from one terrain to another, it may reconfigure to adapt new terrain by changing gait or reducing power usage. The change in the robot could be another reason for reconfiguration, for instance, when sensor is broken or robot is not moving for any reason or the mass of robot has been changed.

The following Figure 8 summarizes the need for reconfigurable platform with resource-adaptive control.

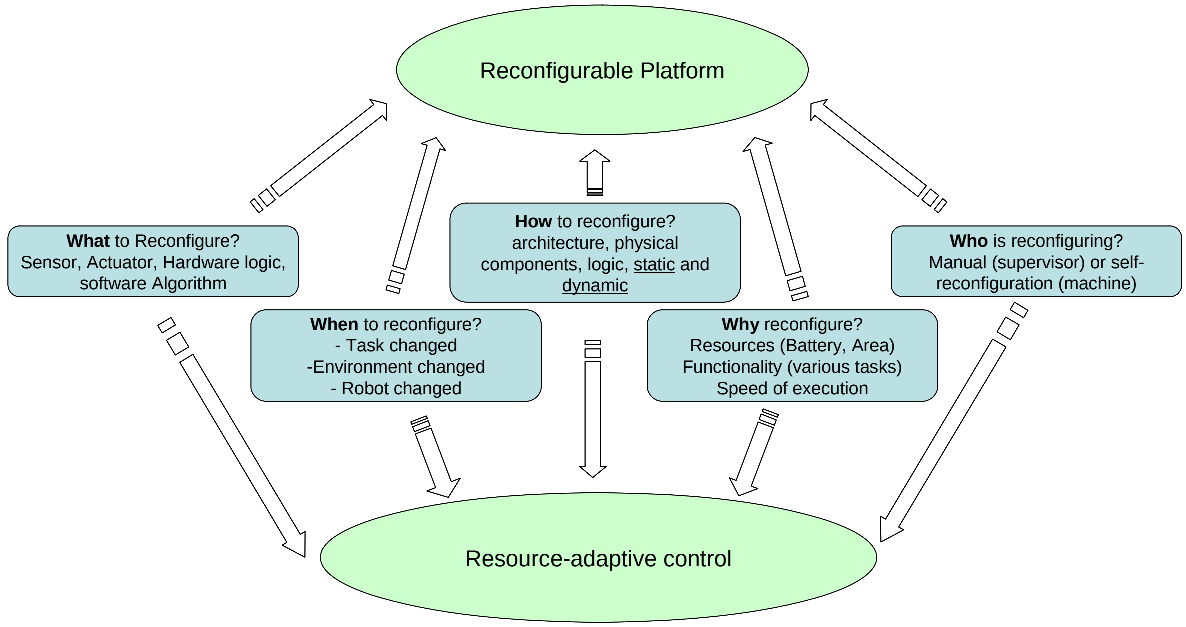
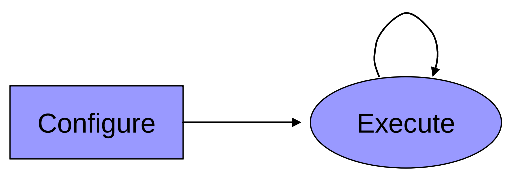


Figure 8. Reconfigurable platform with resource-adpative control.

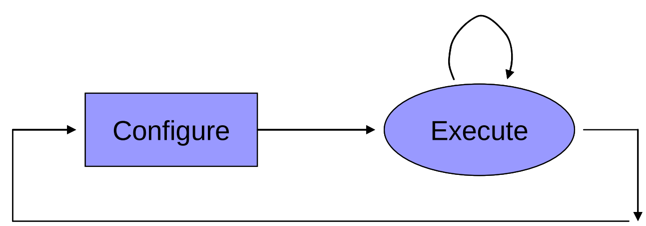
In this thesis, we propose and investigate a flexible low-cost high-performance system with easy configuration. The system is designed around an FPGA framework that allows a user the flexibility of using different devices without having to be concerned with the interfacing details. It is based on the novel "morphing bus" concept [2, 21]. The novelty of this architecture is the absence of interface logic on the side of the sensors and actuators with all the interface logic being moved into the FPGA. The FPGA logic provides the dedicated hardware required to achieve greater performance. The Virtex-II^{TM} Pro FPGA is used for this morphing bus. The morphing bus uses static reconfiguration to interface devices prior to deployment. A prototyping system was built to test its functionality.

I also propose a software architecture for hardware/software dynamic reconfigurability which provides for the reallocation of hardware and software resources at run time as the mobile, resource-constrained robots encounter unknown environmental conditions that render various sensors ineffective. I will characterize all the hardware/software components to drive rational reconfiguration.

The difference between static and dynamic reconfiguration is illustrated in Figure 9. Static reconfiguration, also called compile-time reconfiguration, is configuring a system before dispatching a robot into a field. In contrast to this, dynamic reconfiguration, called run-time reconfiguration, can configure a partial or whole system while run time.



(a) Static reconfiguration (or compile-time reconfiguration)



(b) Dynamic reconfiguration (or run-time reconfiguration)

Figure 9. Static and dynamic reconfiguration.

Resource-adaptive control is applied for two-wheeled balancing robot in order to adapt changing environment or plant variation. A balancing robot has been modeled and this new scheme is to change control parameters at run time or to reduce the order of plant model through FPGA reconfiguration.

1.6 Related work

1.6.1 Reconfigurable computing in robotics

A system is described in [22] that has flexible I/O peripherals whose interfaces can be added and modified by reconfiguring the embedded FPGA. However the only contribution that they claimed in that system was that every pin on the chip they fabricated was identical and could be controlled either as an input or output; either by the microcontroller or FPGA.

Rauma et al [23] proposed a system which takes parameters of bus width, number of modules, number of registers etc and a software tool generates the correct structure for the control applications. Templates were created for every module that described the interface to the bus. This bus however was internal to the FPGA.

Guéganno and D. Duhaut [24] use the FPGA as an I/O card but use external interface logic which our system seeks to eliminate completely.

In [25] a framework called ARCHITECT-R to design and program hardware for robots based on FPGAs is presented. The aim of the project is to allow applications developed in CES (C for Embedded systems) to be translated into hardware/software components, to be executed in a microprocessor and hardware structures, both to be implemented in reconfigurable hardware. This work implemented PID controllers in serial and parallel form. However unlike what is proposed in this thesis, they had not dynamically reconfigured between them.

The YaMoR robotic platform [26] can be reconfigured both electrically and mechanically. Reconfiguration of the electronics is achieved using the Spartan-3 FPGA with a Microblaze soft processor. They use the module based flow to reconfigure the FPGA. The module is defined in VHDL and synthesized by the user. They provide scripts for easily generating the corresponding configuration bitstreams for dynamic partial reconfiguration. These scripts however are only valid for their specific robotic controller and are not general. They also allow the user the flexibility to either implement the control software totally in software or have a mixture of hardware and software. Their scripts will generate the appropriate bitstreams to configure the device.

Paiz et al. used reconfigurable hardware in mini-robotic application [27]. Two application examples are presented: reconfigurable digital controllers and image processing in robot vision. They developed a mini-robot fabricated in Molded Interconnect Device (MID) technology featuring an integration of electrical paths and components into the chassis.

Chakravarthy and Xiao [28] researched resource-constrained miniature robots with high performance. They implemented Pulse Width Modulation (PWM) module and software PID control algorithm, but they didn't research dynamic reconfiguration yet.

The benefit of software-based computation over hardware-based computation is the ability to reconfigure at run time. This Run Time Reconfiguration (RTR) of computation is the basis for the flexibility and rapid growth of software-based solutions. But software requires a hardware target on which to run.

FPGA chips can be reconfigured, too, but most only permit Compile Time Reconfiguration (CTR). However, a new breed of FPGA chips, such as XC6000, Virtex-II Pro or Virtex-4, allows Run Time Reconfiguration (RTR) of logic [29]. Furthermore, some of these new FPGA chips, such as the Virtex-II Pro, have embedded microprocessor units (MPU), making it possible to build power-efficient and highly reconfigurable system-on-chip designs. These systems combine reconfigurable software, a power-efficient hardware target on which the software can run, and reconfigurable hardware all on a single chip.

"hardware/software co-design" usually refers to methodologies that permit the hardware and software to be developed at the same time - splitting some functions to be implemented in hardware for additional speed, while others are implemented in software to free up logic resources. This is normally done offline. The combination of hardware/software co-design techniques with online RTR capability at both the hardware and software levels can optimally assign functions between the FPGA and software dynamically [30].

In order to achieve this level of RTR, the system specification must be partitioned into temporal exclusive segments, a process known as temporal partitioning. A challenge for RTR is to find an execution order of a set of sub-tasks that meet system design goals, a process known as context scheduling. Several approaches can be found in the literature describing these problems (e.g. [31]). All these approaches depend on performance and resource requirements of the requisite sub-tasks to make an optimal tradeoff [30].

FPGA-based System-on-Chip (SoC) designs have been widely applied in digital system applications and RTR research has been addressed by many researchers. Elbirt et al. explored FPGA implementation and performance evaluation for the AES algorithm [32]. Weiss et al. analyzed different RTR methods on the XC6000 architecture [29]. Shirazi described a framework and tools for RTR [33]. Noguera and Badia proposed a hardware/software co-design algorithm for dynamic reconfiguration [26]. FPGA power modeling and power-efficient design have also been studied by various researchers [34-38].

Larson [39] studied gait adaptation which is also a kind of reconfiguration to change robot's gait with different terrain. She classified terrain to appropriately adapt gait. Gait bounce measure was derived from vertical motion of tracked features during locomotion and then characteristics of terrain were extracted from this gait bounce. Efficiency metric was used to show how terrain classification may be used for gait adaptation. Zhao et al [40] investigated an FPGA-based PID motion control system for small, self-adaptive systems. The tradeoffs between different designs are discussed in terms of area, power consumption, and execution time. The characteristics can be used to dynamically reconfigure the robot at run time.

Yan Meng [41] presented an agent-based reconfigurable architecture and self-reconfiguration platform for dynamic reconfiguration at run time by an embedded processor. However, they fail to show experimental results of dynamic reconfiguration feature.

Ramacciotti et al [42] utilized dynamic reconfiguration methods to re-design control electronics of a scientific space experiment module and showed better performance in terms of area and power. Their experiments were limited by the capability of the ATMEL technology.

Danne and Bobda [43] did analytical research on dynamic reconfiguration of distributed arithmetic controllers. They showed the architecture and task-graph of multi-controller and analyzed the performance/resource trade-offs of distributed arithmetic implementation.

Paiz et al [44] showed a design flow for dynamically reconfigurable digital controllers using System Generator from Xilinx. They developed system architecture in high-level descriptions, but they failed to show concrete example of implementation for control applications.

To utilize the reconfiguration feature of FPGA, Blodget et al [45] introduced self-reconfiguration platform using internal configuration access port (ICAP) and embedded processor core inside of the FPGA.

Stewart et al [46] proposed a new software abstraction for designing and implementing dynamically reconfigurable real-time software. They proposed programming model to provide guidelines to control engineers for creating software modules by using port-based object.

Reconfigurable hardware in wearable computing is introduced in [47]. They presented a concept and a prototype of a wearable unit with reconfigurable modules (WURM).

Basic tutorials on FPGA-based dynamic reconfiguration are shown in Appendix A. Other researcher's work on building partial reconfiguration flow is introduced.

1.6.2 Adaptive control and self-balancing robot

There is also continuous research on two-wheeled robots in recent years.

Robots similar to Segway have developed by a lot of researchers recently [48-52]. A hardware design of a two-wheeled inverted pendulum mobile robot was described in [48, 50]. Inverted pendulum type assistant robot (I-PENTAR) was described in [49]. The motion planning of standing and sitting motions for the robot was proposed.

Adaptive control using LQR is developed by many researchers. Petersen [53] introduced gain scheduled LQR to provide stability and robustness with respect to highly uncertain tire characteristics and changing road conditions. The controller gain of LQR depends on the speed which is gain scheduling. Gain scheduling was implemented by switching gain matrices.

Autonomous helicopter at MIT [54] used gain scheduling scheme for LQR synthesis based on discrete switching of gain tables with a changing speed.

Budiyono [55] investigated a new Linear Parameter Varying (LPV) technique to do model identification for small scale helicopter. The identification scheme employed recursive least square (RLS) technique on the LPV system.

Adaptive LQR for Uninterruptible Power Supplies (UPS) was presented in [56, 57]. A recursive least square (RLS) estimator identified the plant parameters which were used to compute LQR gains regularly.

Automated controller implementation process using FPGA was shown in [58]. Control methods such as PID, pole placement, LQR, LQG/LTR, and H_∞ are considered.

1.7 Thesis contributions

Contributions of this thesis are the following.

First, the design of a reconfigurable computing platform for small-scale, resource-constrained robots is described. This platform adopts the morphing bus architecture for static reconfigurability. The platform also incorporates an FPGA with CPU for hardware and software reconfigurability. Application areas using this platform would be robotics, heterogeneous wireless sensor networks, and wireless video sensor networks [59].

The second contribution is a new architecture of the morphing bus for static hardware reconfigurability. This is a new bus architecture to interface devices without bus interface logic. A software tool for configuration maintenance is developed.

The third contribution is a software architecture for hardware/software dynamic reconfigurability for robotic applications in unknown environments. A new methodology to choose an optimal configuration using a configuration tree is presented and metrics for cost functions in the configuration tree are introduced. All the characteristics for each component in the system are discussed in terms of area, power consumption, and execution time. The characteristics can be used to dynamically reconfigure the robot at run time. A plant model can be changed during run-time dynamic reconfiguration.

Finally an adaptive control scheme called resource-adaptive control is introduced in this study and applied to a two-wheeled balancing robot. Resource-adaptive control is a new scheme to change control parameters at run time in an FPGA and to change the hardware architecture of an adaptive controller through FPGA reconfiguration. Modeling of the robot is established and Matlab simulation of resource-adaptive control is conducted with a state-space model of the system. LQR control was utilized to stabilize the robot. If there is any change in plant parameters which are causing instability of the robot, different control parameters are selected in order to meet design specifications by way of FPGA reconfiguration.

1.8 Thesis organization

The organization of this thesis is as follows.

Chapter 2 explains static reconfiguration and the concept of morphing bus. It deals with the tool developed to configure the morphing bus. Experimental setup to implement this concept is illustrated here.

Chapter 3 explains general concept of dynamic reconfiguration which provides reallocation of hardware and software resources at run time.

In chapter 4 resource-adaptive control is introduced with the case study of a balancing robot. Modeling, simulation, and implementation of this control were explained in detail. Dynamic reconfiguration is implemented for resource-adaptive controller.

Chapter 5 summarizes the ideas introduced in this thesis and concludes mentioning future directions that might be sought.

II. Static reconfiguration: morphing bus

2.1 Introduction

Numerous bus protocols exist such as I^2C, USB, PCI, VMEBus, etc. Some like USB achieve plug and play capability by storing interface logic on the device. Thus the protocol is able to query the device to gather interface information from it. Also logic is required for bus arbitration in case multiple devices need to be serviced at any given time. This standard bus is depicted in Figure 10.

The morphing bus exploits the static reconfigurability of the FPGA to provide an interface to modular sensors and actuators without bus interface logic. As seen in Figure 11, the morphing bus architecture gets rid of the need for interface and arbitration logic by providing a dedicated rather than a multiplexed bus for each device with the flexibility to swap the position of each device. The required data handshaking, data translation and signal processing is done on the FPGA.

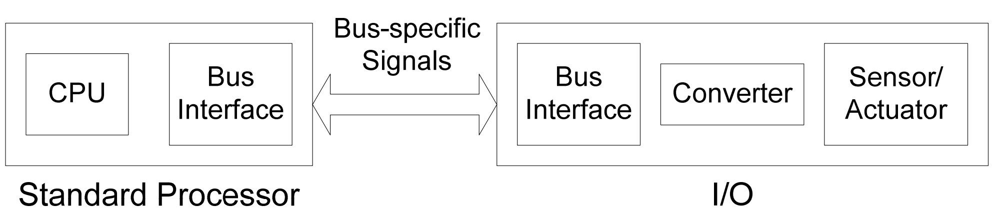


Figure 10. Standard bus.

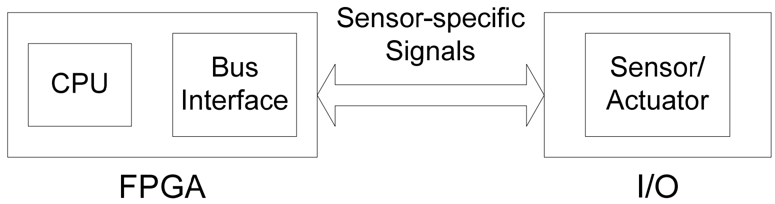


Figure 11. Proposed Morphing bus.

The bus is made up of circuit boards each of which is dedicated to only one or two sensors or actuators. The main emphasis is that the boards should be of low complexity and thus small size. Each board has electrical connectors at both ends. All the boards provide the same interface to the preceding and succeeding stages. Thus their position in the bus can be swapped. Each board uses as many bits of the bus as required to support the logic on that wedge and the remaining are fed to the next connector of the next stage which in turn does the same and so on.

The input lines to a wedge are used as follows: few initial lines are dedicated to power and ground. These are common to all circuit boards and run through all of them. Starting from the next connection the wedge circuitry uses as many I/O pins as it requires. The remaining lines are shifted to the output connector such that the unused lines are now immediately after the power lines. Figure 12 shows an example of how the assignment of I/O of the FPGA takes place as boards with different functionalities are added.

In Figure 12(a) the bare FPGA base board is shown. This is the heart of the system and in our case consists of only the FPGA and supporting hardware along with a connector to start off the bus. Figure 12(b) shows how the bus starts off. A camera plugged into the FPGA uses as many pins are required and the rest are transferred to the start of an output connector. The motor driver board uses 2 of these lines and passes the remaining in a similar fashion. Thus the FPGA pins are assigned sequentially in the same order that the devices are being plugged in. If the positions of two circuit boards in the chain are swapped, the pins of the FPGA connected to each device will differ but overall the same pins will be used.

The morphing bus is designed for use in the TerminatorBot [3] and its structure is shown in Figure 13. Because of the shapes of the wedges, when they are stacked up they take the form of a spiraling staircase. To provide support to this structure mechanical reinforcements are provided. Air is blown from the base upward, which follows the path along the spiral, cooling the ICs on every wedge. The number of devices that can be connected in the morphing bus architecture is limited by the number of available pins routed from the FPGA through the wedges, since each board has a dedicated connection to an FPGA pin. This is ultimately determined by size of the connector that can fit on each circuit board which in our case is limited by the size of the robot this system is being used in. Also a large portion of the wedge is taken up by the pass through routing of the unused lines, which again restricts us. However this is acceptable, since although this places an upper limit on the number of devices, I have the great advantage of being able to do without interface and arbitration hardware on the devices plugged in. Thus they can be very small, ideal for deployed field robots.

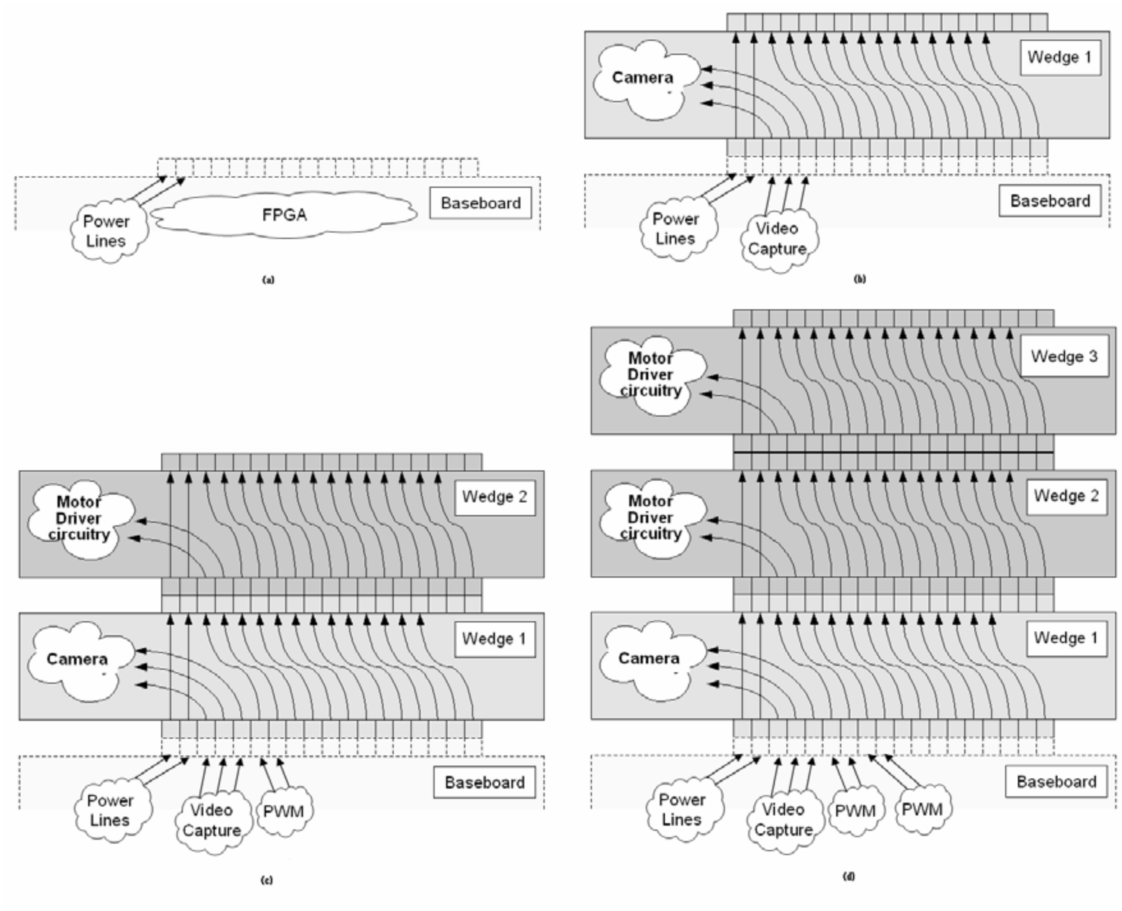


Figure 12. Wedge diagram for morphing bus. (a) The FPGA base board. (b) When the first circuit board is plugged into the base board, it uses some pins for the component supported and the rest are routed through. (c - d) successive boards are plugged into previous ones, forming a chain and all having direct connections to the base board FPGA.

Another concern is that the boards are not hot swappable i.e. they have to be plugged in and the device has to be configured before the system is turned on. This leads to complexity of configuring the system prior to deployment, and dealing with module replacement at run time. To simplify system configuration, a tools that takes in the order of the devices and HDL interface descriptions of each and automatically generating a top level file and a corresponding pin configuration file has been developed. These auto generated files can be used in the place and route process.

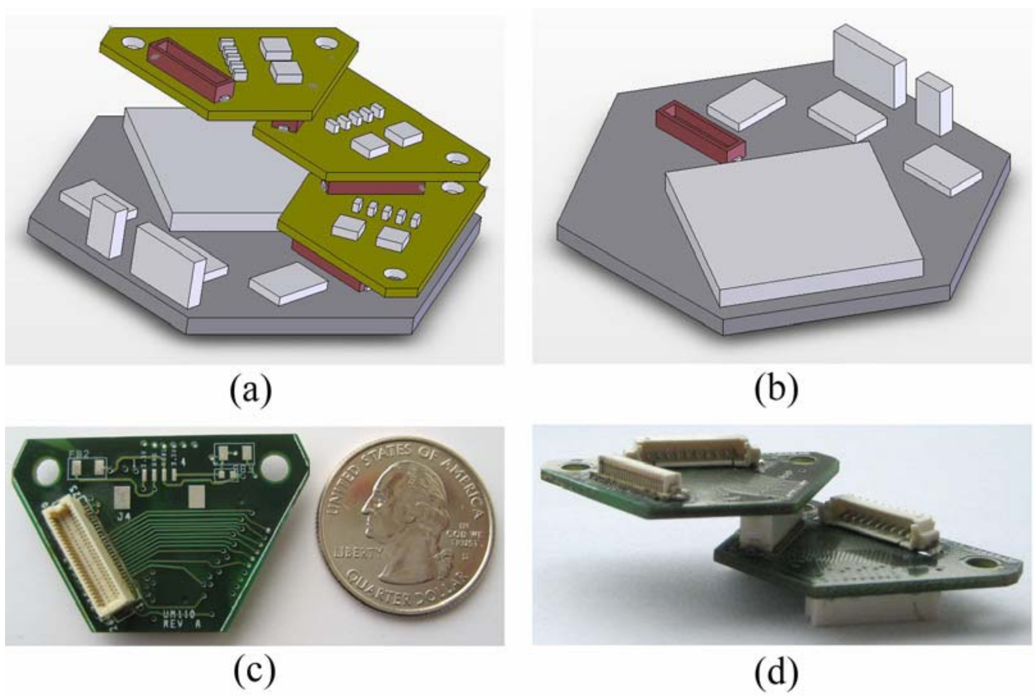


Figure 13. TerminatorBot Morphing bus spiraling structure. (a) One wedge is connected to the base board, starting off a chain where every wedge is connected to the previous. (b) FPGA base board. (c) A single cheese wedge. (d) Two cheese wedges stacked one above the other.

2.2 Tool for bus configuration

Plug and play based peripherals allow easy and quick system setup. However the devices on the morphing bus do not use interface or arbitration logic, and thus do away with the extra circuitry that allows the host to query the device. Due to this the control program has no way of identifying the device type. This knowledge is essential as the FPGA routing and pin assignment depends on it, to ensure that the appropriate module is interfaced to the sensor/ actuator.

A software configuration tool has been developed to support the morphing bus concept. The tool has a database containing a library of modules in VHDL or netlist format which have been individually compiled and tested. The devices connected to the bus prior to deployment, and the order in which they are to be connected can be selected. Then the top-level .vhd and .ucf files are generated. It automates the cumbersome task that would have to be performed manually otherwise.

The tool is only concerned with the interface between the bus and the module instantiated in FPGA logic. Thus even propriety FPGA cores can be included to handle processing requirements of a device if I know the core interfaces to the external system.

2.3 Experiments and observations

In order to verify the functionality and the feasibility of the morphing bus, a prototype system as shown in Figure 14 has been built. A Xilinx ML310 development board served as the computing platform and base board. It contained a Virtex-II Pro™ FPGA. For the prototype system a 20 bit wide morphing bus is used. The devices that I supported were a digital camera, two motors with hardware PID position control, some LEDs and switches on boards to simulate other possible I/O combinations. One board consisted only of LEDs to simulate an output only board on the morphing bus, a board of only switches simulating an input only device, and one consisting of LEDs and switches for I/O. The motors had optical encoders, the outputs of which were fed into the FPGA, and after processing signals are sent to the motors to control their positions. Thus I feel that I had a rigorous setup to test the functioning of the bus. The camera uses commands from the embedded PowerPC core to configure it, and then streams data into the FPGA which handles the data synchronization and stores image data to memory. Image processing algorithms could potentially be implemented.

Detailed information about hardware PID algorithm and schematic circuit diagram for each module are shown in Appendix B [40]. It describes block diagrams of parallel and serial implementation for hardware PID algorithm. Schematics of LEDs, switches and motors are also presented.

I connected the modules in various permutations. Then using the automated tool I generated the top level VHDL module. Using this I programmed the FPGA and tested that all the devices in the bus were working as expected for the different device orderings. The tool freed up a lot of time and effort that it would otherwise have taken to configure the bus.

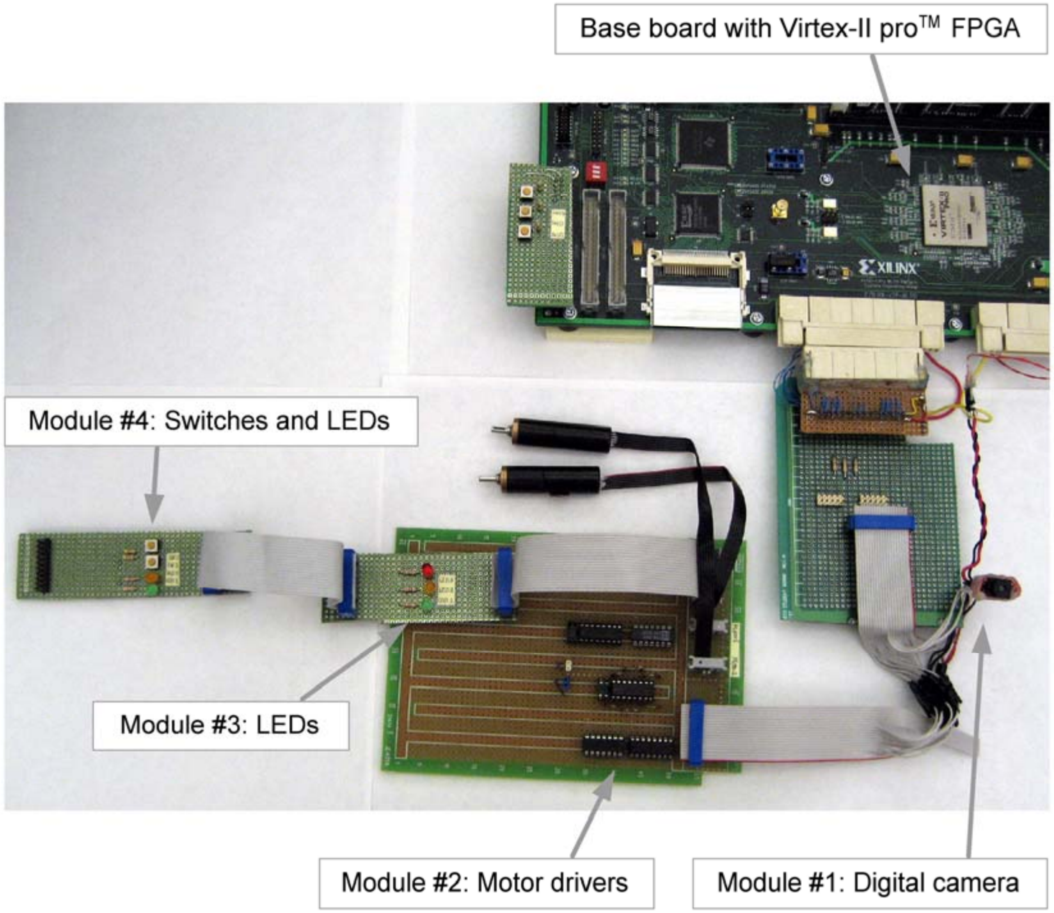
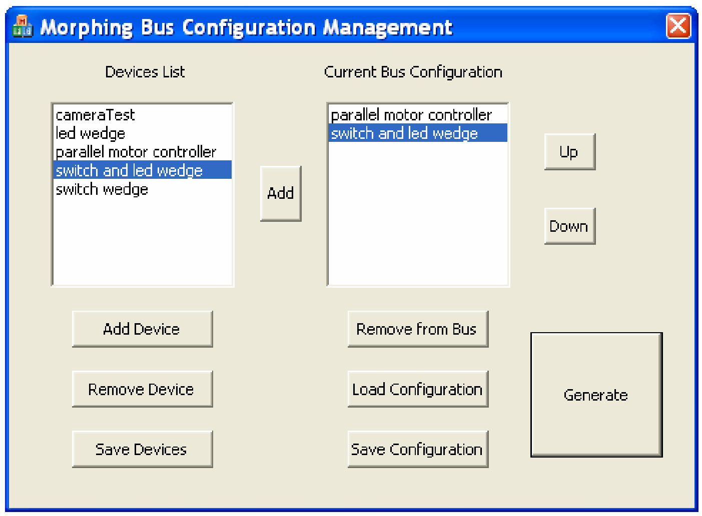


Figure 14. Morphing bus prototype.

The software for bus configuration tool is intuitive and easy to use. As shown in Figure 15 it consists of two panels. The one on the left is the list of available devices that can be connected to the robot, and the list on the right is that subset of those that are actually connected to the bus. Modules can be selected from the database of available devices and added to the current bus configuration. New modules can be added to the database, or old modules can be removed, and the modified list can be saved to be loaded the next time at startup. Similarly different bus configurations can be saved and loaded.

Examples of two configurations are shown below. In Figure 15 we see two modules have been selected and partial code is shown here, which was automatically generated for this configuration.




```
entity toplevel is
port(
FPGA_LCD_DIR : out std_logic;
tl_Clk : in std_logic;
tl_FPGA_LCD_DIR_1 : OUT std_logic;
tl_mainbd_sw_1 : in std_logic_vector(2 downto 0);
tl_mainbd_led_1 : out std_logic_vector(2 downto 0);
tl_EN_M1_1 : out std_logic;
tl_PWM_M1_1 : out std_logic;
tl_EncA_M1_1 : in std_logic;
tl_EncB_M1_1 : in std_logic;
tl_EN_M2_1 : out std_logic;
tl_PWM_M2_1 : out std_logic;
tl_EncA_M2_1 : in std_logic;
tl_EncB_M2_1 : in std_logic;

tl_mb_led1_2 : out std_logic;
tl_mb_led2_2 : out std_logic;
tl_mb_sw1_2 : in std_logic;
tl_mb_sw2_2 : in std_logic
);
end entity toplevel;
```

Figure 15. Morphing bus configuration management tool and Auto-generated code from the current configuration.

Devices are listed in the code in the order they are in the bus configuration list. A new name to be used at the top level is generated for every port signal of the modules in the list. This is to ensure that there are no conflicts if a device is used multiple times – e.g. if more than one motor or camera is used. The name consists of the original port name prefixed with “tl” for top level and suffixed with a number corresponding to its position on the bus. The pin mapping .ucf file as well as the port maps are also simultaneously written.

III. Dynamic reconfiguration

3.1 Introduction

Partial dynamic reconfiguration is defined in [60]: “An important feature in the Xilinx Virtex™ architecture is the ability to reconfigure a portion of the FPGA while the remainder of the design is still operational. Partial reconfiguration is useful for applications that require the loading of different designs into the same area of the device or the flexibility to change portions of a design without having to either reset or completely reconfigure the entire device.”

A reconfigurable computing platform for a small-scale resource-constrained robot is developed. Small-scale robots impose constraints on resources such as power or space for modules, but they still require great functionality to do challenging tasks. The reconfiguration enables them to choose a suitable configuration for different tasks. Also, Quality of Service (QoS) such as task success rate or execution speed can be improved through reconfiguration.

I propose a software architecture for hardware/software dynamic reconfigurability which provides for the reallocation of hardware and software resources at run time in the case of environmental change or different task.

Here are some examples for TerminatorBot when dynamic reconfiguration is required. Suppose navigation is being done with a camera, but it suddenly becomes very foggy, the camera becomes useless. Then the hardware resources previously set aside for the camera can be reconfigured for other sensors to compensate for the loss of sight through reconfiguration. Another example is when a robot is manipulating an object, it can come close to the object using visual servoing. After reconfiguration, it can use, for instance, force servoing to manipulate the object.

3.2 Platform for dynamic reconfiguration

In Figure 16, a reconfigurable computing platform is shown. An embedded processor monitors the system performance and manages reconfiguration through a reconfiguration controller. Fixed logic is determined by the first downloaded FPGA bitstream. An external memory contains the database of component characteristics of resource utilization.

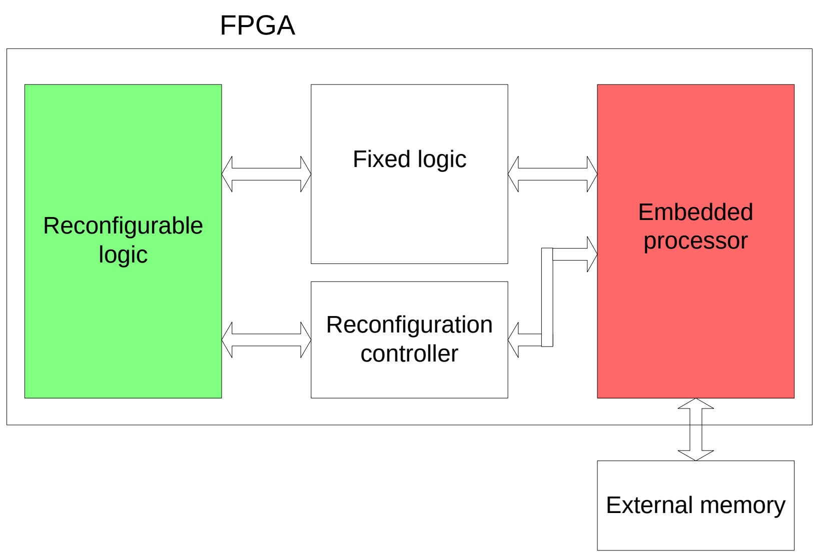


Figure 16. Reconfigurable computing platform using an FPGA.

Figure 17 shows the flow chart of a dynamic reconfiguration scheme. From the result of an initial static configuration, the system is already configured to have a limited number of sensors and actuators. Given a task goal and an initial configuration, the robot will execute the corresponding task. The embedded processor will keep monitoring the status of the robot. When there is a need for reconfiguration, because of an environmental change or new task, a reconfiguration controller will try to find a candidate configuration to meet the task goal. The database of characteristics, which contains hardware and software resource utilization is used to determine valid configurations. When evaluating a potential configuration, metrics are required to test the configuration to see if it meets a task goal. Then, reconfiguration with a new partial bitstream will be performed.

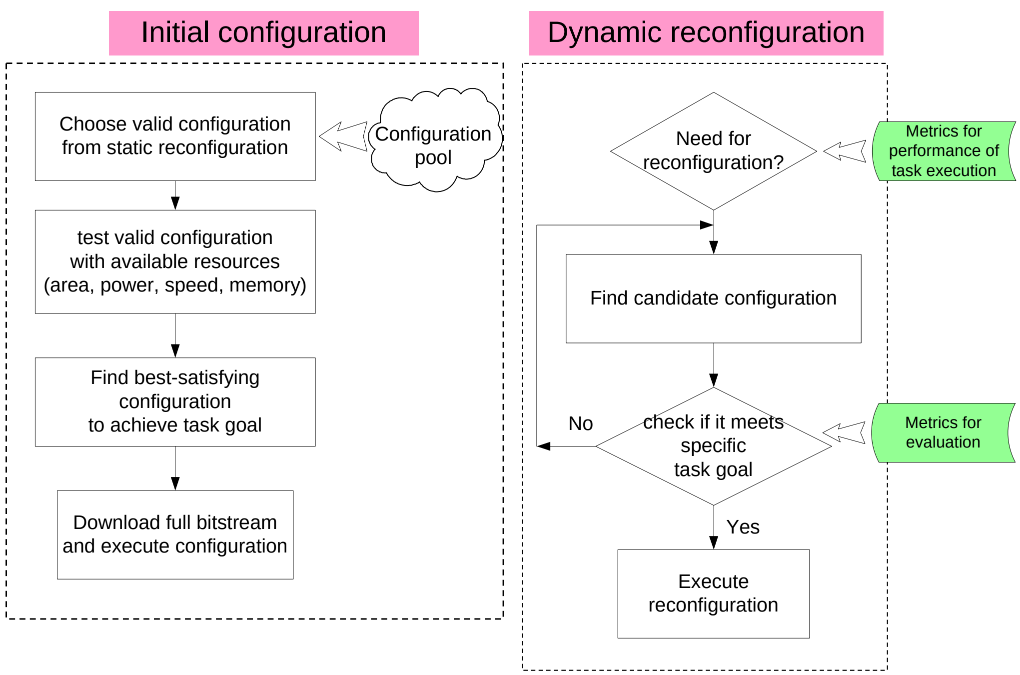


Figure 17. How to reconfigure.

The metrics for evaluating a given configuration, which is shown below, will be the function of various elements including task success rates of the algorithms, execution speed, robustness of the algorithms, etc.

$$(evaluation) = F\{(task\ success\ rate), (execution\ speed), (robustness\ of\ algorithm),\ ...\} \quad (1)$$

At the evaluation stage, the power consumption of each module will be an important issue. Power characteristics in the database can be utilized. Also, battery life can be monitored to determine how much time is left to run the robot. Success rate of algorithms will be another issue. The performance and robustness of sensors will be yet another measure. But, all these metrics are highly task-dependent, so they need to be refined after certain tasks are determined.

3.3 Software architecture for dynamic reconfiguration

In a changing environment, dynamic reconfiguration is essential if a robot is to successfully perform a given task. A new methodology to choose a better configuration is described in the following.

Figure 18 shows a general framework to generate all possible candidate configurations. The robot could have a different global task goal in a different situation. The robot needs to have specific task goals, which can be termed a “local goal”, so that the potential configuration can meet these specific criteria. Top level blocks to achieve a global goal are sensors, actuators, and software algorithms, which are essential for the robot to execute any task. Solid lines are used for these *essential* blocks which are located under the global goal. Because of the resource-constraints, the robot can only use at any time a limited number of sensors and actuators. Even though the robot has capabilities and performance to equip all the sensors available, the robot could have limited sensors and actuators after the static reconfiguration. The dotted line below the sensor block indicates *multiple* choices of different sensors. Each sensor needs to have an actual hardware logic to interface the sensor and an algorithm to process the sensor data. The locomotion could be implemented either in PID hardware or in PID software and the robot may choose different gait motion to adapt to diverse terrains. The algorithms under the global goal mean the integration of sensors and locomotion. In summary, the robot could have many different configurations to accomplish the given task.

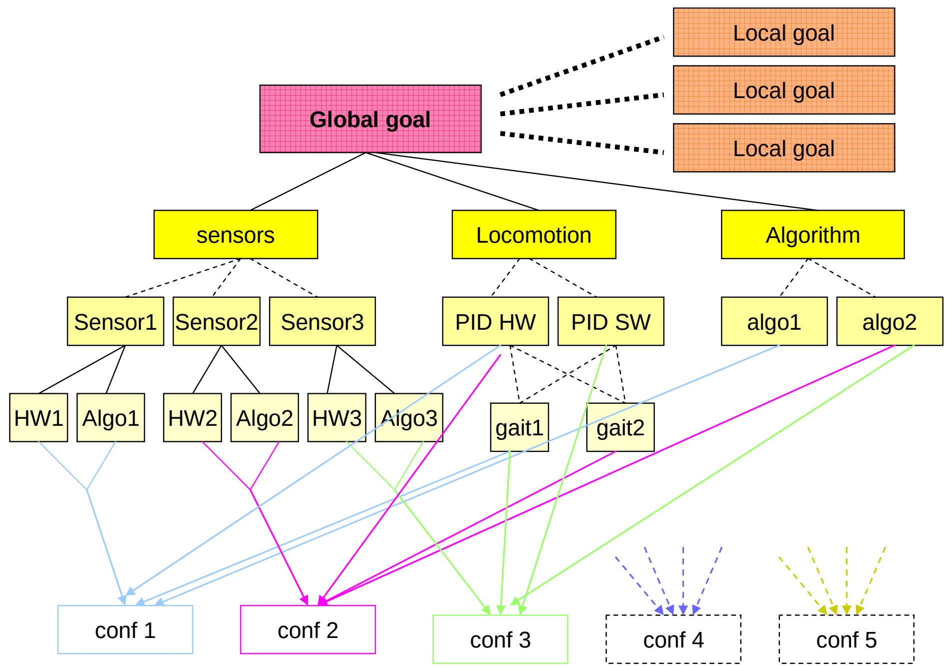
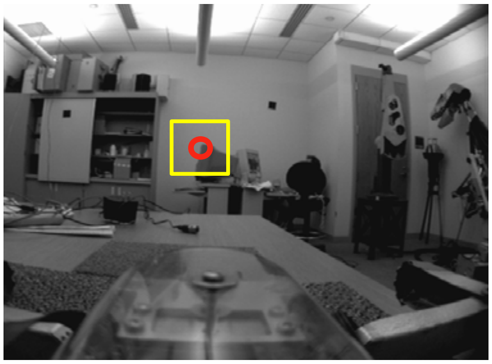


Figure 18. General framework of choosing configurations.

3.3.1 Configuration tree

The configuration of a robot will vary depending on a given task or a surrounding environment. Consider an example task which will help to understand the actual process of finding the optimal configuration. A task such as 'homing in a visual feature' is chosen in this case since an image processing module usually consumes considerable resources, thus creating a need for dynamic reconfiguration. Specific task goals would be related to locomotion power, success rate of the algorithm, and performance or robustness with different sensors.

In Figure 19, two methods of visual servoing are presented, which will be used in this example task. With a user-selected feature, a user or supervisor points to a tracking feature with a laser as shown in Figure 19 (a). A self-selected feature is shown in Figure 19 (b), a robot is trying to track an ellipse which is in back of the other robot.



(a) User-selected feature

(b) Self-selected feature

Figure 19. Methods of visual servoing.

Selecting an optimal configuration from all possible configurations is based on searching a tree-like graph which is similar to structure charts [61] because it is a top-down modular design and has lines that connect modules . Branches in the tree are composed of *method* branches and *resource* branches. In resource branches, resources can be allocated in the following ways:

1. static allocation
2. dynamic allocation
3. cooperative allocation

In static allocation, resources are determined at compile time when a robot is statically configured. In dynamic allocation, resources can be assigned or removed at run time by using dynamic reconfiguration. In cooperative allocation, a human is involved in the resource usage and resources can be allocated at run time.

Method branches represent all the possible ways to implement the given task and *one* method branch will be selected among all the method branches. Resource branches are comprised of all the available hardware and software implementations, and thus *multiple* resources can be selected.

The constraint in this tree structure is that branches must terminate at resources since combinations of available resources will eventually determine a candidate configuration. At the end of branches all the resources will have cost functions related to the resource utilization and functionality of the corresponding resources. Cost functions will be explained in detail in a later section 3.3.2.

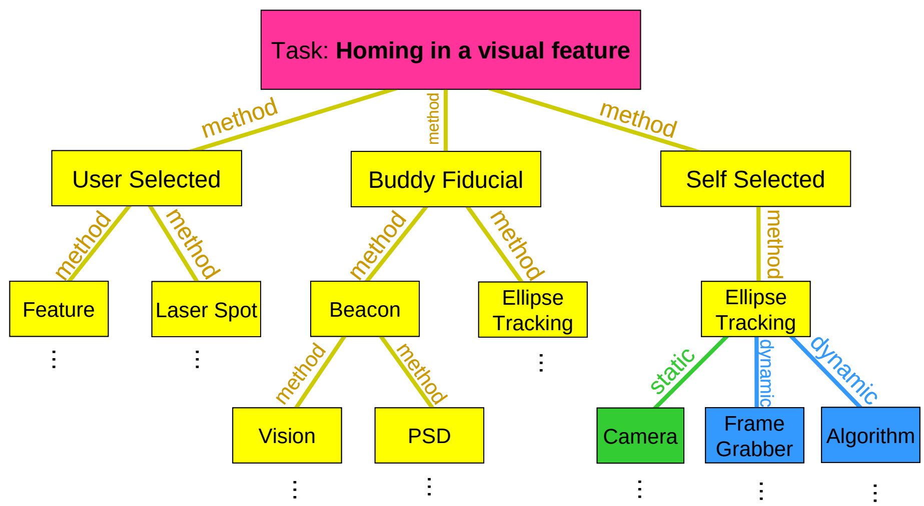


Figure 20. Example task of 'homing in a visual feature'.

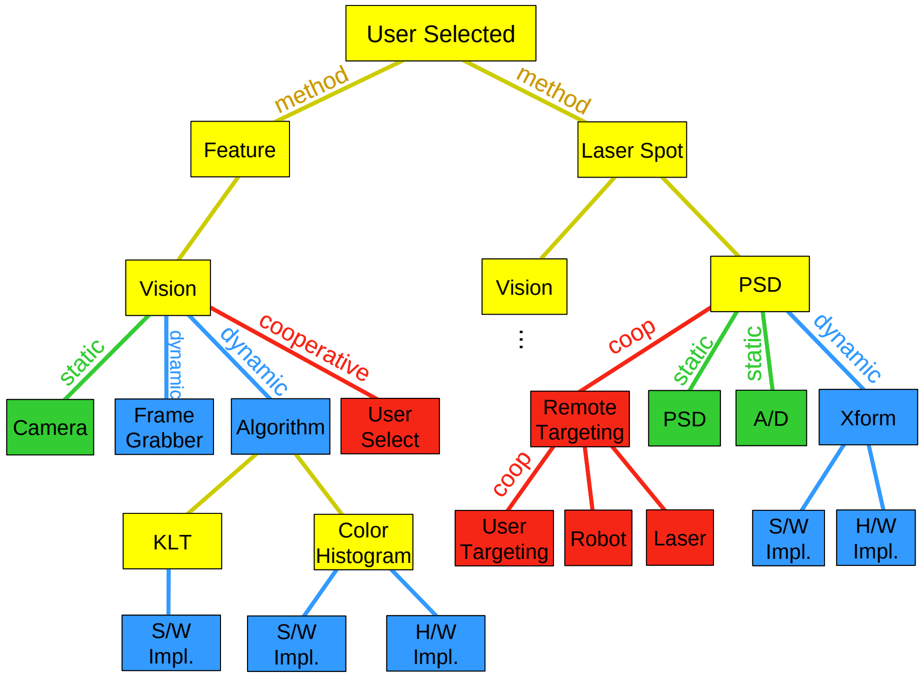
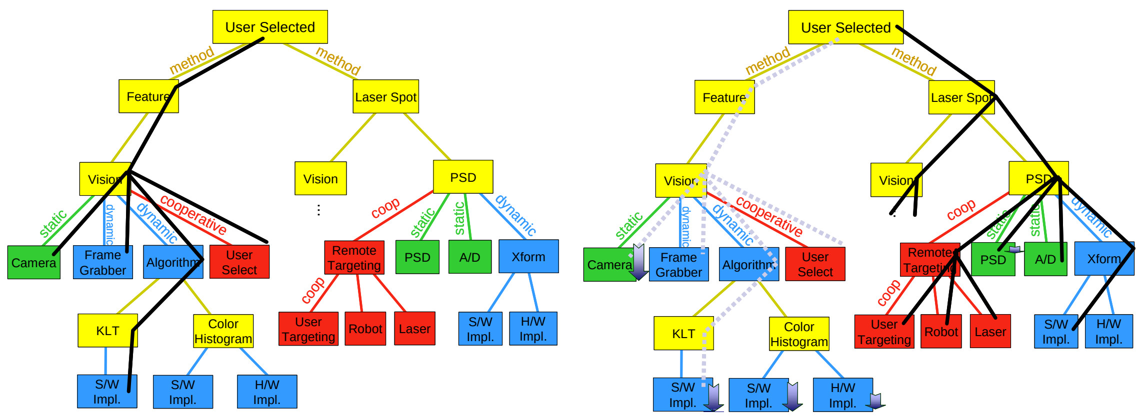


Figure 21. Configuration tree with 'user selected' method.

Figure 22 displays a possible example of switching to a different configuration. In Figure 22 (a), a robot is using a KLT algorithm [62] for visual feature tracking. When there is any environmental change and the camera is not functioning properly, the robot can reconfigure the FPGA to use a laser spot for tracking as shown in Figure 22 (b).



(a) use KLT for visual feature tracking

(b) use laser spot for tracking

Figure 22. Different configuration for visual servoing.

Using the resource utilization and potential functionality of each component, cost functions for each resource node are calculated first. Then, the optimal configuration will be chosen by searching the tree with minimum cost functions.

The detailed bottom-up procedure to determine the cost values for each node in the configuration tree is shown in Figure 23. The procedure is as follows:

1. Start assigning cost values for hardware or software implementation at the end nodes (C1, C2, C3, …, Cn), where n is the total number of hardware or software implementations.

2. Label cost values for resources at each end node (R1, R2, R3, …, Rm), where m is the number of available resources.

3. Transfer cost values to the upper node and combine cost values.

4. When combining cost values, add all the resources to cost values if there are existing resources which are located at the same level of the tree.

5. Finish transferring cost values when they reach the top nodes.

6. The optimal configuration will be chosen by examining the cost values at the top nodes.

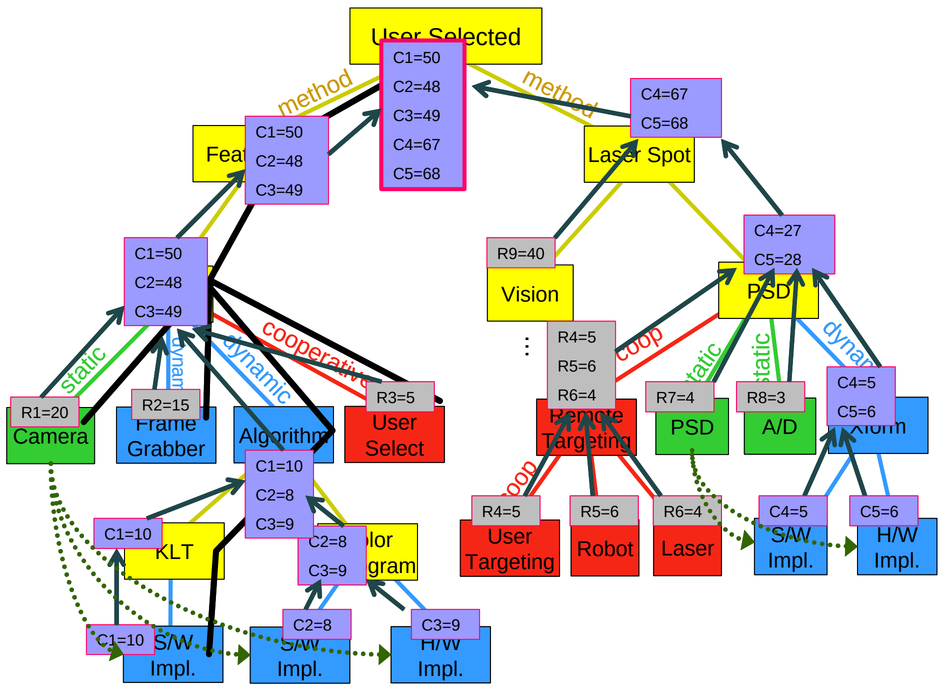


Figure 23. Procedure to determine cost values in a configuration tree.

3.3.2 Metrics for the cost function

The functionality of each device will vary with a changing environment. Dotted lines in Figure 23 indicate how the performance of a device affects the functionality of the implementation and the cost values for the corresponding implementation will be dynamically updated at run time. Thus, cost values are functions of functionality and resource utilization measures.

In order to search the configuration tree, cost functions for each node need to be assigned. The cost function not only includes resource utilization, but also includes functionality. If resource utilization is only used for the cost function, the chosen configuration may not achieve the task goal.

Consider, for example, a camera. A camera requires use a lot of resources, such as a frame memory or image processing algorithms, so that it will have a high cost value of resource utilization. But a camera may be the best choice for feature tracking so that it will have the lowest cost value of functionality. Therefore, cost functions will have the following relationships.

$$(cost\ \ function) = F\{(resource\,utilization), (functionality)\} \tag{2}$$

$$(resource\,utilization) = G\{(power), (area), (speed), (memory), \ldots\} \tag{3}$$

$$(functionality) = H\{(feature\,tracking), (laser\ detection), (gas\ detection), \ldots\} \tag{4}$$

The functionality value will vary dynamically depending on the situations. For example, when it's foggy, we need to find another configuration. This means that a camera is not functioning well for feature tracking. So, the functionality value of the camera will be a higher value for this foggy situation.

Some resources have strong relationships with functionality. For example, a camera will be very effective at feature tracking, but a gas detector can not detect visual features. Some of the resources, such as human, will not have concrete characteristics of resource utilization.

There are still challenging issues of quantifying the functionality of an implementation such as feature tracking functionality of the KLT, or laser detection functionality of a position sensitive detector (PSD).

In summary, all of these characteristics need to be tabulated in order to determine the cost functions in a particular configuration tree.

3.4 Experiments

3.4.1 Experimental platform

As a prototype system for the reconfigurable computing platform, experimental platform is introduced and DC motor control is performed. The experimental FPGA based system is shown in Figure 24. As shown in this figure, the required components of a complete system for motor control include a trajectory generator, a PID module, a PWM module, an amplifier and motor, a shaft encoder, and an encoder interface. The trajectory generator is implemented in software, the PWM module and encoder interface is implemented on the FPGA, and the amplifier, motor, and shaft encoder are external to the system. The PID module, which is the focus of this prototype system, is implemented both in hardware on the FPGA, and for comparison, in software.

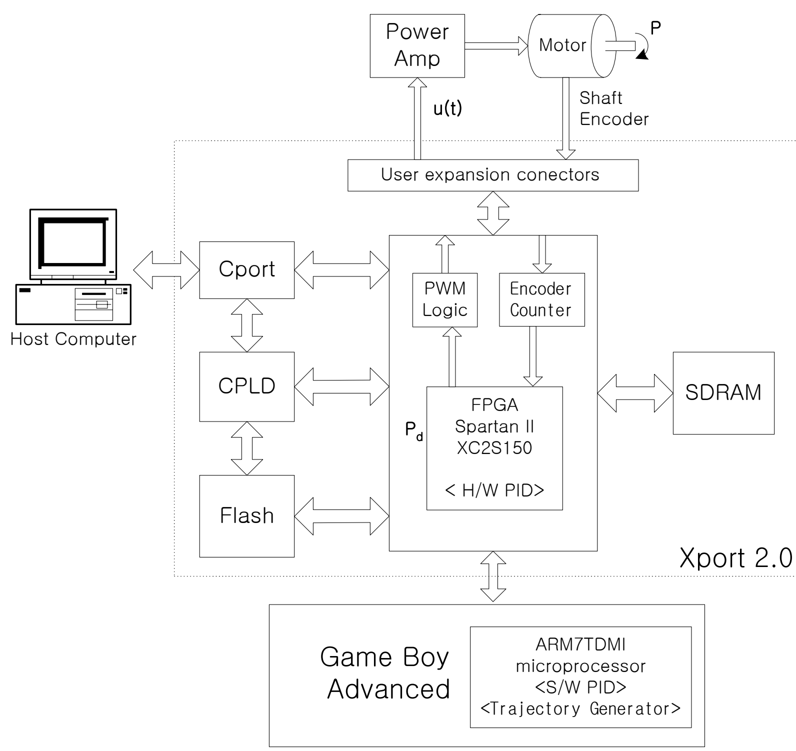


Figure 24. FPGA experimental system.

3.4.2 Functional test

A performance evaluation is meaningful only after the design is verified as functionally correct. Each PID hardware design, which is shown in Appendix B, were implemented and used to perform step response control of a motor. Additionally, a software implementation was developed and tested. The same parameters and sampling period were applied to the hardware PID implementations in the FPGA to perform the step response control tests. Experiment results of step response control for all designs are shown in Figure 25.

To test motor control for each design, the motor was set to an initial position of 1000, then a desired position command of 1200 was issued. From Figure 25, the horizontal dashed line is the desired position, while the other curves are the real responses sampled from the encoder counter. The results show that all the designs performed correctly and similarly. Response speed is fast, overshoot is small, and static accuracy is high. The average rise time is 30.32 ms and the standard deviation of the rise time is 0.7451. The steady state error is 0.

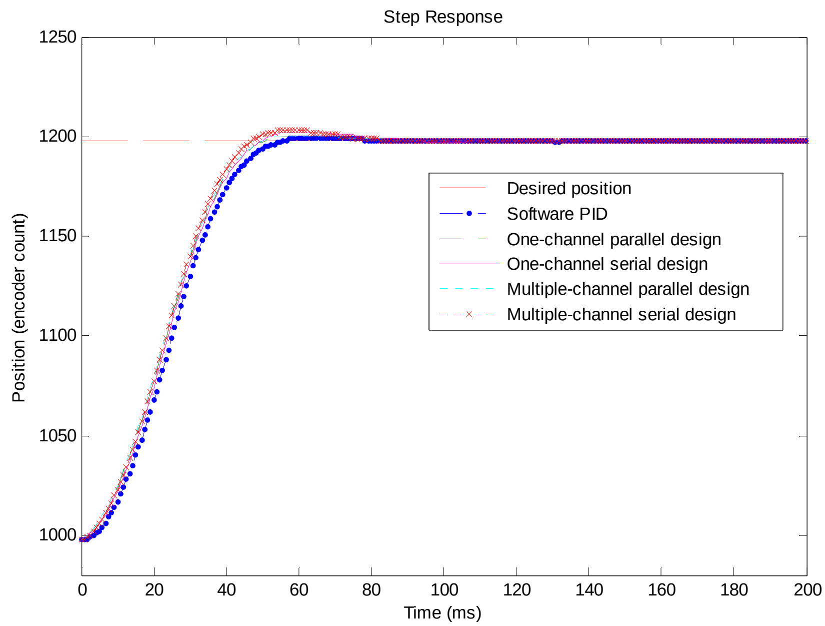


Figure 25. Step response control experiment results.

3.4.3 Performance test

PID hardware design was implemented and used to perform step response control of a DC motor. Additionally, a software implementation was developed and tested. Once the correctness of the designs was verified, performance was analyzed. Xilinx provides a variety of performance analyses, including resource utilization, speed, and power consumption, based on simulations of the hardware design, as reported in [40]. Performance was based on these reports.

Resource utilization for each design was measured and analyzed, which is shown in Figure 26. In one-channel and multi-channel serial designs, all arithmetic operations share one multiplier and one adder, while in parallel designs there are 3 multipliers and 4 adders. Because of this, serial designs have an obvious space advantage. However, some of this space savings is used up with additional control logic.

Area and speed are conversely related. The advantages in area requirements shown for serial design are countered by their disadvantage in speed. While the datapath in the serial design is shorter, thus the delay is shorter, more clock cycles are required. As expected, execution times for serial design were longer, which is shown in Table 1.

Experimental results on power dissipation are shown in Figure 27 and Figure 28. Power consumption is dependent upon the sampling and control clock frequency. Thus, to compare power performance, motor commands were generated and the PID module was run at various frequencies. The test data obtained in the step response experiments were used as input to the hardware simulation of each PID design. In multiple-channel designs, for the same sampling frequency of 0.12MHz and control clock frequency, power dissipation increases linearly as the number of channels increases. It was expected that for the same sampling frequency and execution time, the multiple-channel parallel based design would consume less power, because the clock frequency of the parallel based design is lower. For one channel, the parallel based design does consume less power; however, for a large number of channels, the parallel-based design consumes more power than the serial-based. The result also shows that for the same sampling frequency, the channel-level parallel design with a serial PID unit consumes the least power, but the area requirements of the channel-level parallel design rapidly exceed the capacity of the FPGA as the number of channels increases.

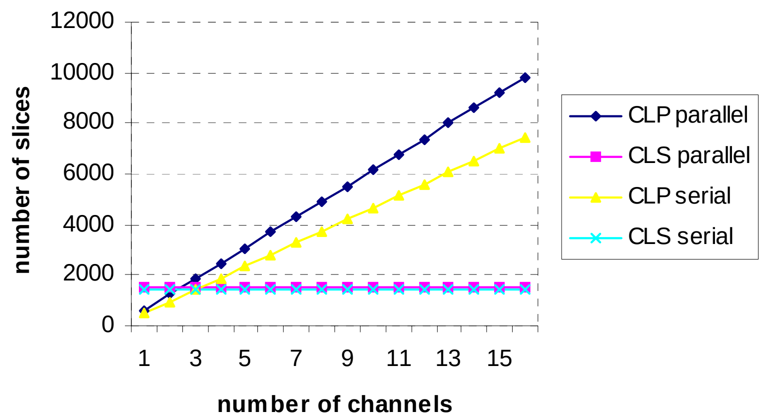


Figure 26. Area comparison of multiple-channel designs.

Table 1. Clocks and execution times of designs.

Designs	**One-Channel**		**Multiple-Channel (CLS)**	
	Parallel	**Serial**	**Parallel Based**	**Serial Based**
Clock period	44.270 ns (≈50)	29.146 ns (≈30)	48.955 ns (≈50)	29.816 ns (≈30)
# of cycles	1	4	2	6
Sample period	≈ 50 ns	≈ 120 ns	≈ 100 ns	≈ 180 ns

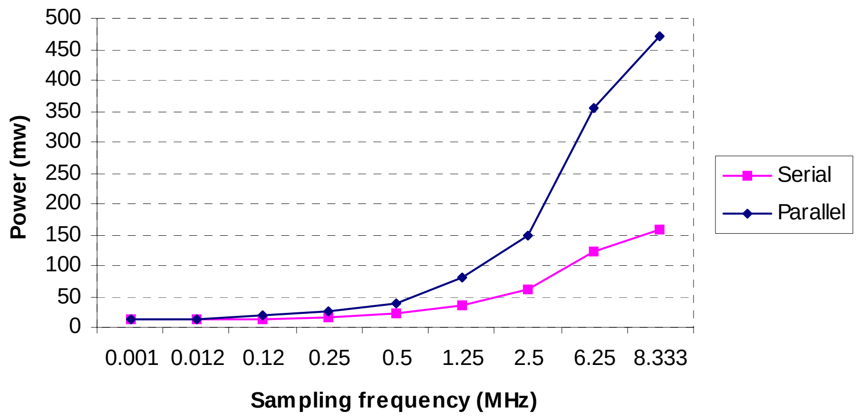


Figure 27. Power dissipation of one-channel serial and parallel design.

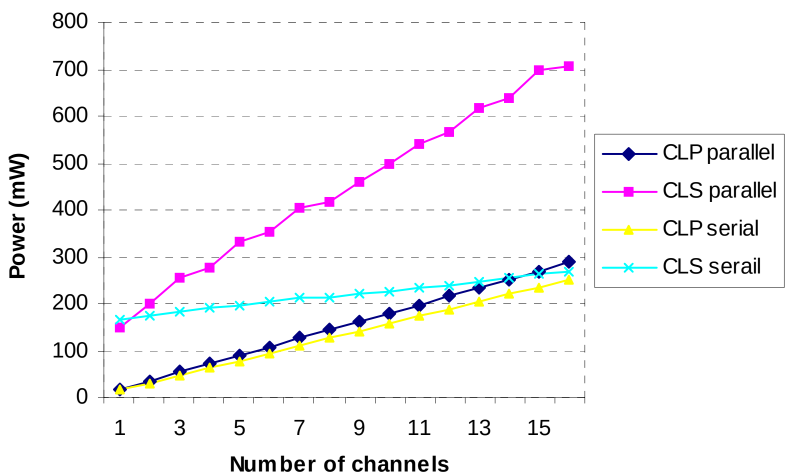


Figure 28. Power comparison of multiple-channel designs.

3.4.4 Discussion on performance analysis

In this experiment of prototype system, preliminary work was conducted to explore control system design for a resource-constrained robot based on an FPGA technique. Parallel and serial architectures of the PID control algorithm were designed and implemented for one-channel and multiple-channel architectures.

Performance tests show that for a small number of channels, channel-level parallel design with serial PID has the smallest area and consumes the least power. For more channels, the channel-level serial design with serial PID has the smallest area requirements and the channel-level parallel design with serial PID still consumes the least power, but the area requirements of the channel-level parallel design with serial PID increases very quickly.

In order to adapt different environment, robot doesn't need to choose one design to achieve one performance goal, for example, maximum power. On the contrary, robot could change designs or reconfigure the FPGA fabrics to adapt different situation considering all the trade-offs such as circuit area, execution speed, and power consumption, and robot could find optimal configuration for various condition.

IV. Application of resource-adaptive control

4.1 Introduction

An adaptive control scheme, called resource-adaptive control, is applied in this study for a two-wheeled balancing robot. In resource-adaptive control parameters are changed at run time. Linear Quadratic Regulator (LQR) control was chosen to stabilize the robot, since it is a full-state feedback controller and suitable to work with multiple-input multiple-output (MIMO) system. Two feedback signals of the robot, which are position and tilt, are used for the LQR controller. When there is any change in plant parameters which is causing instability of the robot, different Q or R values of the LQR controller are chosen in order to meet design specifications.

4.2 Plant modeling and state-space equation

The modeling of a two-wheeled autonomous robot is explained in Appendix C. The equations of motion for the wheels are obtained and the free-body diagram of the robot and chassis are depicted. The continuous state space equation and the transfer function of a robot position and an applied input toque are obtained.

Given the following values for each parameter written in MATLAB code, poles and zeros of the transfer function can be obtained from the open-loop transfer function, equation (38).

```
Mp = .5;       % Mass of the robot's chassis
Mw = 0.2;      % Mass of the wheel
```

```
km = 0.04;      % Torque constant
ke = 0.005;     % Back emf constant
ip = 0.0005;    % Inertia of the robot's chassis
iw = 0.0005;    % Inertia of wheel
Rm = .5;       % Motor Resistance
r = 0.03;      % radius of the wheel
L = 0.1;     % distance between the center of the wheel and the robot's
              % center of gravity
g = 9.8;       % gravitational accelerations
```

$$\frac{X(s)}{V_a(s)} = \frac{0.0147s^2 - 1.3067}{0.0044s^4 + 0.0024s^3 - 0.6152s^2 - 0.2178s} \tag{5}$$

, where $X(s)$ represents the robot position and $V_a(s)$ is the applied torque input.

The state space equation for the open-loop system is calculated as:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -0.5549 & 5.5612 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -5.0441 & 139.6469 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 3.3291 \\ 0 \\ 30.2648 \end{bmatrix}$$
$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{6}$$

The state space equation for a closed-loop system can be obtained with the following LQR parameters [63].

$$Q = \begin{bmatrix} 2000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 50 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad R = 1$$

$$K = \begin{bmatrix} -44.7214 & -15.7400 & 31.1845 & 3.0307 \end{bmatrix}$$

$$A_C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 148.9 & 51.8 & -98.3 & -10 \\ 0 & 0 & 0 & 1 \\ 1353.5 & 471.3 & -804.1 & -91.7 \end{bmatrix} \quad B_{cn} = \begin{bmatrix} 0 \\ -148.9 \\ 0 \\ -1353.5 \end{bmatrix} \tag{7}$$

$$C_C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad D_C = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The closed-loop transfer function can be calculated as,

$$\frac{X(s)}{V_a(s)} = \frac{-149s^2 + 13264}{s^4 + 40s^3 + 655s^2 + 4619s + 13264} \tag{8}$$

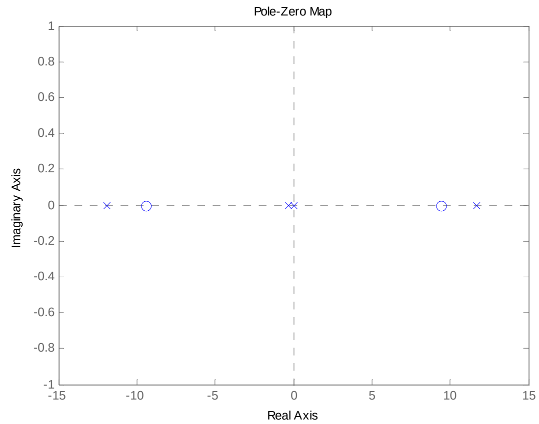
Table 2 shows the values of poles and zeros of the open-loop and closed-loop system with different values of mass. As shown in Table 2, the open-loop system includes one right-half-plane pole located at 11.7201 with initial mass. This means that the system is unstable with an open loop case. With increased mass the open-loop system has larger value of a pole located at 20.995 which means a larger oscillation frequency which might be caused by the bigger mass.

The open-loop system also has one negative pole at -0.3528 which represents the velocity of the robot, in addition to the negative conjugate pole. This pole located at left-half plane has smaller magnitude with respect to the increased mass, which means slower velocity of the robot.

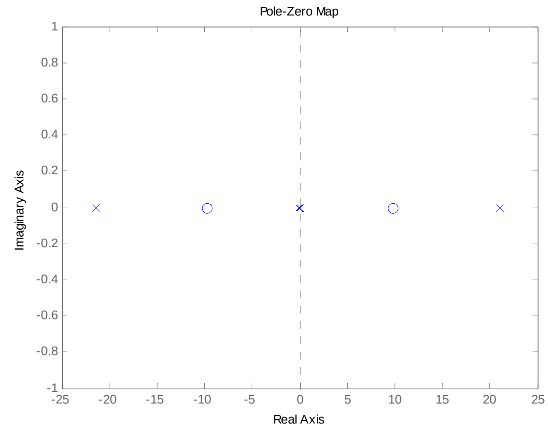
Table 2. Poles and zeros of the system.

	Initial mass, M_p = 0.5	Increased mass, M_p = 3
Open-loop system	Poles = [0 ; 11.7201; -11.9212; -0.3538] Zeros = [9.4388 ; -9.4388]	Poles = [0 ; -21.4290; 20.9950 ; -0.1183] Zeros = [9.8180 ; -9.8180]
Closed-loop system	Poles = [-13.9734 + 8.5762i; -13.9734 - 8.5762i ; -6.0266 + 3.6088i; -6.0266 - 3.6088i] Zeros = [9.4350; -9.4350]	Poles = [-21.8851 + 5.9720i; -21.8851 - 5.9720i; -4.1513 + 3.2441i; -4.1513 - 3.2441i] Zeros = [9.8245 ; -9.8245]

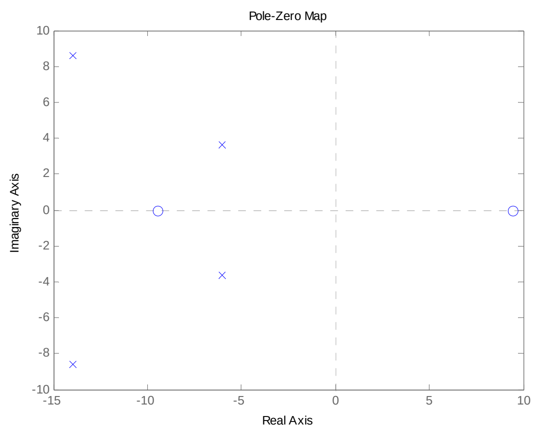
Pole-zero plot is shown in Figure 29. In the figure, the open-loop system has one pole in a right-half plane which means unstable nature of the robot similar to inverted pendulum. Introducing LQR controller, the closed-loop system relocates all the poles to a left-half plane and makes the system stable. One might try to use different techniques to balance the robot, for example, PID control, root locus method, or frequency response method, but those methods fail to stabilize the robot since they can just control one variable, such as, a tile angle not the robot's position [64].



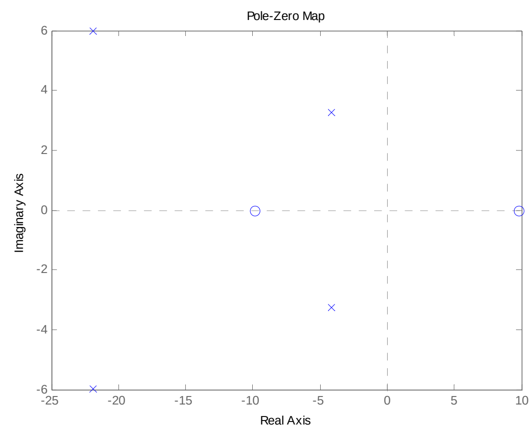
(a) open-loop system when $M_p = .5$



(b) open-loop system when $M_p = 3$



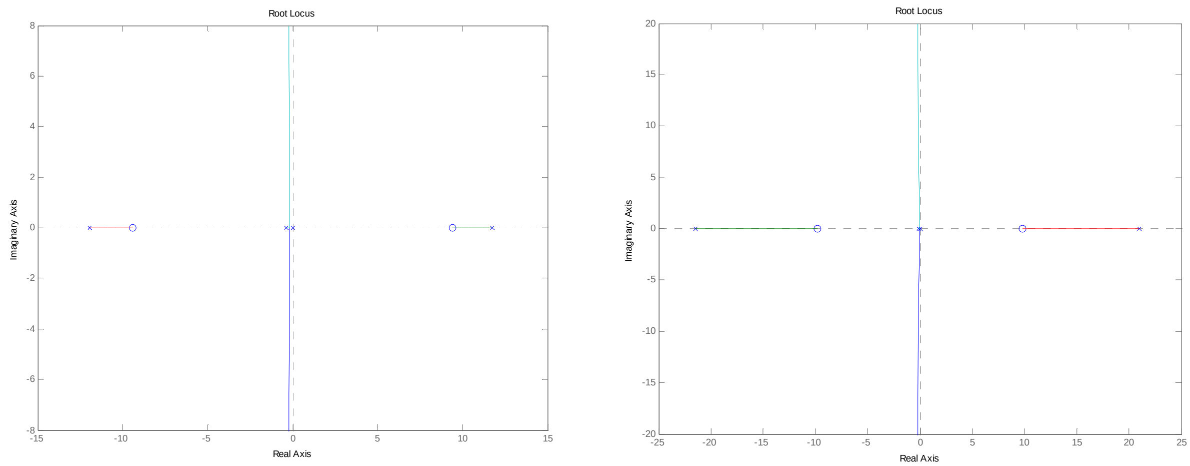
(c) closed-loop system when $M_p = .5$



(d) closed-loop system when $M_p = 3$

Figure 29. Pole-zero map for the system.

The root locus of an open-loop system with Mp of .5 shows that there is always one right-half-plane pole with a unity feedback closed-loop system (refer to Figure 30). This means that a simple unity feedback system with any proportional gain cannot guarantee stability for the system.



(a) open-loop system when $M_p = .5$

(b) open-loop system when $M_p = 3$

Figure 30. Root locus for the system.

4.3 Simulation of resource-adaptive control

MATLAB simulation of the proposed adaptive control scheme is shown in this section. As an example of a plant parameter change, the mass of a robot chassis is considered to be changed. A mass is usually an important factor in adaptive control since an aircraft or a space shuttle, for example, is consuming fuel during its flight and its mass would decrease. When there is any change in the plant parameter, the performance of the robot can be varying. In the case when extra mass is loaded on top of the robot, it is expected that since the center of mass has moved to an upper location, the robot would have more oscillation and longer settling time. The next figure shows the increase of the rise time and settling time for the robot position when the mass of robot chassis is increasing. This means that when there is any increase in the robot mass, performance criteria does not meet at some instance, which requires suitable control technique to recover to the original performance.

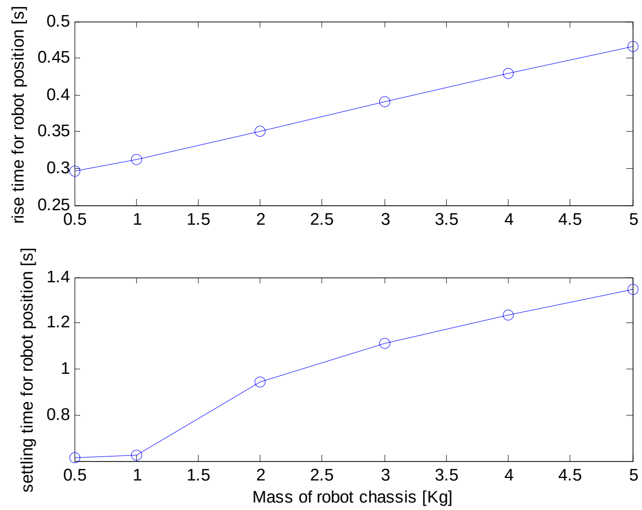


Figure 31. Rise time and settling time for robot chassis w.r.t. the mass change of robot chassis.

4.3.1 Simple adaptive control scheme

A simple adaptive control scheme is presented in order to deal with plant changes. Here is the simulation algorithm which performs LQR control and tries to adaptively modify control parameters when it detects any performance degradation. This simulation algorithm will be verified using MATLAB and then will be implemented in microcontroller and finally in FPGA.

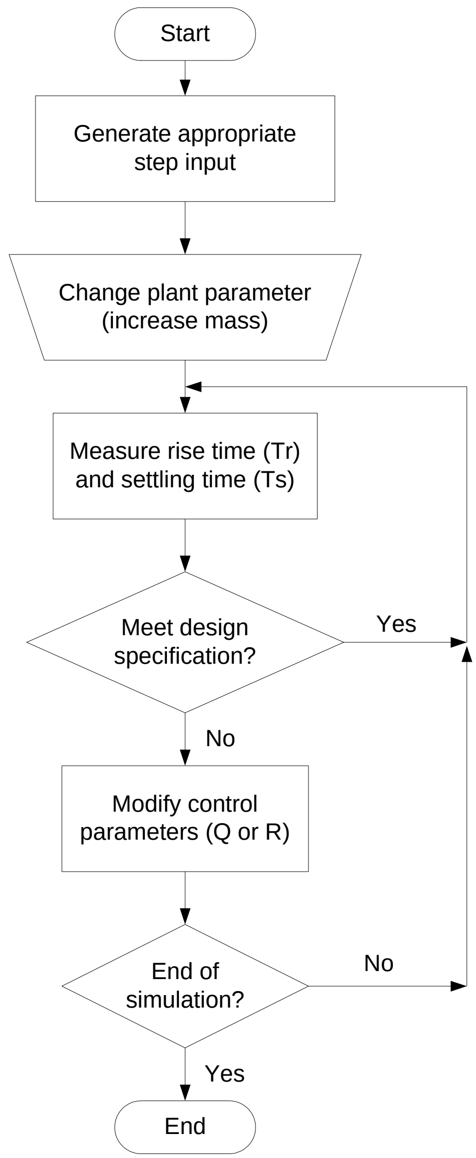


Figure 32. Flowchart of the simulation of adaptive LQR controller.

4.3.2 Simulation flow of adaptive LQR controller using MATLAB

Here are the detailed steps to simulate the LQR controller for this system using MATLAB software.

Performance criteria for this specific case were a rise time should be less than 0.4 second and a settling time needs to be less than 1 second. (Tr < 0.4 and Ts < 1.0)

1. Build state space model (A,B,C,D) for open-loop system

2. LQR controller for closed-loop system

   a. Assign initial values of Q, R matrix

   b. Run 'lqr' library routine to obtain K matrix

   c. Ac = A-B*K

   d. State space model(Ac,B,C,D)

3. Nbar calculation to adjust reference input

   a. Run 'rscale' user-defined routine to calculate Nbar

   b. Bcn = Nbar*B

4. Step response

   a. Generate Time (T), Input (U) matrix

   b. Run 'lsim' library routine using the model(Ac, Bcn, C, D)

5. Measure performance

   a. Run 'stepspecs' user-defined routine to find settling time(Ts) and rise time(Tr)

   b. If Ts or Tr does not meet criteria, change Q or R parameter and calculate settling time and rise time again.

In Figure 33, magnitude of the step input is changed every 6 seconds and system performance is measured and displayed. The mass of a robot chassis is changed from 0.5 Kg to 3 Kg at 4 seconds. At the time of 6 seconds, settling time is greater than 1 second which means system performance does not meet design criteria. Therefore, at the time of 8 seconds, R value of LQR controller is changed from 1 to 0.6, and then at the time of 12 seconds, the system shows reasonable performance which meets design criteria.

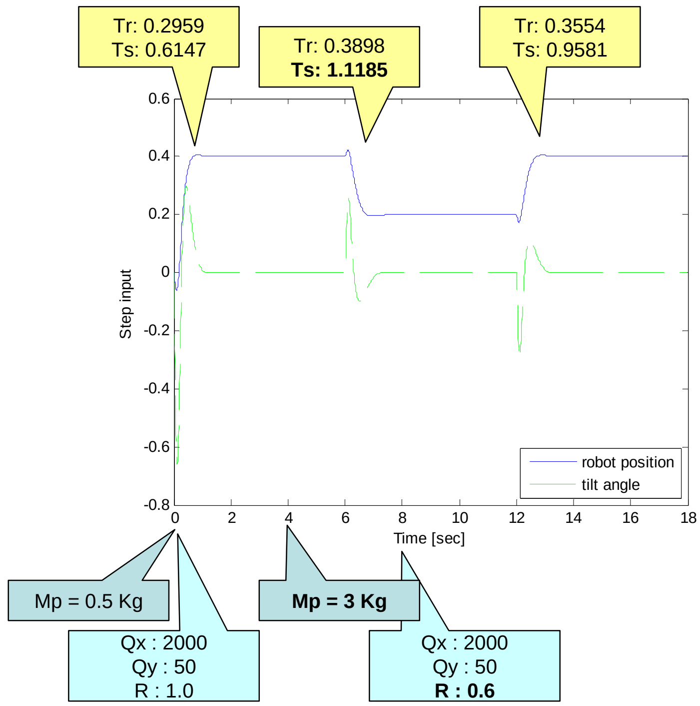


Figure 33. Graphical output of the step response of LQR controller when using MATLAB.

4.4 Solving algebraic Riccati equation using VisualC

Before implementing LQR control in a microprocessor, it was verified using VisualC software. In order to implement LQR control in a microprocessor, K vector needs to be calculated by solving an Algebraic Riccati equation [5, 65]. CLAPACK [66] is a C language version of LAPACK which provides routines for solving linear systems, eigenvalue problems, associated matrix factorizations, etc. CLAPACK, version 3.1.1 for windows, was used to solve an Algebraic Riccati Equation. An environmental setup to compile a source code in VisualC is explained in Appendix F. Source code to implement LQR control using CLAPACK library and VisualC software is attached in Appendix G and the result of execution is displayed in Appendix H.

4.5 Implementation in Atmel microcontroller

After fixing several coding errors, LQR control in a microprocessor was implemented. ATmega128 from Atmel [67] was chosen as a microcontroller since it is one of powerful 8 bit microcontroller and has enough features for this work. It provides the following features: 128K bytes of programmable flash, 4K bytes EEPROM, 4K bytes SRAM, four Timer/Counters with compare modes and PWM, 2 USARTs, 10-bit ADC, etc. Here is the functional block diagram of the software flow for the robot. Blocks within the dotted lines were implemented in Atmel microcontroller.

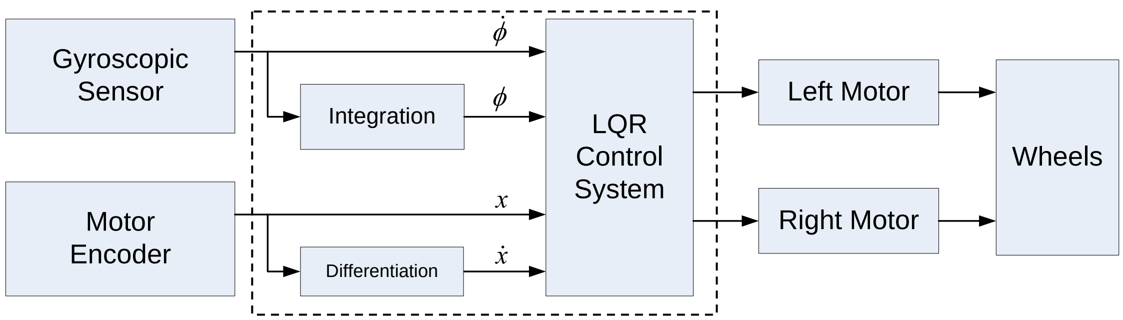


Figure 34. Functional block diagram of wheeled balancing robot.

4.6 FPGA reconfiguration

4.6.1 FPGA Implementation of resource-adaptive control

Hardware block diagram for resource-adaptive control using FPGA are shown below. While a

PowerPC (PPC) processor is continuously monitoring the performance of the robot and stability, gain matrices for an LQR controller will be updated by solving an Algebraic Riccati Equation. A plant model can also be attempted to be changed to a reduced-order model to achieve the required performance by reconfiguring the FPGA. Reconfiguring a whole controller can also be considered, but controllers usually tend to provide similar performance when the states of a system are given. Replacing a controller can be reasonable when there is any resource issue, for example, when one controller is occupying too much of FPGA areas and there is a need to free up some of logics.

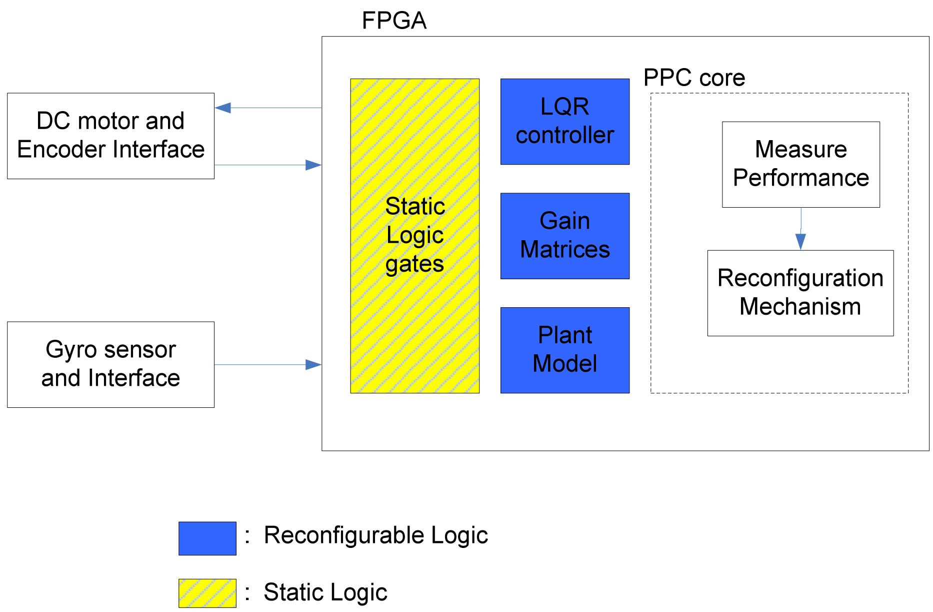


Figure 35. Basic hardware block diagram of FPGA implementation.

Here is an in-depth hardware block diagram to implement dynamic reconfiguration using FPGA.

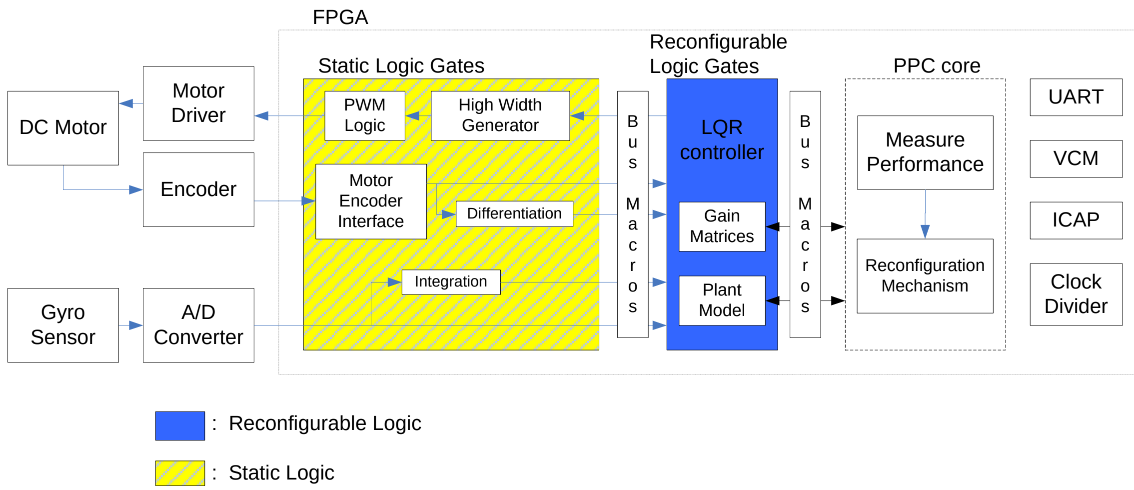


Figure 36. Detailed hardware block diagram of FPGA implementation.

Refering to Figure 36, the following are specific modules which are to be implemented in FPGA chip.

- Static logic

PWM Logic: This module generates pulse-width-modulated output signal to drive a motor.

High Width Generator: The output of an LQR controller is digital data. High width generator module calculates the width size of a high state pulse.

Motor Encoder Interface: This module builds digital signals of motor position from A and B phase of an encoder.

Differentiation: It can generate velocity information from motor position.

Integration: It can make tilt angle of the robot from angular velocity of a robot body.

UART: A serial communication module can be used to debug the board and communicate with a host PC.

VCM: A Virtex configuration manager will be used to configure the chip during a reconfiguration stage.

Clock Divider: This module generates slower clock from the main clock.

ICAP: Internal Configuration Access Port should be instantiated in order to use a reconfiguration feature in a Xilinx FPGA. ICAP provides a port internal to the FPGA for configuring the FPGA device.

- Reconfigurable logic

Gain Matrices: This module includes Q, R, K, and T matrices which need to be updated during reconfiguration. First, the LQR controller will load the Q and R matrices and calculate the K matrix. Then an old K matrix will be updated with the new one. T matrix is used to calculate new state vectors with a reduced-order plant model.

Plant Model: In the adaptive LQR controller, a plant will be modeled with various orders. Depending on the decision of reconfiguration mechanism, different order of a plant model will be chosen and an FPGA logic will be reconfigured with corresponding order of a matrix.

- Power PC (PPC) microprocessor core

Measure Performance: This routine will measure the performance of a robot which includes a rise time and a settling time. Depending on the criteria, it will generate a suitable command to the reconfiguration mechanism whether to change values of Q, R, and K, or to change the order of a model.

Reconfiguration Mechanism: This module will actually perform reconfiguration according to the output of performance-measuring module.

In order to achieve partial reconfiguration the following software tool chains are required to be installed.

- Xilinx ISE 9.2i SP4
- Xilinx Partial Reconfiguration Patch for ISE 9.2i SP4
- Xilinx PlanAhead 10.1i
- Bus Macros
- Xilinx Platform Studio EDK 9.2i
-

Xilinx PlanAhead software is used to floorplan a design, assign partial reconfiguration region, and then implement partial reconfiguration flow by using ExploreAhead feature of PlanAhead.

4.6.1.1 Integration module for implementation

Here is a block diagram for integration which will be used in the design of FPGA implementation of resource-adaptive control. Trapezoidal Rule was used to implement the hardware version of numerical integration.

$$\int_a^b f(x)dx \approx (b-a)\frac{f(a)+f(b)}{2} \tag{9}$$

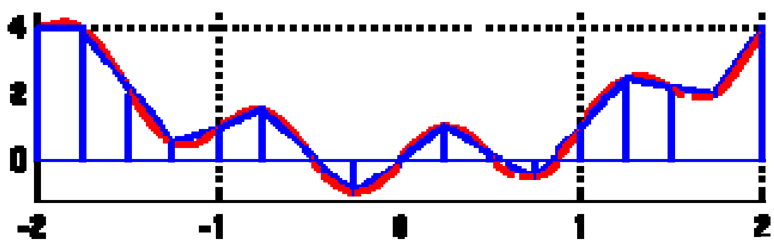


Figure 37. Numerical integration using trapezoidal rule is expressed in discrete-time domain.

$$y_n = y_{n-1} + \frac{(b-a)}{2}\{f(a)+f(b)\} \tag{10}$$

, where y_n indicates the numerical integration at the index of n. a and b are the points in a small trapezoid.

Hardware block diagram of numerical integration is shown in the following Figure 38. This block diagram will be implemented in an FPGA chip. The current value of gyroscope sensor will be added with previous one. The result will be multiplied by $h/2$ which is small time division. h is the sampling period. The result of multiplication will be added with previous result of multiplication. Finally the calculated output will be the numerical integration of gyroscope sensor data which represents tilt angle.

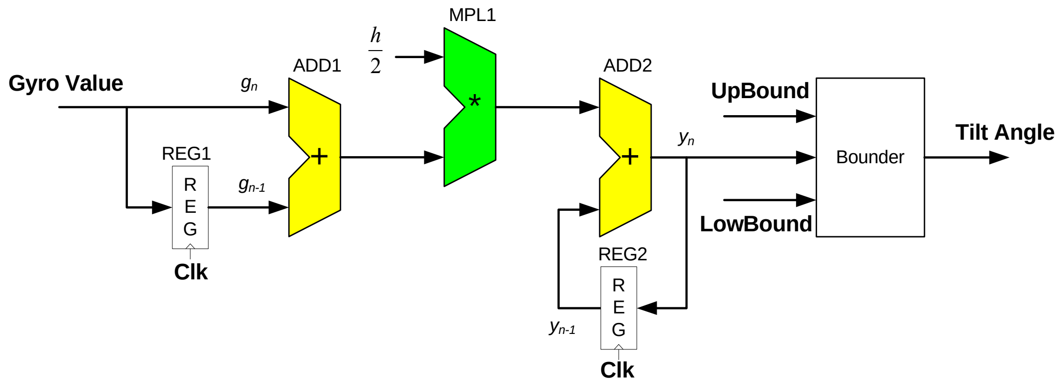


Figure 38. Hardware block diagram for implementing numerical integration.

4.6.1.2 Differentiation module

A simple differentiation using two points was used in this implementation.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{11}$$

, where $f(x)$ represents position value from motor encoder. h is the sampling period for each position value.

In the following numerical differentiation block diagram, current value of position is subtracted by previous one and the result is multiplied by $1/h$. The output of multiplier represents velocity which is numerical differentiation of position.

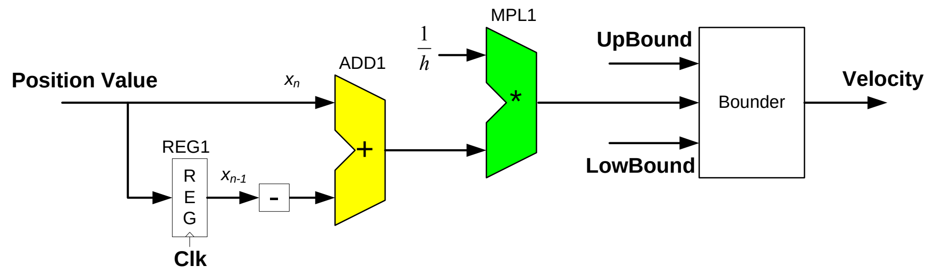


Figure 39. Hardware block diagram for implementing numerical differentiation.

4.6.1.3 VHDL module for LQR controller

A control input to the plant in a LQR controller is given below.

$$u = -Kx + \bar{N}r \tag{12}$$

, where u represents a control signal for the plant and r is a reference signal to the plant. $\bar{N}$ is introduced to compensate the problem of non-zero steady-state error with an LQR state feedback controller.

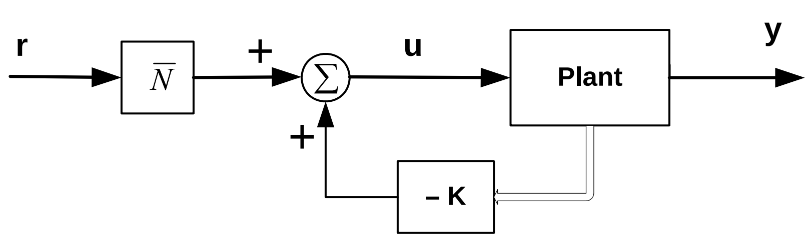


Figure 40. LQR state feedback controller with a compensation block.

Figure 41 represents hardware block diagram for implementing LQR state feedback controller and the output u of this module denotes a control signal for the plant.

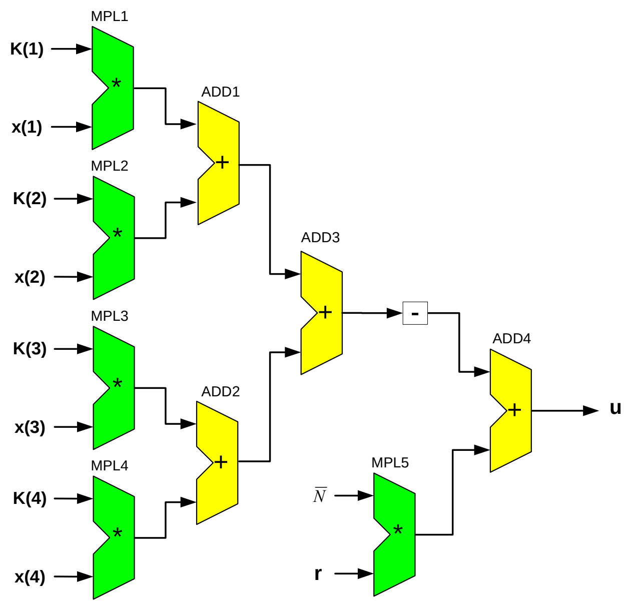


Figure 41. Hardware block diagram for implementing LQR controller.

4.6.2 Dynamic reconfiguration

The following example is to change a plant model during run-time dynamic reconfiguration. While a Supervisor module will constantly check the performance of the robot, it can generate the command of selecting a different plant model and feed input to a configuration manager module. The configuration manager will try to load a new plant model into a memory. When finished the calculation of a new plant model, there is a command to switch a plant model and the system will have the new plant model which might provide better performance.

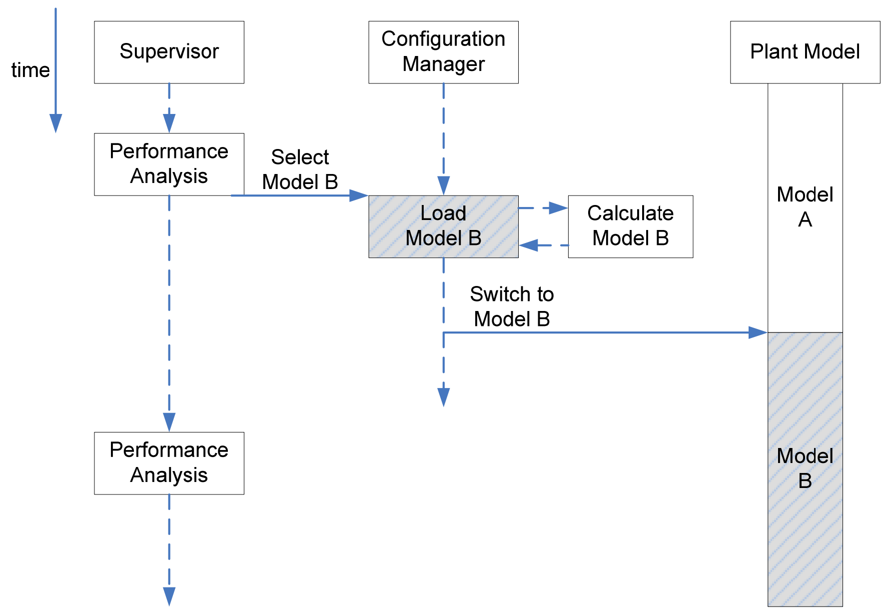


Figure 42. Run-time dynamic reconfiguration of plant model.

Reconfiguration will be achieved by the following three strategies.

1. Change LQR control parameters:

While running a LQR controller, if there is any need to change gain matrices by measuring the performance, gain matrix K will be updated to meet the design specification.

2. Change the order of a model:

If the system is still performing poorly though the gain matrix was changed, reducing the order of a plant model can be attempted.

3. Change a controller:

When there is any resource problem (such as FPGA logic area, power usage, or speedup of logic), a controller can be changed. For example, parallel PID hardware algorithm can be exchanged with serial PID implementation in order to secure more logic space and reduce power consumption [40].

4.6.3 Plant model reduction

In this section plant model reduction which is one way of dynamic reconfiguration to achieve better performance is explained. A state space equation for an open-loop system was shown in equation () which is fourth-order system:

$$
\begin{aligned}
&A=\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -0.5549 & 5.5612 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -5.0441 & 139.6469 & 0 \end{bmatrix} \quad B=\begin{bmatrix} 0 \\ 3.3291 \\ 0 \\ 30.2648 \end{bmatrix} \\
&C=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad D=\begin{bmatrix} 0 \\ 0 \end{bmatrix}
\end{aligned} \tag{13}
$$

A new plant model with the reduced order of three is derived using square-root balanced truncation method [68] and specific steps of calculation to build the following third-order system are shown in Appendix J.

$$
\begin{aligned}
&A_{M1}=\begin{bmatrix} -0.3541 & 0 & 0 \\ 0 & 11.6539 & 0.6924 \\ 0 & 1.1155 & 0.0663 \end{bmatrix} \quad B_{M1}=\begin{bmatrix} 2.4494 \\ 2.3268 \\ 2.2326 \end{bmatrix} \\
&C_{M1}=\begin{bmatrix} -2.4494 & -0.2662 & 2.9864 \\ 0.0002 & 0.5053 & 0.0300 \end{bmatrix} \quad D_{M1}=\begin{bmatrix} 0 \\ 0 \end{bmatrix} \\
&totbnd=0.1044
\end{aligned} \tag{14}
$$

,where *totbnd* indicates a bound for the infinity norm of an error between original and reduced model.

Parameters for the LQR controller can be obtained as shown below.

$$Q_{M1} = 2000 \times C_{M1}{}^{T} C_{M1} = \begin{bmatrix} 11999 & 1304 & -14630 \\ 1304 & 652 & -1560 \\ -14630 & -1560 & 17839 \end{bmatrix} \quad R_{M1} = R = 1$$
$$K_{M1} = \begin{bmatrix} 93.7535 & 58.9185 & -130.7624 \end{bmatrix} \tag{15}$$

Another plant model with second-order system can be also obtained as follows:

$$A_{M2} = \begin{bmatrix} -0.3541 & 0 \\ 0 & 1.7953\text{e-}15 \end{bmatrix} \quad B_{M2} = \begin{bmatrix} 2.4494 \\ -2.4495 \end{bmatrix}$$
$$C_{M2} = \begin{bmatrix} -2.4494 & -2.4495 \\ 2.4468\text{e-}004 & -3.0184\text{e-}016 \end{bmatrix} \quad D_{M2} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{16}$$
$$totbnd = 0.3269$$

Parameters for the LQR controller for the second order system can be obtained as shown below.

$$Q_{M2} = 2000 \times C_{M2}{}^{T} C_{M2} = \begin{bmatrix} 11999 & 12000 \\ 12000 & 12000 \end{bmatrix} \quad R_{M2} = R = 1$$
$$K_{M2} = \begin{bmatrix} -104.0638 & -109.5445 \end{bmatrix} \tag{17}$$

Bode plot of original open-loop system and reduced-order systems are displayed in Figure 43 and Figure 44. Regarding the first output of bode plot, which is the position of the robot, both of the reduced-order systems show little difference from the original system in magnitude response. But for the second output, which is the tilt angle of the chassis, the third-order system shows about maximum of 20 dB deviations from the original system but the second-order system indicates about maximum of 60 dB differences in magnitude.

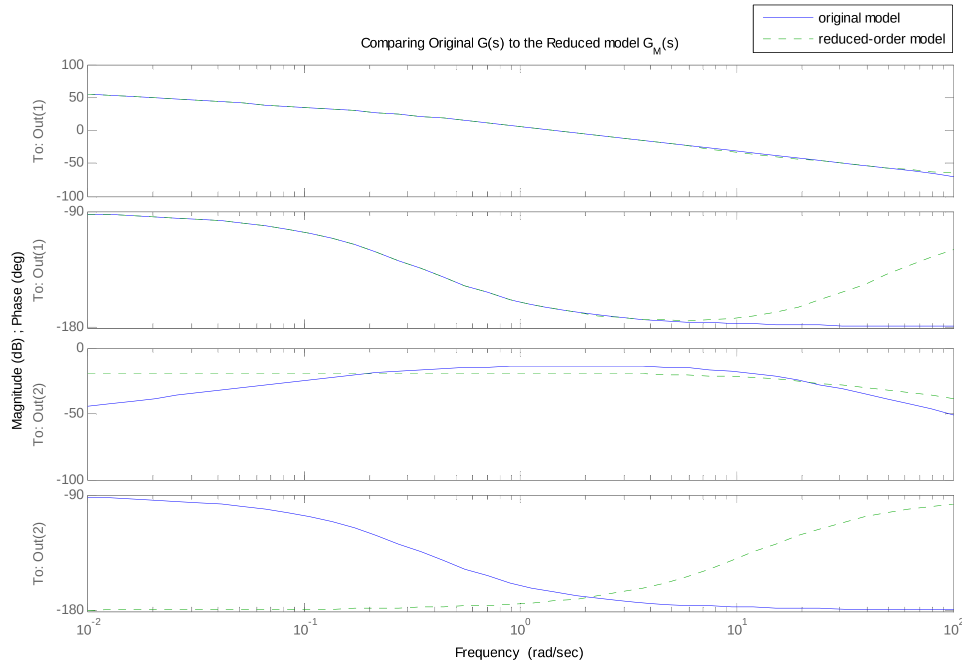


Figure 43. Bode plot of the original fourth-order model and the reduced third-order model.

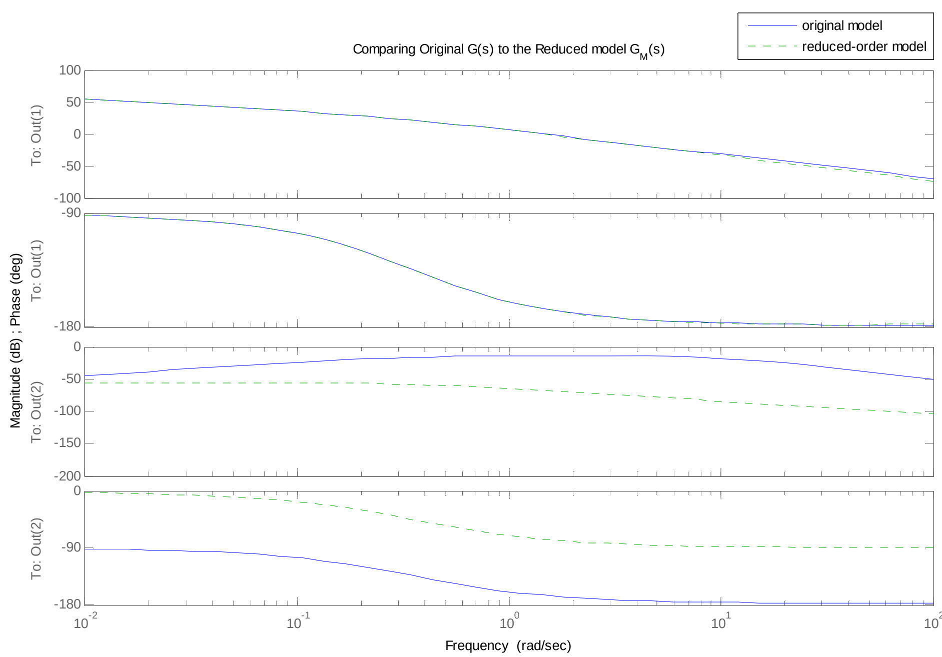


Figure 44. Bode plot of the original fourth-order model and the reduced second-order model.

Simulation results of step responses for the reduced-order model are investigated using MATLAB. The following figures show the step response of the original fourth-order model and reduced-order models with LQR control, where Matlab is used in this simulation. These responses indicate that the initial backward motion of the robot goes smaller and the swing width of the tilt angle lessens as the order of plant model reduces.

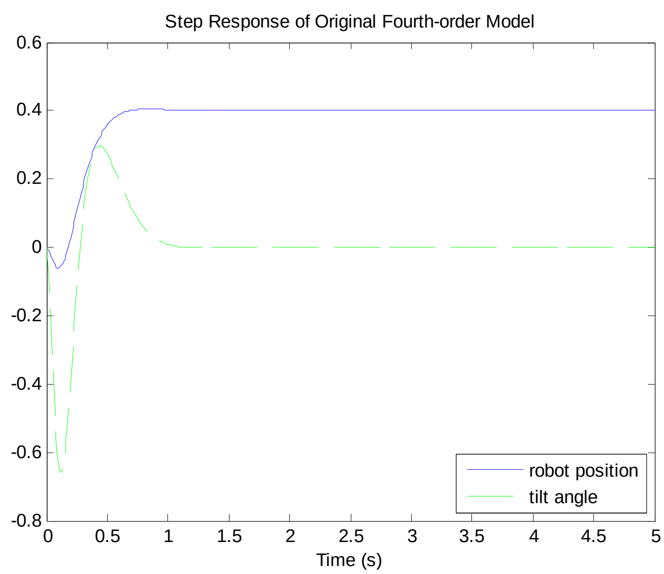


Figure 45. Step response of original fourth-order model (rise time: 0.2960, settling time: 0.6147).

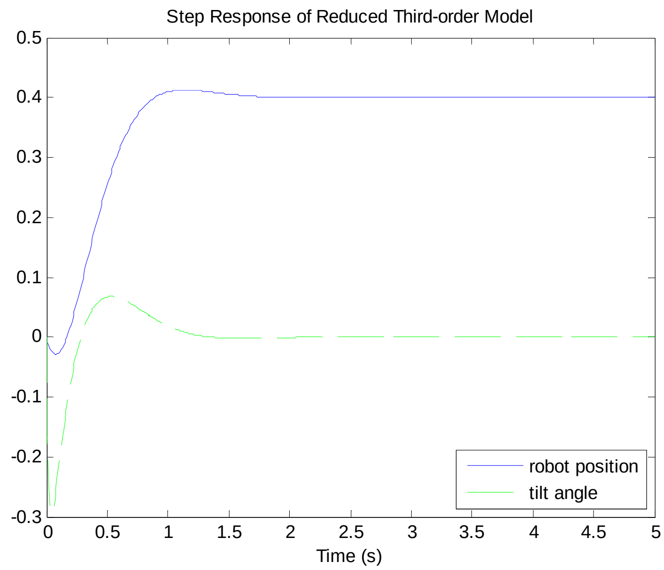


Figure 46. Step response of reduced third-order model (rise time: 0.4844, settling time: 1.4048).

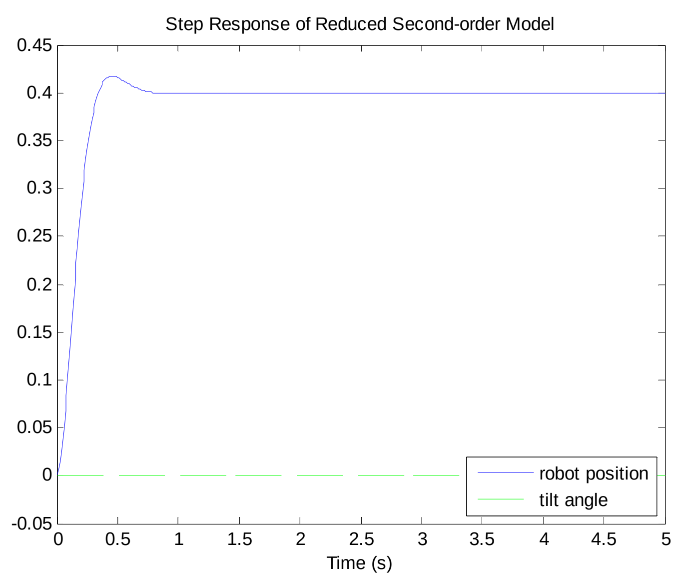


Figure 47. Step response of reduced second-order model (rise time: 0.2206, settling time: 0.6114).

4.7 Experiments and observations

4.7.1 Experimental platform

In order to verify the functionality of resource-adaptive control, a balancing robot has been built using FPGA board as shown in Figure 48. Digilent FX12 FPGA board with Xilinx Virtex-4 FX12 FPGA served as the main platform and processing unit. Two sensors are used in this robot. One is gyroscope sensor ADXRS401 from Analog Devices and it can measure angular rate with a range of ±75 °/s. The other one is accelerometer sensor ADXL203 from Analog Devices and it can measure acceleration with a full-scale range of ±1.7 *g*. A robotic frame was selected from Digilent robotic kit which includes rugged metal frame with holes and two 1/19 ratio motor/gearbox with ABS plastic wheels. For motor amplifiers Digilent PMOD HB5, H-bridge motor amplifiers were used which also includes encoder circuitry. For Analog-to-digital (A/D) converter, Digilent PMODAD1 has been chosen. It has two 12-bit A/D converters with a maximum sampling rate of one million samples per second which is fast enough for the most robotic applications.

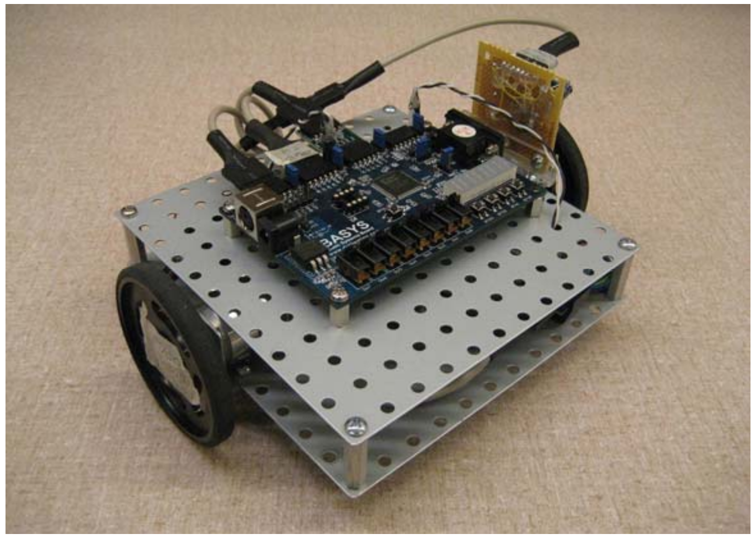


Figure 48. Two-wheeled balancing robot using FPGA board.

Figure 49 shows an example image to gather sensor information. The screen capture of Xilinx Chip Scope Pro software displays how to obtain the data of gyroscope sensor and the sensor data are intrinsically noisy and typically drifts over time.

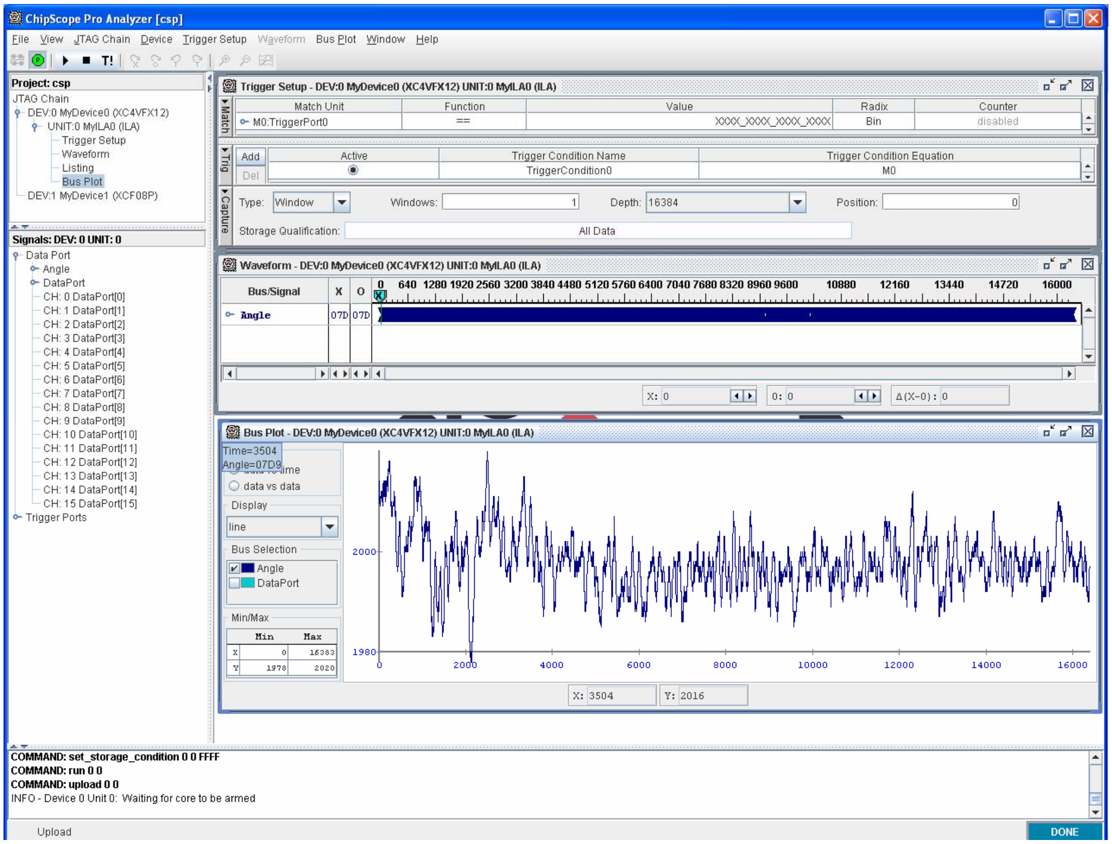


Figure 49. Data capturing of gyroscope sensor for 6.72 seconds when the robot is stationary upright position.

4.7.2 Experimental results of reduced-order models

4.7.2.1 Step responses without any load

Experimental results of step responses for different plant models are investigated. In this experiment there is no load to the motors, which means that a robot doesn’t have any contact with ground and it’s in the air. The following figures show the step response of the original fourth-order model and reduced-order models with LQR control, which is conducted in Virtex-4 FPGA board. LQR controller was implemented in FPGA board written in VDHL and encoder data are captured using Xilinx Chip Scope Pro software. Experimental results show that both the rise time and settling time decrease with reduced-order model.

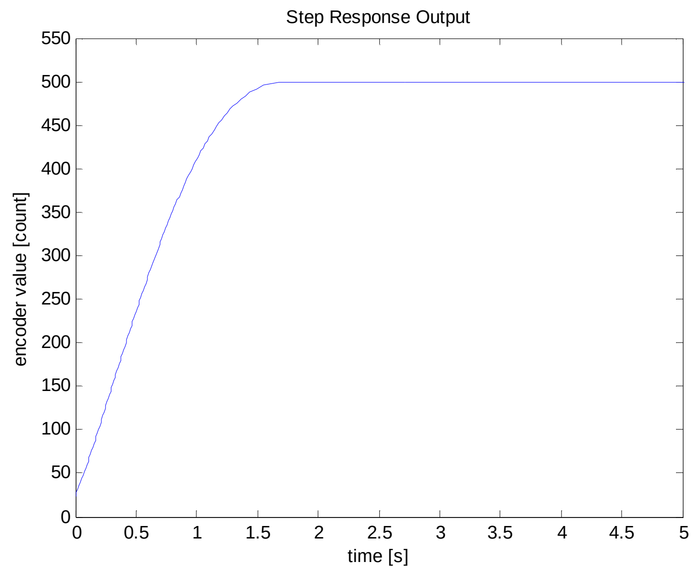


Figure 50. Step response of original fourth-order model (rise time: 1.0592, settling time: 1.4674).

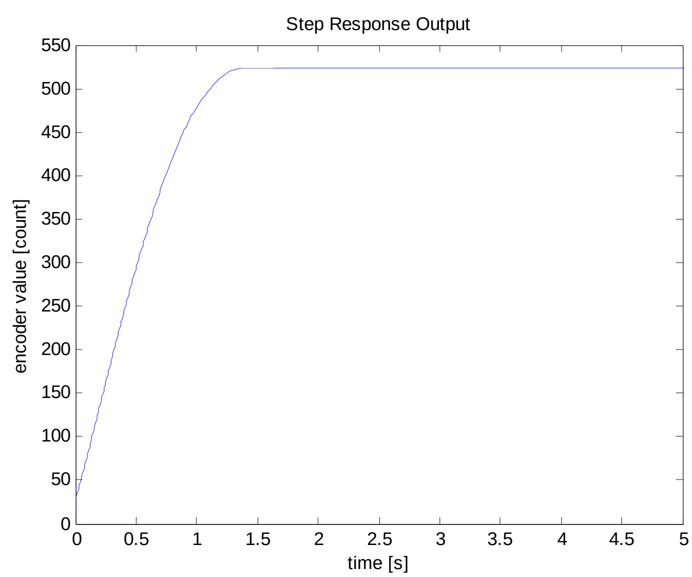


Figure 51. Step response of reduced third-order model (rise time: 0.8851, settling time: 1.2130).

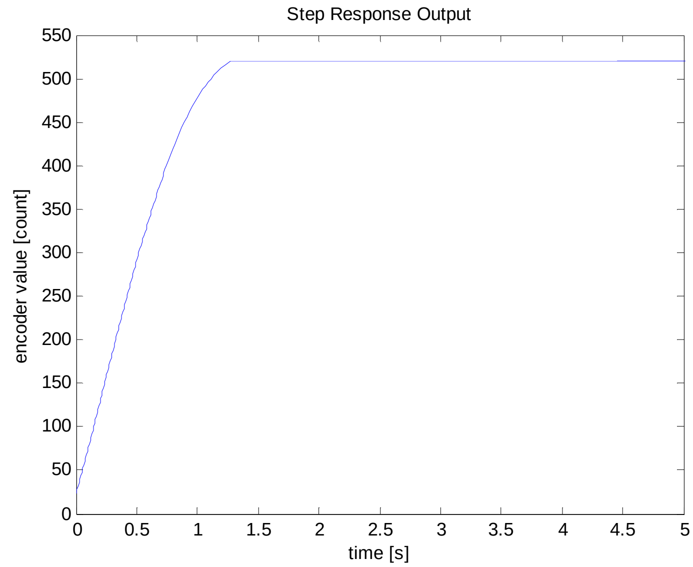


Figure 52. Step response of reduced second-order model (rise time: 0.8685, settling time: 1.1750).

4.7.2.2 Step responses with load

The following figures show an example of step responses of the robot with load. At this time a robot is on the ground and moving around. Thus output signals contain high frequency noise and spikes. Step input was employed at time 3 seconds and zero input was again applied at time 11 seconds. All those output figures show noisy data due to the intrinsic nature of balancing robot. As the order of robot model is reduced, the output indicates a little bit higher magnitude of step response and more perturbation or sway.

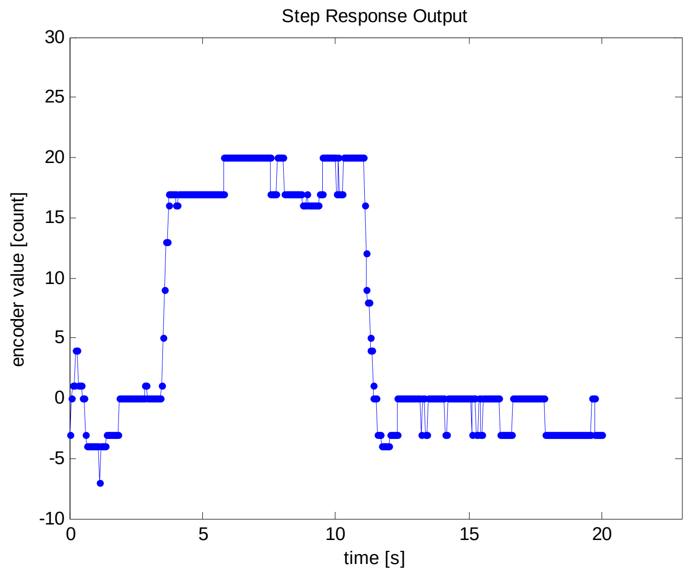


Figure 53. Step response of the robot with load (fourth-order plant model).

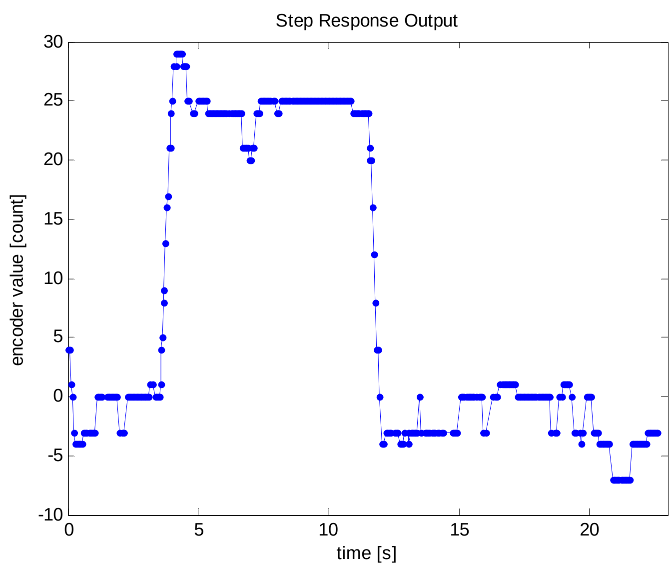


Figure 54. Step response of the robot with load (third-order plant model).

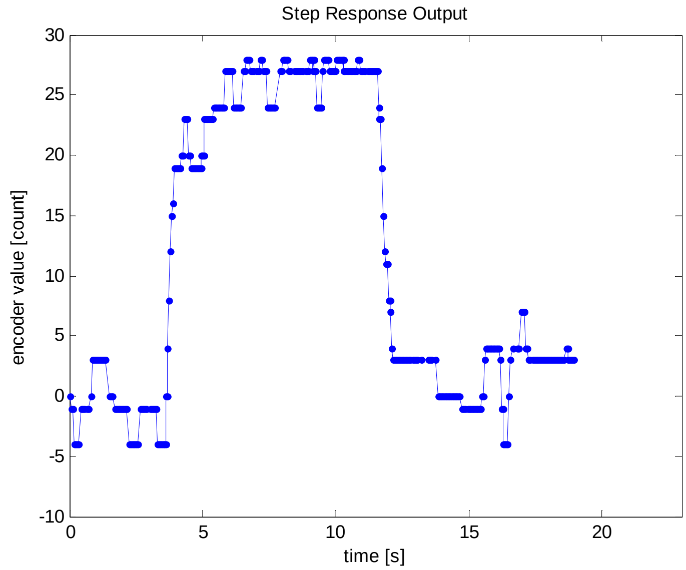


Figure 55. Step response of the robot with load (second-order plant model).

The following figure represents tilt angle of the robot while conducting step response experiment. Since the robot is continuously balancing itself, the tilt angle data are pretty noisy and centered at zero angle.

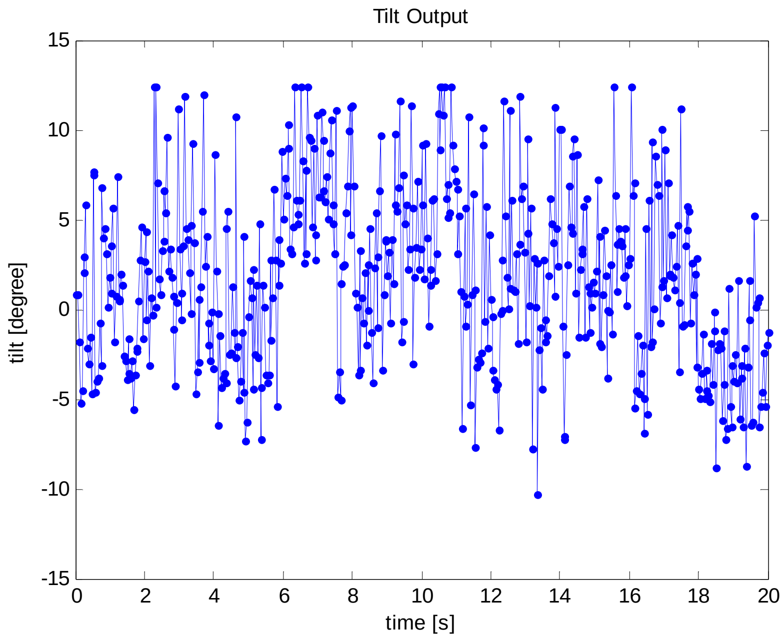


Figure 56. Capturing of tilt information of the robot during step response experiment.

4.7.3 Experimental results of plant change

In this experiment the mass of plant has been changed. More mass of 200g was added to the robot in order to examine any change in system performance with plant change. From the result of simulation, slower rise time and slower settling time can be expected, but it's not easy to identify those properties from the experimental results. The following figure shows a step response of the robot with same step input profile as previous experiment. It shows even less noisy position values at steady state.

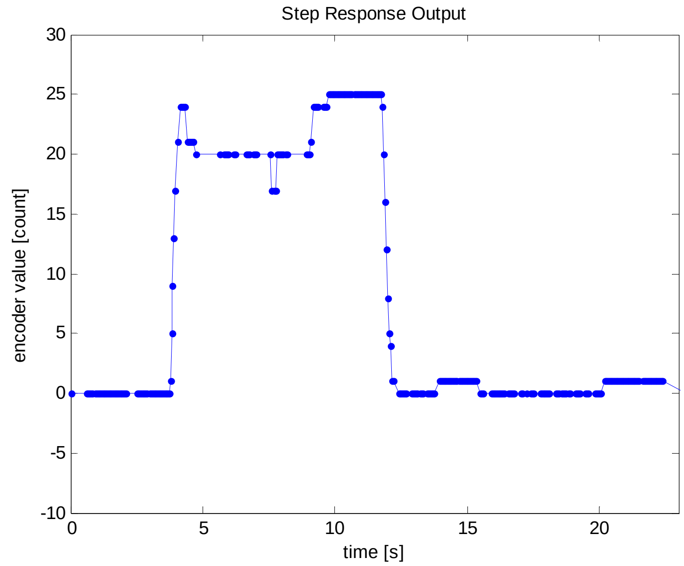


Figure 57. Step response of the robot with added mass (fourth-order plant model).

4.7.4 Experiment with LEGO balancing robot

Experimental results with LEGO balancing robot are shown in this section in order to compare the result with FPGA board.

First experiment is to control the self–balancing robot while maintaining its position. The following measurements are used to control the robot; angular velocity, angle, position, and velocity of the robot. Angular velocity is obtained from gyroscopic sensor and the tilt can be calculated by numerical integration. Position of the robot is measured by encoder of the motor and velocity of robot can be computed by numerical differentiation. All of these four measurements are being used to determine driving power for the two motors.

Source code for stationary self-balancing robot is shown in Appendix K and detailed specification of LEGO Mindstorms NXT is shown in Appendix L.

The following figure shows a captured tilt angle of LEGO robot while maintaining its position.

The data are captured for 6 seconds and 3000 points are captured. The robot has tilt range of -14º and 8 º.

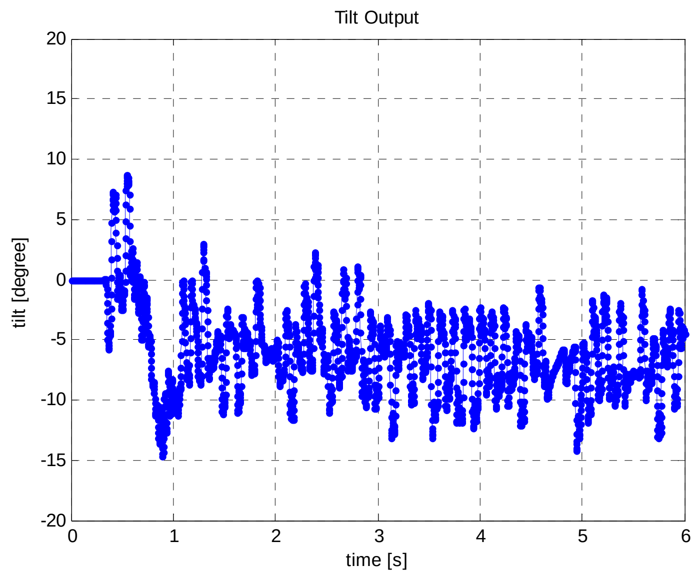


Figure 58. Captured tilt angle of LEGO robot while maintaining its position.

Then additional mass of 650g is appended to the LEGO robot in order to examine the robustness of the robot to a plant change. There is slight change when there is additional mass to the robot. The magnitude of oscillation is decreased a little and the frequency of oscillation has been increased, which can be expected when there is an increase in mass of plant.

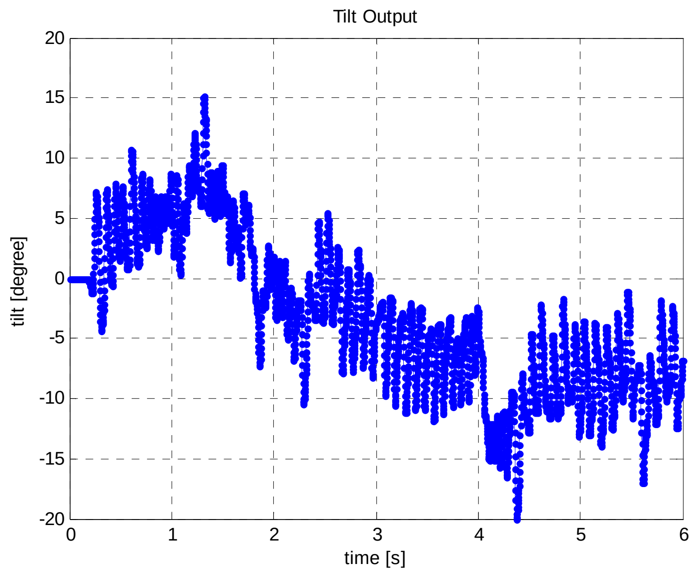


Figure 59. Captured tilt angle of LEGO robot with added mass while maintaining its position.

The following figures show step responses of the LEGO robot. A step input was employed at time 1 second and zero input was applied at time 3. The first figure was generated when there is no extra mass and the next figure was made when there is additional mass of 650g. As you can see from the step response result, there is obviously decreased rise time with appended mass.

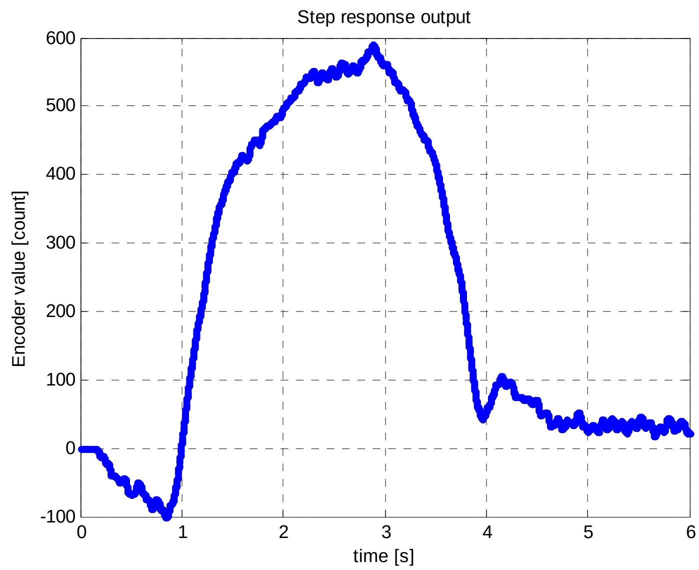


Figure 60. Step response plot without extra mass.

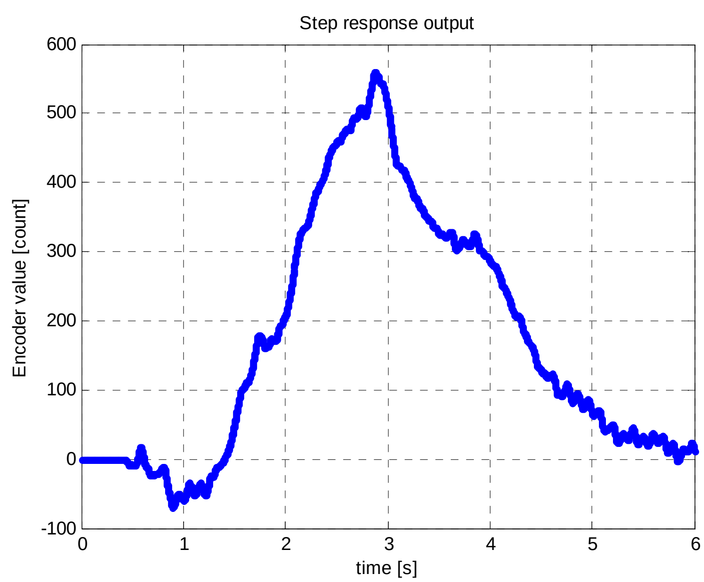


Figure 61. Step response plot with additional mass.

4.7.5 Partial reconfiguration flow

Partial reconfiguration design flow is shown here using Xilinx ISE 9.2 with Xilinx PlanAhead 10.1.

PlanAhead software is used to floorplan a design, having one partial reconfiguration region with two reconfigurable modules, and then partial reconfiguration implementation flow is chosen by using ExploreAhead feature of PlanAhead. In order to verify the design in hardware, Digilent FX12 board is utilized, which has Xilinx Virtex-4 FX12 FPGA. iMPACT software is used to download the full and partial bitstreams.

Here are some basic terminologies used in partial reconfiguration.

Reconfigurable Region (RR) is the physical area on the FPGA device which will be reconfigured by different reconfigurable modules. The specified region will be maintained and single top level netlist is assigned.

Reconfigurable Module (RM) is the logic which will occupy the reconfigurable region. Usually more than one reconfigurable module is defined for each reconfigurable module.

Bus Macro is the logic macro which is interfacing between static logic and reconfigurable region. In PlanAhead they must be manually placed to decide the interconnection points.

Partial Bitstream is the bitstream file which is generated from each reconfigurable module. The number of partial bitstream will correspond to the number of reconfigurable modules.

Merged Full Bitstream is the file which contains one configuration combining static logics with reconfigurable modules for each reconfigurable region.

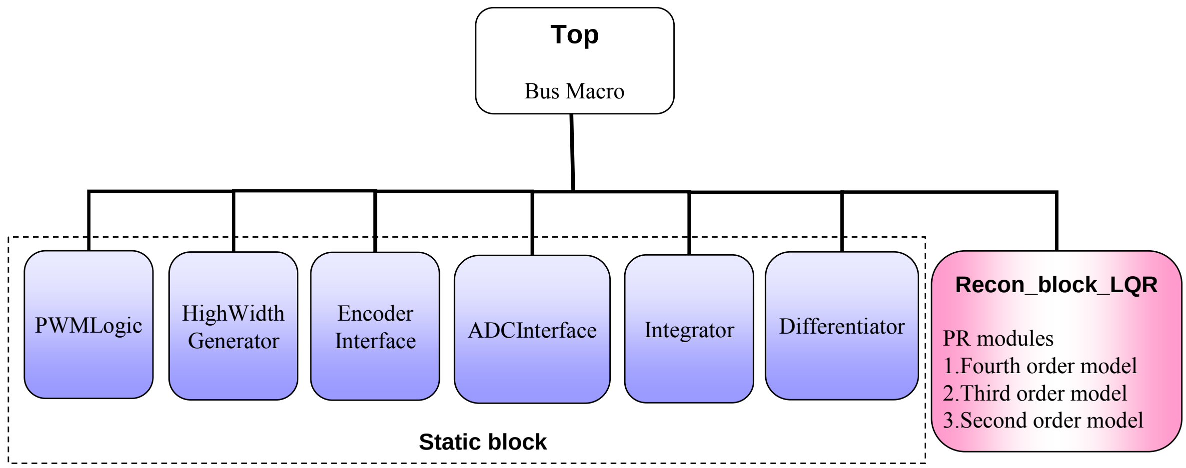


Figure 62. Complete system block diagram.

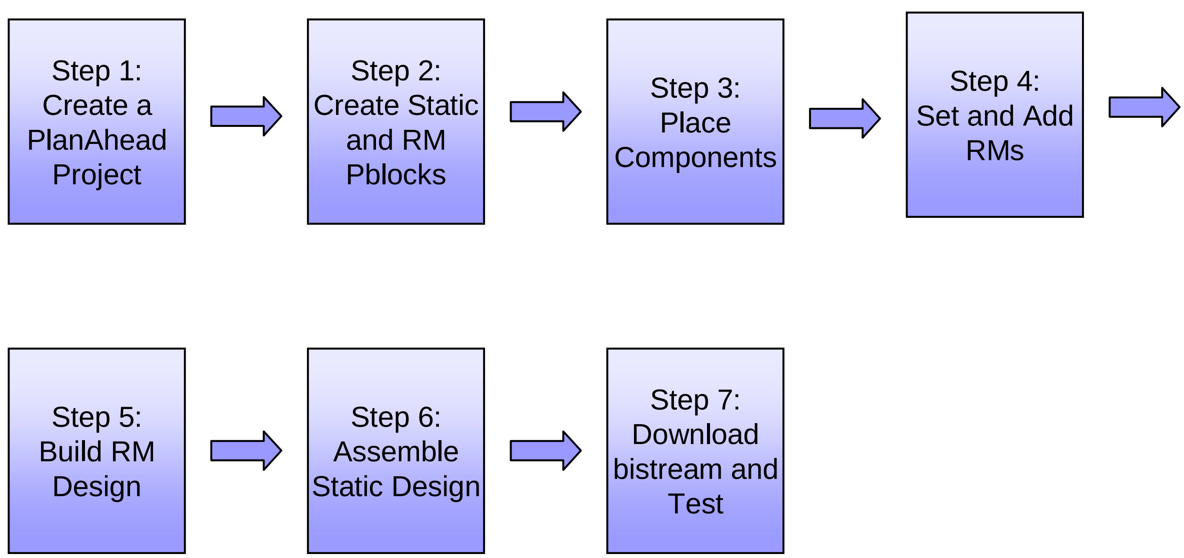


Figure 63. Project Flow using PlanAhead.

Specific steps for partial reconfiguration are explained in Appendix M in detail.

Here are the results of implementation.

In Table 3 DSP48 resource is required in reconfigurable module and thus reconfigurable module needs to be deliberately placed where DSP48 is located. From Table 4 the size of resultant bitstream is shown and it can be beneficial to figure out the size of file while downloading. Boundary crossing nets in Table 5 represent the totally utilized wires used to interconnect between static and reconfigurable module and bus macros are sitting between two modules.

Table 3. Physical resource utilization for partially reconfigurable module.

Type of Site	Available	Required	% Utilization
LUT	864	229	26.50
FF	864	0	0.00
SLICEL	216	70	32.41
SLICEM	216	70	32.41
DSP48	8	5	62.50

Table 4. Statistics for partially reconfigurable module.

Type	Value
Bitstream size	43783 Bytes
No of Frames	14
Number of Frame Region	2

Table 5. Boundary crossing nets statistics.

Boundary Crossing Nets
128

Table 6. Physical resource counts for partially reconfigurable module.

Cellview Name	Count
DSP48	5
GND	11
INV	31
LUT1	1
LUT2	192
LUT3	4
LUT4	32
MUXCY	219
MUXF5	16
VCC	11
XORCY	155

The following Figure 64 shows floor plan of the FPGA chip (Xilinx Virtex-4 FX12) in PlanAhead and the rectangular box at upper right side indicates the reconfigurable region. Bus macros are placed on the right edge of the reconfigurable region. LQR controller is implemented in this reconfigurable region. For example, with initial static configuration, the reconfigurable region could be configured to have LQR controller with fourth-order plant model. During dynamic reconfiguration, this region could be reconfigured to contain LQR controller with third-order plant model. Refer to Figure 36 for detailed hardware block diagram of LQR controller.

In Figure 65 connected nets are displayed and all the modules are placed. As it can be easily identified, the red rectangular box at upper right side indicates a reconfigurable region and all the bus macros are located the right side of the red box with gray color.

Floor planning view for each static or reconfigurable module is also attached in Figure 66 and Figure 67.

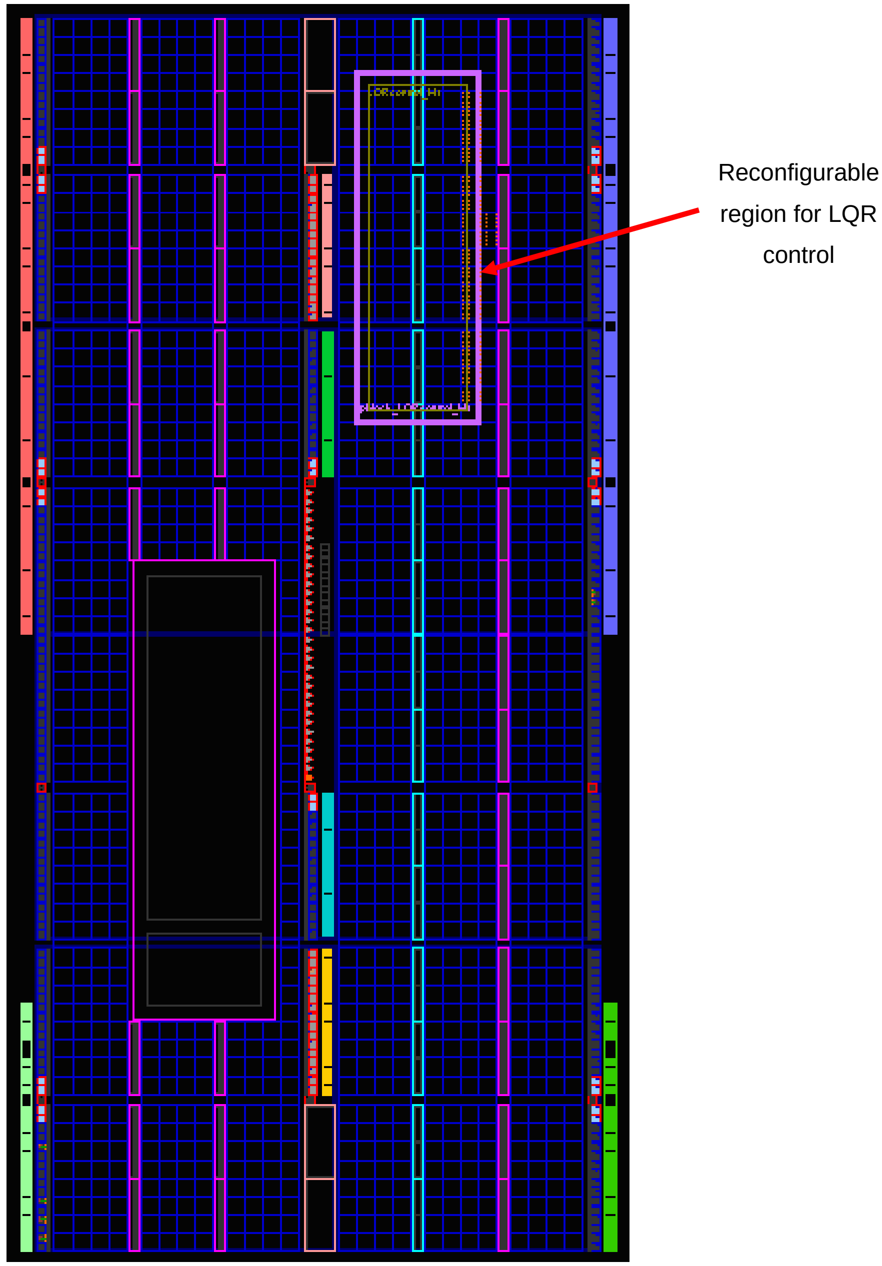


Figure 64. Floor planning in PlanAhead software.

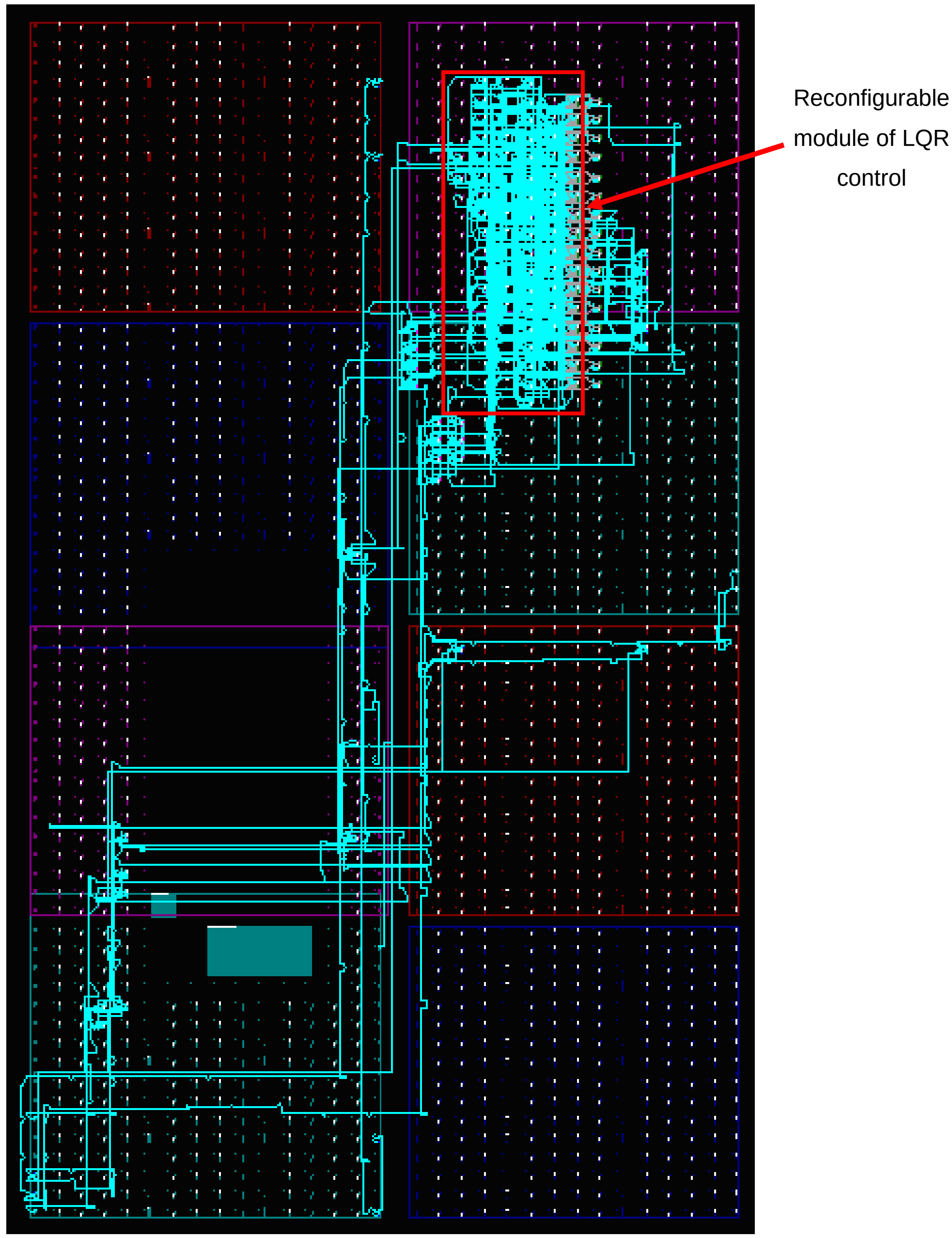


Figure 65. floor planning view of both static and reconfigurable module using FPGA editor.

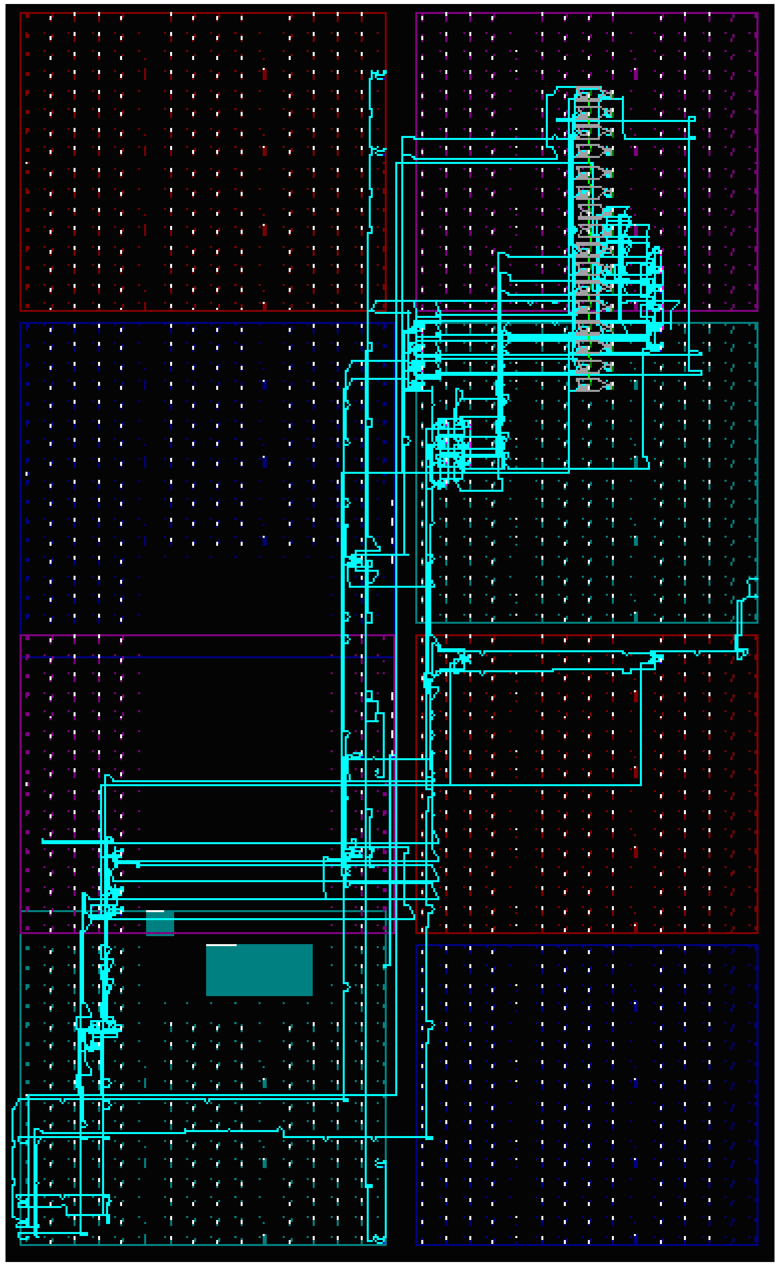


Figure 66. floor planning view of only static module using FPGA editor.

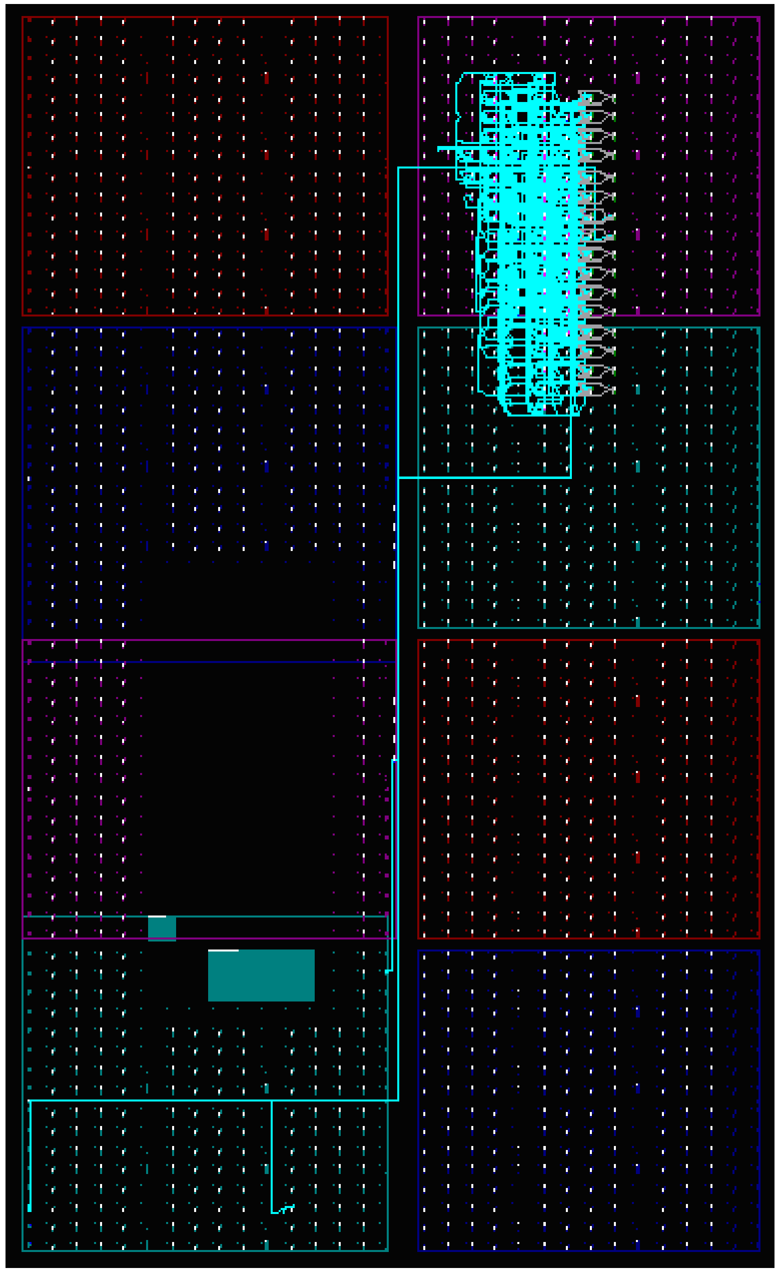


Figure 67. floor planning view of only reconfigurable module (lqr_fourth) using FPGA editor.

V. Conclusion

This dissertation has developed hardware and software architectures for a reconfigurable computing platform targeting a small-scale resource-constrained robot.

In this thesis, a static reconfiguration architecture called Morphing Bus was presented, which allows flexible and rapid assembly of sensors and actuators. This is a novel architecture to interface multiple sensor and actuator systems without a bus interface logic. With the morphing bus, a quick and easy method to deploy systems can be achieved. It has the flexibility of swapping and adding or removing devices prior to deployment along with dedicated connections to the computing platform without the use of a bus interface logic or an arbitration logic. A tool that configures the bus was developed and rigorously tested to show the validity of the morphing bus.

A software architecture for hardware/software dynamic reconfigurability was also proposed, which provides the reallocation of hardware and software resources at run time. A new methodology to choose an optimal configuration using a configuration tree is presented and metrics for cost functions in the configuration tree are introduced. The cost function not only includes resource utilization, but also involves functionality. Preliminary testing of the system was done on PID control architectures, which were experimented with serial and parallel structures of the PID control. The reconfigurable hardware gives us the ability to save resources as well as accommodate new types of sensors.

This reconfigurable platform constituted the backbone for the TerminatorBot and such platform would be applied to other resource-constrained systems. For example, in heterogeneous wireless sensor networks, we could construct two robots with the different set of sensors and make them cooperate to do one task.

This reconfigurable platform was applied to an adaptive control scheme where control parameters can be modified to adapt a changing environment. Resource-adaptive control, which is a new type of adaptive control, utilized LQR control to decide feedback gains for a FPGA-based balancing robot. The dynamic reconfigurable platform can chose different gains for LQR controller or change the order of the plant model. The reconfigurable platform was evaluated with a change in the mass of a plant. Lastly, the performance of the FPGA-based balancing robot was compared with a LEGO balancing robot which was developed in our laboratory. Similar to the FPGA-based robot, the LEGO robot had a gyroscopic sensor to obtain a tilt angle of the robot body. Rise time and settling time were measured with changing plant parameters for both of the robots.

Future work would include developing task aware reconfiguration of FPGAs where a processor can choose suitable reconfigurable modules depending upon given tasks.

The morphing bus concepts can also be extended to an internal morphing bus where a symmetric morphing bus structure can be extended internally into the FPGA to facilitate dynamic system reconfiguration. Then, it can also be applied to develop operating systems for FPGAs. Morphing bus structure can provide relocatable system modules. Some modules can be used to run control algorithms, while others can be used to interface devices on the bus and these modules will eventually constitute operating systems for FPGAs.

This hardware reconfigurable platform can be combined with a software adaptation technique. For example, when the platform is applied to Larson [39]'s gait adaptation, a robot can change its hardware structure by way of reconfiguration in order to support gait adaptation.

References

[1] H. Jones and M. Snyder, "Supervisory control of multiple robots based on a real-time strategy game interaction paradigm," in *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, 2001, pp. 383-388 vol.1.

[2] B. H. Kim, C. D'Souza, R. M. Voyles, J. Hesch, and S. Roumeliotis, "A Reconfigurable Computing Platform for Plume Tracking with Mobile Sensor Networks," in *Proceedings of the 2006 SPIE Defense and Security Symposium* Orlando, FL, April, 2006.

[3] R. M. Voyles, "TerminatorBot: A Robot with Dual-Use Arms for Manipulation and Locomotion," in *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 2000, pp. 61-66.

[4] "http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf."

[5] W. F. Arnold and A. J. Laub, "Generalized eigenproblem algorithms and software for algebraic Riccati equations," in *Proceedings of the IEEE*, 1984, pp. 1746-1754.

[6] "http://web.mac.com/ryo_watanabe/iWeb/Ryo's%20Holiday/NXTway-G.html."

[7] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel Processing in a Restructurable Computer System," *Electronic Computers, IEEE Transactions on,* vol. EC-12, pp. 747-755, 1963.

[8] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (CSUR) archive,* vol. 34 pp. 171 - 210, June 2002.

[9] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the conference on Design, automation and test in Europe*, 2001.

[10] A. DeHon and J. Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," in *Proceedings of the 36th ACM/IEEE conference on Design automation*, 1999, pp. 610 - 615.

[11] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *Computers and Digital Techniques, IEE Proceedings,* vol. 152, pp. 193-207, 2005.

[12] F. J. Kurdahi, "Reconfigurable computing: is it ready for industry," in *Pervasive*

Services, 2005. ICPS '05. Proceedings. International Conference on, 2005, pp. 337-343.
[13] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An Overview of Reconfigurable Hardware in Embedded Systems," in *EURASIP Journal on Embedded Systems*, 2006, pp. 1-19.
[14] "http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html."
[15] C. S. Draper and Y. T. Li, "Principles of optimalizing control systems and an application to the internal combustion engine," *ASME publication,* 1951.
[16] H. P. Whitaker, J. Yamron, and A. Kezer, "Design of model reference adaptive control systems for aircraft," Instrumentation Lab., MIT Report No. R-164, 1958.
[17] Y. T. Li and W. E. van der Velde, "The philosophy of nonlinear adaptive control," in *Proc. IFAC Congress*, Moscow, 1960.
[18] K. J. Astrom and B. Wittenmark, "On self-tuning regulators," in *Automatica*, 1973, pp. 185-199.
[19] Y. Zhang and J. Jiang, "Bibliographical review on reconfigurable fault-tolerant control systems," in *Proceeding of the SAFEPROCESS 2003: 5th Symposium on Detection and Safety for Technical Processes, IFAC* Washington D.C., USA, 2003, pp. 265-276.
[20] "http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores."
[21] C. D'Souza, B. H. Kim, and R. Voyles, "Morphing Bus: A rapid deployment computing architecture for high performance, resource-constrained robots," in *2007 IEEE International Conference on Robotics and Automation (ICRA 2007)* Roma, Italy, April 2007, pp. 311-316.
[22] M. Borgatti, F. Lertora, B. Foret, and L. Cali, "A Reconfigurable System Featuring Dynamically Extensible Embedded Microprocessor, FPGA, and Customizable I/O," in *IEEE Journal of Solid-State Circuits*, March 2003, pp. 521 – 529.
[23] K. Rauma, O. Laakkonen, T. Harkonen, O. Pyrhonen, and J. Luukko, "New Bus Structure for Programmable Logic Devices Controlling Power Electronics," in *2005 IEEE 36th Conference on Power Electronics Specialists*, June 12, 2005, pp. 2705 – 2708.
[24] C. Guéganno and D. Duhaut, "A hardware/software architecture for the control of self reconfigurable robots," in *DARS-2004* France, June 2004.

[25] R. A. Gonçalves, P. A. Moraes, J. M. P. Cardoso, D. F. Wolf, M. M. Fernandes, R. A. F. Romero, and E. Marques, "ARCHITECT-R: A System for Reconfigurable Robots Design," in *Symposium on Applied Computing (SAC 2003)*, Melbourne, Florida, March 9-12, pp. 679-683.

[26] A. Upegui, R. Moeckel, E. Dittrich, A. Ijspeert, and E. Sanchez, "An FPGA Dynamically Reconfigurable Framework for Modular Robotics," in *Workshop Proceedings of the 18th International Conference on Architecture of Computing Systems 2005 (ARCS 2005)*, Berlin, Germany, pp. 83-89.

[27] C. Paiz, T. Chinapirom, U. Witkowski, and M. Porrmann, "Dynamically Reconfigurable Hardware for Autonomous Mini-Robots," in *IEEE Industrial Electronics, IECON 2006 - 32nd Annual Conference on*, 2006, pp. 3981-3986.

[28] N. Chakravarthy and X. Jizhong, "FPGA-based Control System for Miniature Robots," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, 2006, pp. 3399-3404.

[29] K. Weiss, R. Kistner, A. Kunzmann, and W. Rosenstiel, "Analysis of the XC6000 Architecture for Embedded System Design," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, April 15-17, 1998, pp. 245-252.

[30] J. Noguera and R. M. Badia, "HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures," in *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 10, August 2002, pp. 399-415.

[31] R. Maestre, F. J. Kurdahi, M. Fernandez, and R. Hermida, "A Framework for Scheduling and Context Allocation in Reconfigurable Computing," in *Proc. of the International Symposium on System Synthesis*, 1999, pp. 134-140.

[32] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," in *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 9, August 2001, pp. 545-557.

[33] N. Shirazi, W. Luk, and P. Y. K. Cheung, "Framework and tools for run-time reconfigurable designs," in *IEE Proceedings of Computers and Digital Techniques*, Vol. 147, May 2000, pp. 147-151.

[34] S. Choi, R. Scrofano, V. K. Prasanna, and J.-W. Jang, "Energy-Efficient Signal Processing Using FPGAs," in *Proceedings of ACM/SIGDA 11th ACM International Symposium on FPGA*, Monterey CA, Feb. 23-25, 2003, pp. 225-233.

[35] S. Gupta and F. N. Najm, "Power Modeling for High-Level Power Estimation,"

in *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 8, Feb. 2000, pp. 18-29.

[36] L. Shang and N. K. Jha, "Hardware-Software Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs," in *Proceedings of the 15th IEEE International Conference on VLSI Design (VLSID'02)*, 2002.

[37] L. Shang and N. K. Jha, "High-level Power Modeling of CPLDs and FPGAs," in *Proceedings of 2001 IEEE International Conference on Computer Design*, Sep. 23-26, 2001, pp. 46-51.

[38] A. Garcia, W. Burleson, and J. L. Danger, "Low Power Digital Design in FPGAs: A Study of Pipeline Architectures implemented in a FPGA using a Low Supply Voltage to Reduce Power Consumption," in *Proceedings of 2000 IEEE International Symposium on Circuits and Systems*, Geneva, May 28-31, 2000, pp. 561-564.

[39] A. C. Larson, G. K. Demir, and R. M. Voyles, "Terrain Classification Using Weakly-Structured Vehicle/Terrain Interaction," in *Autonomous Robots*, v. 19, 2005, pp. 41-52.

[40] W. Zhao, B. H. Kim, A. C. Larson, and R. M. Voyles, "FPGA Implementation of Closed-Loop Control System for Small-Scale Robot," in *Proceedings of the 2005 International Conference on Advanced Robotics*, Seattle, WA, July 2005, pp. 70-77.

[41] Y. Meng, "An agent-based reconfigurable system-on-chip architecture for real-time systems," in *2nd International Conference on Embedded Software and Systems 2005*, Dec. 16-18, 2005.

[42] T. Ramacciotti, L. Serafini, L. Fanucci, and S. Baldacci, "A novel approach based on dynamic reconfiguration for process controls with FPGA," *Research in Microelectronics and Electronics, 2005 PhD,* vol. 1, pp. 141- 144, 2005.

[43] K. Danne and C. Bobda, "Dynamic reconfiguration of distributed arithmetic controllers: design space exploration and trade-off analysis," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 26-30 April 2004, p. 140.

[44] C. Paiz, B. Kettelhoit, and M. Porrmann, "A design framework for FPGA-based dynamically reconfigurable digital controllers," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, 27-30 May 2007, pp. 3708 - 3711.

[45] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A self-reconfiguring platform," in *Field-Programmable Logic and Applications*, September 2003, (Springer-Verlag), pp. 565–574.

[46] D. Stewart, R. Volpe, and P. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," in *IEEE Trans. On Software Engineering*, vol. 23, December 1997.

[47] P. Christian, E. Rolf, W. Herbert, B. Jan, P. Marco, T. Lothar, T. Gerhard, and ster, "The case for reconfigurable hardware in wearable computing." vol. 7: Springer-Verlag, 2003, pp. 299-308.

[48] S. W. Nawawi, M. N. Ahmad, and J. H. S. Osman, "Development of a Two-Wheeled Inverted Pendulum Mobile Robot," in *The 5th Student Conference on Research and Development -SCOReD 2007* Malaysia, 11-12 December 2007.

[49] S. Jeong and T. Takahashi, "Wheeled Inverted Pendulum Type Assistant Robot: Inverted Mobile, Standing, and Sitting Motions," in *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems* San Diego, CA, USA, Oct 29 - Nov 2, 2007.

[50] S. W. Nawawi, M. N. Ahmad, and J. H. S. Osman, "Real-Time Control of a Two-Wheeled Inverted Pendulum Mobile Robot," in *Proceedings of World Academy of Science, Engineering and Technology*, May 2008.

[51] D. P. Anderson, "nBot Balancing Robot, http://www.geology.smu.edu/~dpa-www/robo/nbot/."

[52] P. Deegan, B. J. Thibodeau, and R. Grupen, "Designing a Self-Stabilizing Robot for Dynamic Mobile Manipulation," in *Robotics: Science and Systems - Workshop on Manipulation for Human Environments*, 2006.

[53] I. Petersen, T. A. Johansen, J. Kalkkuhl, and J. Ludemann, "Wheel slip control in ABS using gain scheduled constrained LQR," in *ECC* Porto, 2001.

[54] V. Gavrilets, I. Martinos, B. Mettler, and E. Feron, "Control Logic for Automated Aerobatic Flight of a Miniature Helicopter " in *AIAA Guidance, Navigation, and Control Conference*, Monterey, California, 2002.

[55] A. Budiyono and H. Y. Sutarto, "Linear Parameter Varying Model Identification for Control of Rotorcraft-based UAV," in *The 5th Taiwan-Indonesia Workshop on Aeronautical Science, Technology and Industry* Taiwan, Nov. 13-16, 2006.

[56] R. Boualaga, B. Amar, M. Ammar, and L. Loron, "Parameters and States Estimation with Linear Quadratic Regulator Applied to Uninterruptible Power Supplies (UPS)," in *IECON 2006 - 32nd Annual Conference on IEEE Industrial*

Electronics, Nov. 2006, pp. 2055 - 2060.

[57] K. Shabaani and M. Jalili-Kharaajoo, "Application of adaptive LQR with repetitive control for UPS systems," in *Proceedings of 2003 IEEE Conference on Control Applications*, June 23-25, 2003, pp. 1124-1129

[58] J. S. Kelly, V. S. Rao, H. J. Pottinger, and H. C. Bowman, "Design and implementation of digital controllers for smart structures using field programmable gate arrays," *Smart Mater. Struct.,* vol. 6, pp. 559-572, 1997.

[59] J. Bae, A. Larson, and R. Voyles, "Wireless Video Sensor Networks over Bluetooth : High-Bandwidth Multi-Hop Networks for Resource-Constrained Robots," in *2007 IEEE International Conference on Robotics and Automation (ICRA 2007)*, Roma, Italy, April 2007.

[60] Xilinx, "Two flows for partial reconfiguration: Module based or difference based," in *Application Note 290, Xilinx*, 2004.

[61] S. Brooks, "Structure Charts and Basic Programming," *MATYC Journal,* vol. 15, pp. 107-112, 1981.

[62] C. Tomasi and T. Kanade, "Detection and Tracking of Point Features," in *Carnegie Mellon University Technical Report CMU-CS-91-132*, 1991.

[63] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, Fourth ed.: Prentice Hall, 2001.

[64] "http://www.engin.umich.edu/group/ctm/examples/pend/invPID.html."

[65] A. J. Laub, "A Schur method for solving algebraic Riccati equations," *IEEE Transactions on Automatic Control,* vol. 24, pp. 913- 921, Dec. 1979.

[66] "CLAPACK library, http://www.netlib.org/clapack/."

[67] Atmel, "ATmega128 datasheet, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf."

[68] M. G. Safonov and R. Y. Chiang, "A Schur Method for Balanced Model Reduction," *IEEE Transactions on Automatic Control,* vol. AC-34, pp. 729-733, July 1989.

Appendix A. Tutorials on FPGA-based dynamic reconfiguration

The basic tutorial on partial reconfiguration flow is explained in Xilinx application note, xapp290, which is based on modular design flow.

This modular design flow and source code is working fine in ISE 6.3 Xilinx software, but it doesn't support later version of ISE such as ISE 7.1, 8.1 or later.

From ISE 8.2, Xilinx starts to provide new way of design flow called Easy Access Partial Reconfiguration (EAPR).

But, it was not an easy work to build a working partial reconfiguration flow. Setting up the software environment was took considerable time. All the software chain need to match exact version. It can generate strange error while synthesizing or place and route. Furthermore, when you're using wrong version of software, even though you may not have any problem until the final stage of implementation, you can encounter a fatal error which you cannot identify at all.

Appendix B. Hardware block diagrams of modules for morphing bus

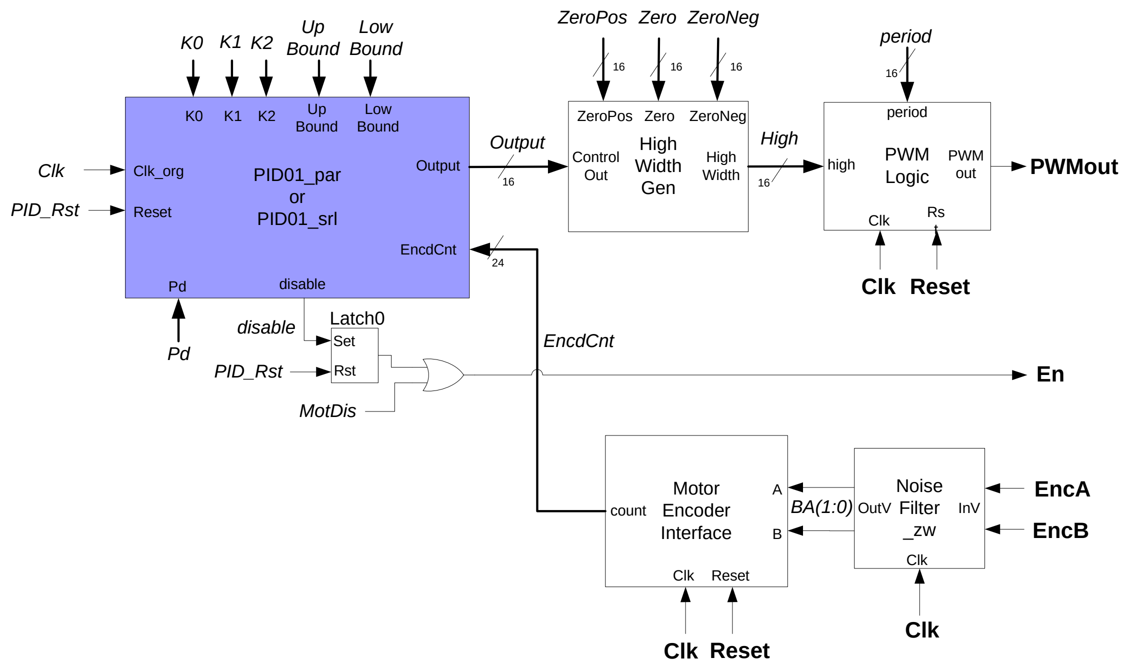


Figure 68. Top level module of PID algorithm.

Figure 69 shows the parallel design of a PID incremental algorithm. The design requires 4 adders and 3 multipliers. All bold signals are I/O ports, while others are internal signals.

The clock signal *clk* is used to control sampling frequency. *EncdCnt*, the encoder counter value, represents the current position *P*. The negation of *P*, *P_neg*, is generated by bit-wise complementing and adding 1. At the rising edge of control, signal *e(n)* of the last cycle is latched at register REG1, thus becomes *e(n-1)* of this cycle. In the same manner, *e(n-2)* and *u(n-1)* are recorded at REG3 and REG4 by latching *e(n-1)* and *u(n)* respectively. The registers can be set to initial values of 0 by asserting the reset signal, *Reset*. As long as the desired position P_d is also initialized to 0 when the system is reset, the control output is 0, which can keep the controlled object (e.g. the motor, in this system) static.

The computed control output variable *u(n)* may exceed the range that the controlled object can bear. *Bounder* is a value limitation logic that keeps the output in the user defined range of *UpBound* and *LowBound*.

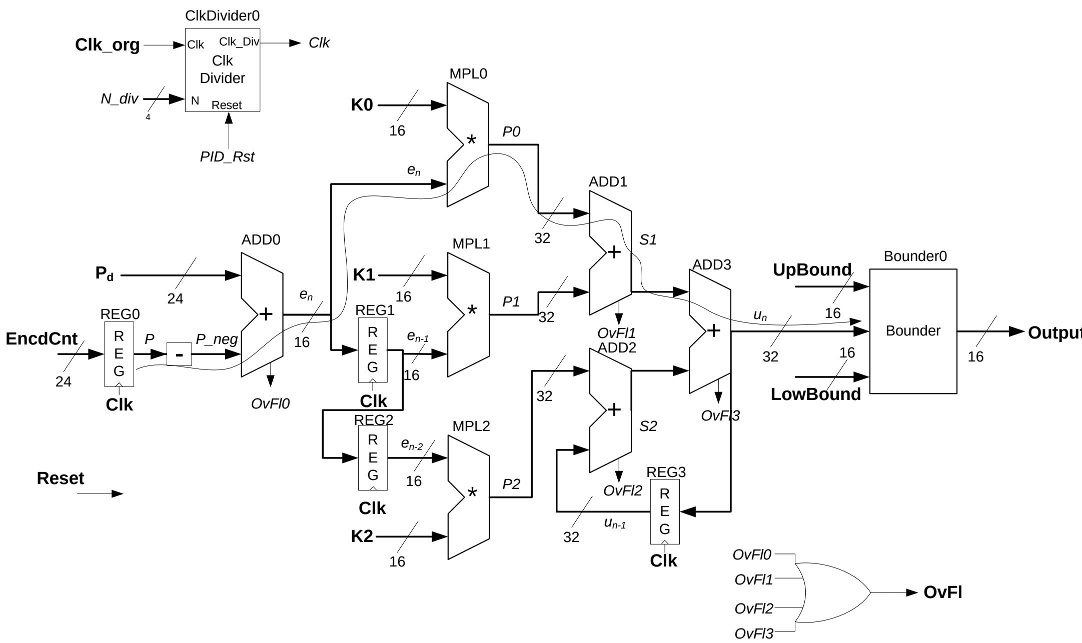


Figure 69. Parallel implementation of PID algorithm (PID01_par).

To minimize area, in the serial design only one arithmetic operator is used for each kind of arithmetic operation. As shown in Figure 70, there is one adder and one multiplier. Other parts, including arithmetic operators, registers, multiplexers and other logic, are called the datapath. Registers are used to store intermediate results. At the rising edge of the clock, the input signal of the register can be latched only if the load input signal is asserted. In each clock cycle, the control unit, which is a finite state machine, sets selection signals of multiplexers according to the current state and input, effectively defining the input to each operator.

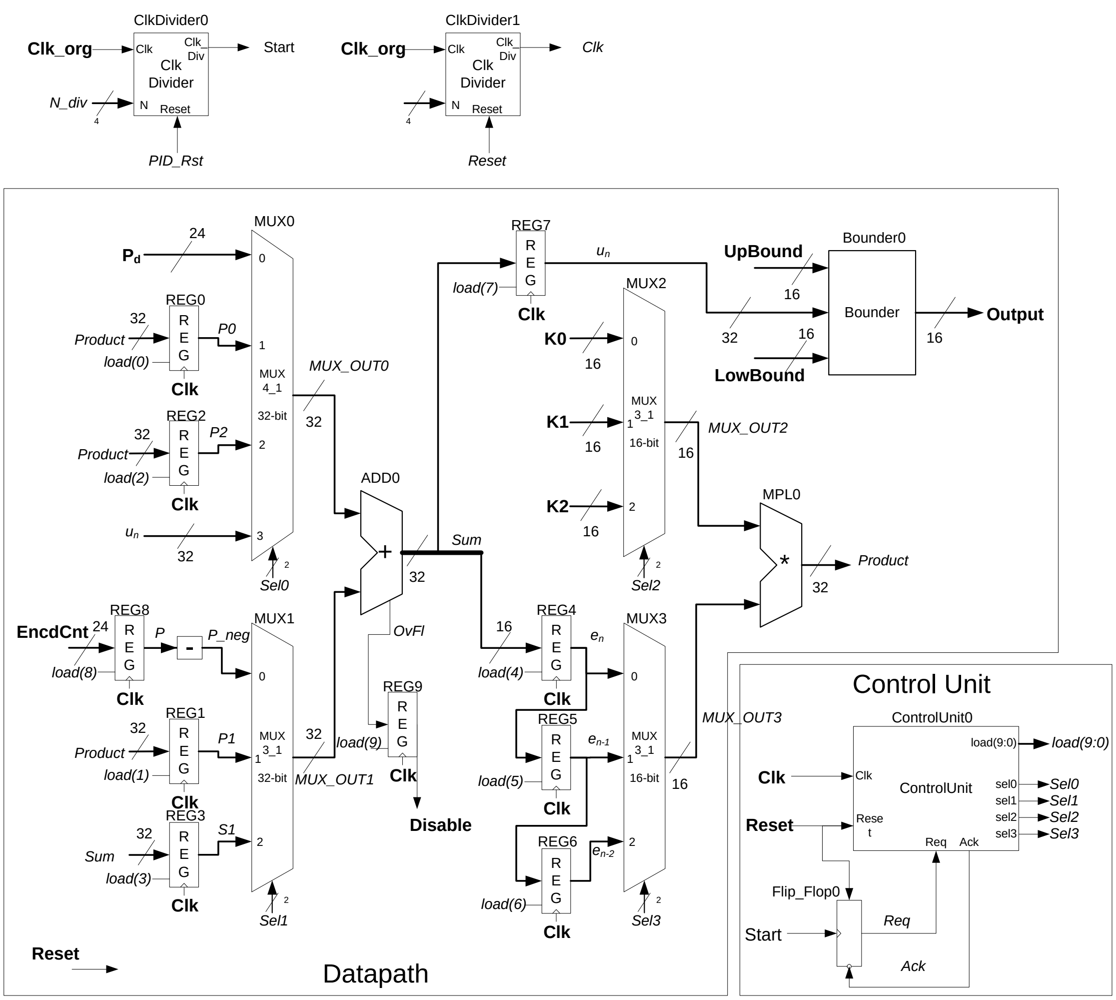


Figure 70. Serial implementation of PID algorithm (PID01_srl).

In multiple-channel design, either a PID unit is dedicated to each channel, referred to as Channel-Level Parallel (CLP) Design, or one PID unit is shared by all channels, referred to as Channel-Level Serial (CLS) Design. The tradeoffs are in area, speed, and complexity. A parallel design occupies quite a large area as the number of channels increase, whereas a serial design requires a more complex control unit and obviously takes longer to compute all channels. Because CLP designs are so straight-forward, only CLS designs are presented in this section. The PID units of each design can be either serial (referred to as serial PID based design) or parallel (referred to as parallel PID based design).

The serial PID based CLS design is shown in Figure 71. It requires two cycles for context switching in addition to the four cycles required for a serial PID control unit. One read cycle is required before the start of the PID algorithm to load both the previous computation results and the channel-specific parameters from RAM. Also, a write-back cycle is required after completion of the PID algorithm to store those data.

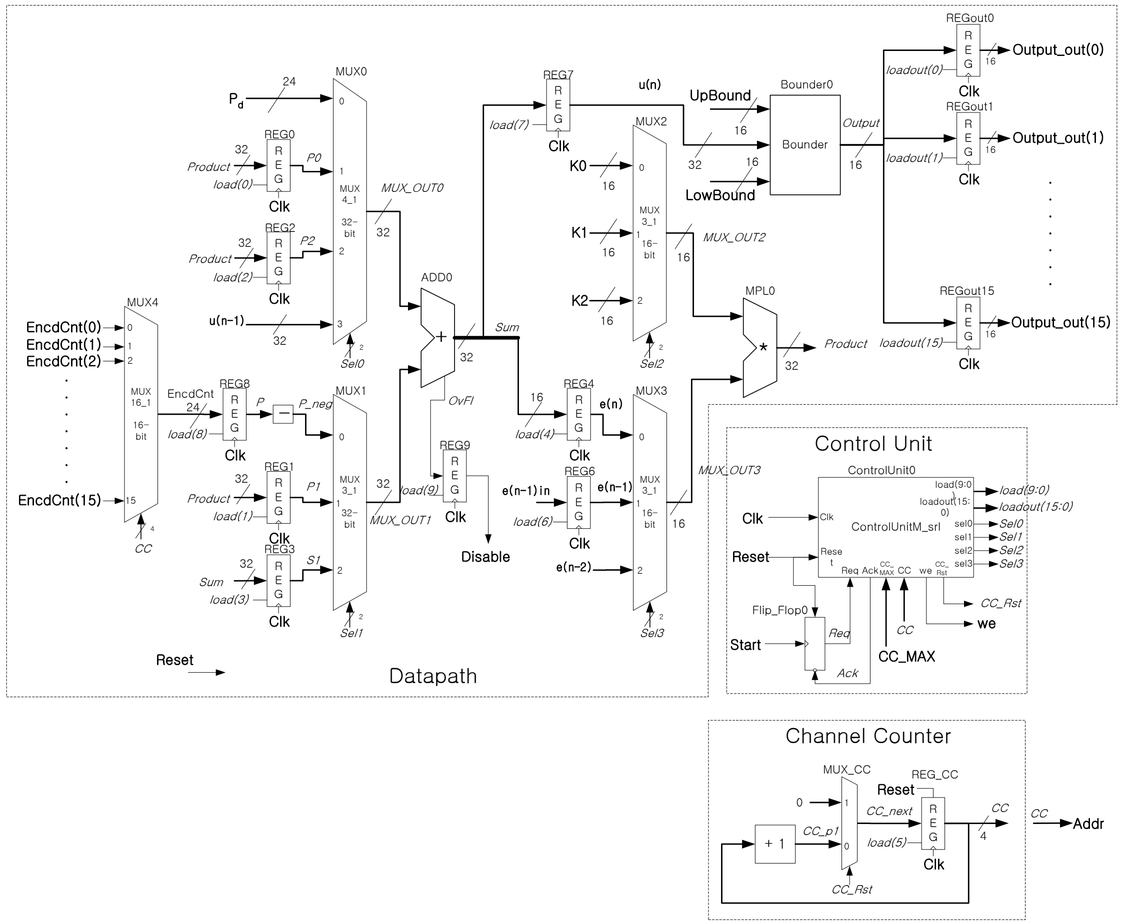


Figure 71. Serial PID based multiple-channel design.

The parallel PID based CLS design is shown in Figure 72. In this design, the PID algorithm is executed in only one cycle, but context switching is still required. The read cycles for both parallel and serial based designs are the same. The writing of the results to RAM could be implemented either as a separate write-back cycle or as part of the PID computation cycle. A separate cycle increases the cycle count for each channel, thus increases the delay, which is significant because the critical path delay of the parallel design is quite long relative to the serial design. Therefore, the write-back of the results is included in the PID algorithm.

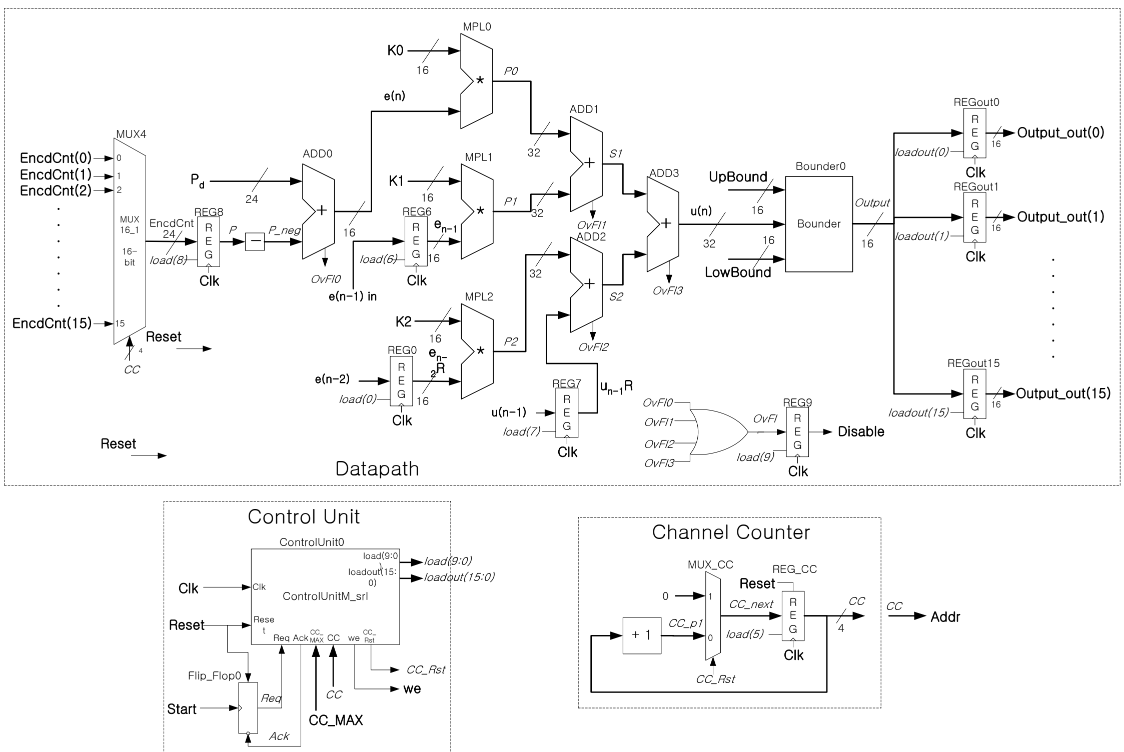


Figure 72. Parallel PID based multiple-channel design.

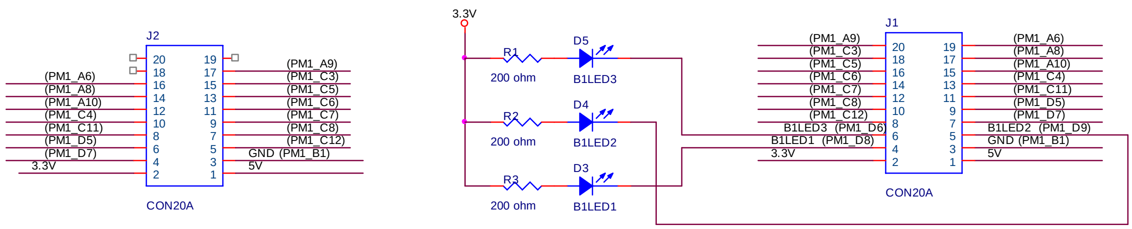


Figure 73. Test module #1: three LEDs

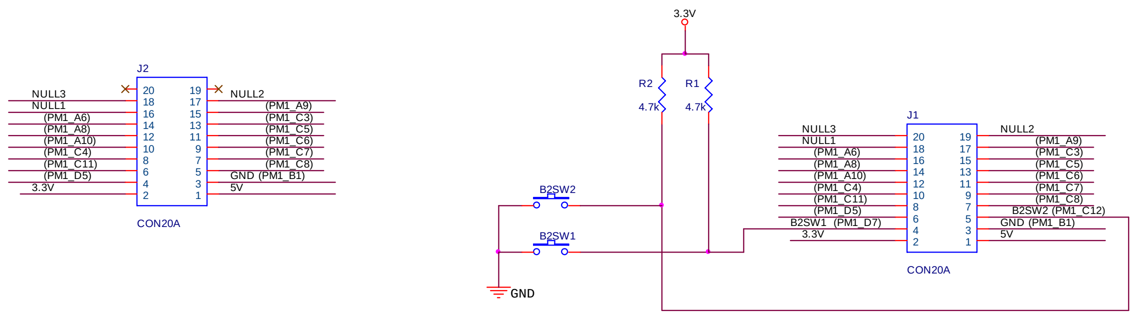


Figure 74. Test module #2: two switches.

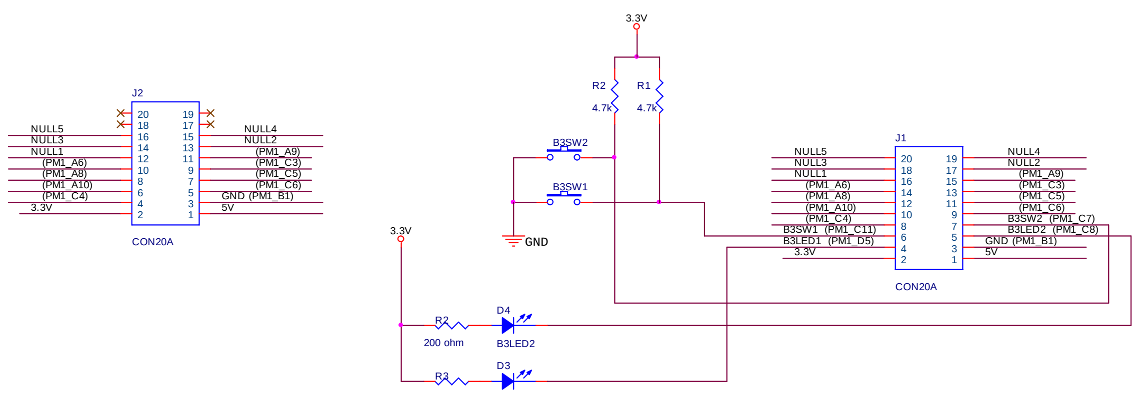


Figure 75. Test module #3: LEDs and switches.

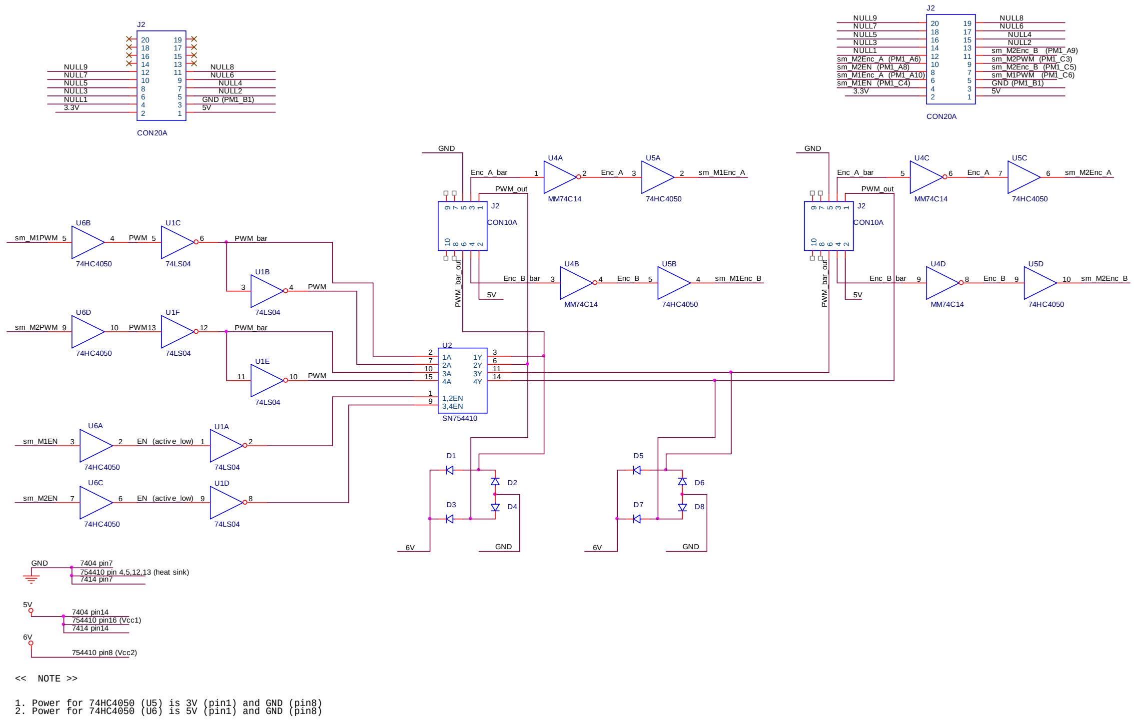


Figure 76. Test module #4: motor driver.

Appendix C. Modeling of balancing robot

This section provides modeling of a two-wheeled balancing robot. First, the equations of motion for wheels are obtained. The free body diagram of the robot is shown below in Figure 77. The lists of used symbols are shown in Appendix D.

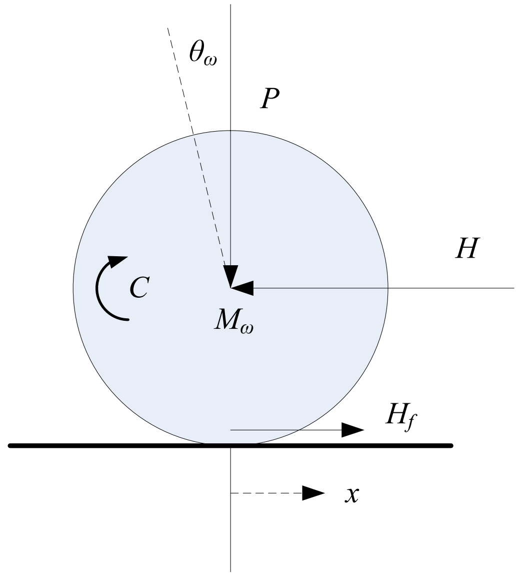


Figure 77. The free body diagram of wheeled balancing robot.

According to Newton's law of motion, the sum of forces on the horizontal x direction gives

$$\sum F_x = Ma$$

$$M_\omega \ddot{x} = H_f - H \tag{18}$$

Since there is no acceleration on the y direction, the sum of moments around the center of the wheel gives

$$\sum M_0 = Ia$$

$$I_\omega \ddot{\theta}_\omega = C - H_f r \tag{19}$$

From the DC motor dynamics, the output torque to the wheels can be obtained

$$C = I\frac{d\omega}{dt} = \frac{-k_m k_e}{R_m}\dot{\theta}_\omega + \frac{k_m}{R_m}V_a \tag{20}$$

Thus, plugging equation (20) into (19),

$$H_f = \frac{-k_m k_e}{R_m r}\dot{\theta}_\omega + \frac{k_m}{R_m r}V_a - \frac{I_w}{r}\ddot{\theta}_\omega \tag{21}$$

Substituting equation (21) into (18) to get the equation for the wheels,

$$M_\omega \ddot{x} = \frac{-k_m k_e}{R_m r}\dot{\theta}_\omega + \frac{k_m}{R_m r}V_a - \frac{I_w}{r}\ddot{\theta}_\omega - H \tag{22}$$

The angular rotation can be transformed into linear motion by simple transformation,

$$\ddot{\theta}_\omega r = \ddot{x}$$
$$\dot{\theta}_\omega r = \dot{x}$$

It yields,

$$\left(M_\omega + \frac{I_w}{r^2}\right)\ddot{x} = \frac{-k_m k_e}{R_m r^2}\dot{x} + \frac{k_m}{R_m r}V_a - H \tag{23}$$

Next, the robot chassis can be modeled as an inverted pendulum using Newton's law of motion.

The free body diagram of the chassis is shown below.

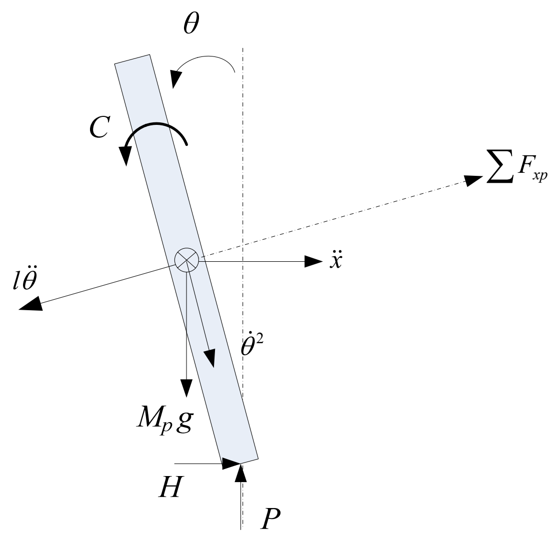


Figure 78: The free body diagram of the chassis.

The sum of forces in the horizontal direction:

$$\sum F_x = M_p \ddot{x} = H - [M_p l \ddot{\theta} \cos\theta + M_p l \dot{\theta}^2 \sin\theta] \tag{24}$$

The sum of forces perpendicular to the pendulum:

$$\sum F_{xp} = M_p \ddot{x} \cos\theta = H \cos\theta + P \sin\theta - M_p g \sin\theta - M_p l \ddot{\theta} \tag{25}$$

The sum of moment around the center of mass of pendulum:

$$-H\, l \cos\theta - P\, l \sin\theta = I_p\, \ddot{\theta} \tag{26}$$

Combining equation (25) and (26) the final equation is shown as below

$$I_p \ddot{\theta} + M_p g l \sin\theta + M_p l^2 \ddot{\theta} = -M_p l \ddot{x} \cos\theta \tag{27}$$

Using equation (23) and (24),

$$\left(M_w + \frac{I_w}{r^2}\right)\ddot{x} = \frac{-k_m k_e}{R_m r^2}\dot{x} + \frac{k_m}{R_m r}V_a - M_p\ddot{x} - M_p l\ddot{\theta}\cos\theta + M_p l\dot{\theta}^2\sin\theta \tag{28}$$

Rearranging these equations (27) and (28),

$$(I_p + M_p l^2)\ddot{\theta} + M_p gl\sin\theta = -M_p l\ddot{x}\cos\theta \tag{29}$$

$$\left(M_w + \frac{I_w}{r^2} + M_p\right)\ddot{x} + \frac{k_m k_e}{R_m r^2}\dot{x} + M_p l\ddot{\theta}\cos\theta - M_p l\dot{\theta}^2\sin\theta = \frac{k_m}{R_m r}V_a \tag{30}$$

Linearized by assuming $\theta = \pi + \phi$, where ϕ represents a small angle from the vertical upward direction.

Therefore, $\cos\theta = -1,\ \sin\theta = -\phi$ and $\left(\frac{d\theta}{dt}\right)^2 = 0$

The linearized equation of motion is

$$\ddot{\phi} = \frac{M_p l}{(I_p + M_p l^2)}\ddot{x} + \frac{M_p gl}{(I_p + M_p l^2)}\phi \tag{31}$$

$$\ddot{x} = \frac{M_p l}{\left(M_w + \frac{I_w}{r^2} + M_p\right)}\ddot{\phi} + \frac{-k_m k_e}{R_m r^2\left(M_w + \frac{I_w}{r^2} + M_p\right)}\dot{x} + \frac{k_m}{R_m r\left(M_w + \frac{I_w}{r^2} + M_p\right)}V_a \tag{32}$$

The continuous state space equation is obtained as:

$$\begin{bmatrix}\dot{x}\\ \ddot{x}\\ \dot{\phi}\\ \ddot{\phi}\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 0\\ 0 & \frac{-\left(I_p + M_p l^2\right)k_m k_e}{R_m r^2 Q_{eq}} & \frac{M_p{}^2 gl^2}{Q_{eq}} & 0\\ 0 & 0 & 0 & 1\\ 0 & \frac{-M_p l\ k_m k_e}{R_m r^2 Q_{eq}} & \frac{M_p gl\left(M_w + \frac{I_w}{r^2} + M_p\right)}{Q_{eq}} & 0\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\\ \phi\\ \dot{\phi}\end{bmatrix} + \begin{bmatrix}0\\ \frac{\left(I_p + M_p l^2\right)k_m}{R_m r Q_{eq}}\\ 0\\ \frac{M_p l k_m}{R_m r Q_{eq}}\end{bmatrix}V_a \tag{33}$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \phi \\ \dot{\phi} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} V_a \tag{34}$$

where, $Q_{eq} = \left[\left(M_w + \frac{I_w}{r^2} \right) \left(I_p + M_p l^2 \right) + M_p I_p \right]$

From the continuous state space equations (33) and (34), let's start to obtain transfer function between $X(s)$ and $V_a(s)$.

Laplace transform of equation (31) and (32) are as follows,

$$(I_p + M_p l^2) s^2 \Phi(s) = M_p l s^2 X(s) + M_p g l \Phi(s) \tag{35}$$

$$\left(M_w + \frac{I_w}{r^2} + M_p \right) s^2 X(s) = M_p l s^2 \Phi(s) + \frac{-k_m k_e}{R_m r^2} s X(s) + \frac{k_m}{R_m r} V_a(s) \tag{36}$$

When you rearrange equation (35),

$$\Phi(s) = \frac{M_p l s^2}{\left[(I_p + M_p l^2) s^2 - M_p g l \right]} X(s) \tag{37}$$

plugging equation (37) into (36) , an open-loop transfer function of position $X(s)$ and input $V_a(s)$ is obtained.

$$\frac{X(s)}{V_a(s)} = \frac{\frac{k_m}{R_m r} \left(I_p + M_p l^2 \right) s^2 - \frac{k_m}{R_m r} M_p g l}{Q_{eq} s^4 + \frac{k_m k_e}{R_m r^2} \left(I_p + M_p l^2 \right) s^3 - M_p g l \left(M_w + \frac{I_w}{r^2} + M_p \right) s^2 - \frac{k_m k_e}{R_m r^2} M_p g l \, s} \tag{38}$$

Appendix D. List of symbols for modeling of a balancing robot

x : Displacement

$\dot{x}$: Displacement velocity

ϕ : Angle

$\dot{\phi}$: Angular velocity

θ : Tilted angle of the chassis

θ_w : Rotation angle of the wheel

r : Radius of wheel

V_a : Applied torque

k_m : Torque constant

k_e : Back emf constant

R_m : Motor resistance

P : Reaction force between the wheel and the chassis of y-component of the force

H : Reaction force between the wheel and the chassis of x-component

H_f : Friction force between the ground and the wheel

C : Output torque

M_w : Mass of the wheel

M_p : Mass of the robot's chassis

g : Gravitational acceleration, 9.81 m/s^2

l : Distance between the center of the wheel and the robot's center of gravity

I_p : Inertia of the robot's chassis

I_w : Inertia of the wheel

Appendix E. Source code for MATLAB simulation

```
%  Resource-adaptive control implementation using LQR
%
clear all;

fprintf('\n\n========================\n');
display('Start of simulation !!!');

global Qx;
global Qy;
global R;
Qx=2000;
Qy=50;
R=1;
fprintf('\nOriginal values of Q and R -->  ');
fprintf('Qx : %6.1f   Qy : %6.1f   R : %3.1f \n', Qx, Qy, R );


global Mp;
Mp=0.5;
fprintf('Original values of Mp : %3.1f Kg \n', Mp);

f_LQR = 1;
f_stepspecs = 1;
X0 = [0 0 0 0];
```

```
Yall = [];
Tall = [0:.01:18];     % remove the ending integer to match with size of step input
signal
NumPeriod = ceil(Tall(end)/2);       % divide time period by two seconds which is
each simulation period

U1=0.4*ones(1,length([0:0.01:6])-1);
U2=0.2*ones(1,length([6:0.01:12])-1);
U3=0.4*ones(1,length([12:0.01:18])-1);
Uall = [U1, U2, U3];

for ii=1:NumPeriod
   Ttemp = find(Tall<=(ii-1)*2);
   Tstart = Ttemp(end);
   Ttemp = find(Tall<=ii*2);
   Tend = Ttemp(end)-1;
   T = Tall(Tstart:Tend);       % determine each simulation time, T
   U = Uall(Tstart:Tend);

   if ii == 3
      Mp = 3;            % increased mass
      f_LQR = 1;          % since there is change in system model, I need to
recalculate LQR to obtain new K
      fprintf('---------------------\n');
      fprintf('Change mass of robot chassis to %3.1f Kg at time %3.1f seconds \n',
Mp, (ii-1)*2);
   end

   if f_LQR == 1
      f_LQR = 0;
      [Ac, Bcn, Cc, Dc] = LQRcal();
   end

   % ---------------------------------------------------------------
   % step response
   [Y,X]=lsim(Ac,Bcn,Cc,Dc,U,T,X0);
   Yall = [Yall; Y];
   X0 = X(end,:);      % initial value for step response of the next period

   %-----------------------------------------------------------
   % measure performance
   %-------------------------
   if mod(ii,3)==1           % generate new step input every 6 seconds
```

```
      [x_overst,x_Ts,x_Tr]=stepspecs(T,Y(:,1)');
      display('Controller performance ...');
      fprintf('rise time: %6.4f   settling time: %6.4f \n', x_Tr, x_Ts);

      %############
      % check if plant change
      if x_Tr >= .4 | x_Ts >= 1
         f_LQR = 1;        % need to run LQR again to recalculate K
         R = R*.6;
         %Q = Q*1.5;
         fprintf('---------------------\n');
         fprintf('Since design specification does not meet, Modify Q or R  ...\n');
         fprintf('Qx : %6.1f   Qy : %6.1f   R : %4.2f \n\n', Qx, Qy, R );
      end

   end

end


figure(1);
plot(Tall(1:end-1),Yall);
legend('robot position', 'tilt angle', 'Location','SouthEast');
```

Appendix F. How to compile a code in VisualC using CLAPACK

A method of how to compile a code to solve Riccati equation in VisualC with CLAPACK library is presented in this appendix.

When compiling any code using CLAPACK, special consideration to setup LIB directory is required.

a. Add '.\LIB' directory in C/C++ general configuration properties (refer to Figure 79).

b. Add '.\LIB' directory in Linker properties (refer to Figure 80).

c. Add specific LIB files in Linker Input tab (refer to Figure 81) which are 'BLASd.lib clapackd.lib libf2cd.lib' for Additional Dependencies and 'LIBCMTd.lib ; LIBCMT.lib' for 'Ignore Specific Library'.

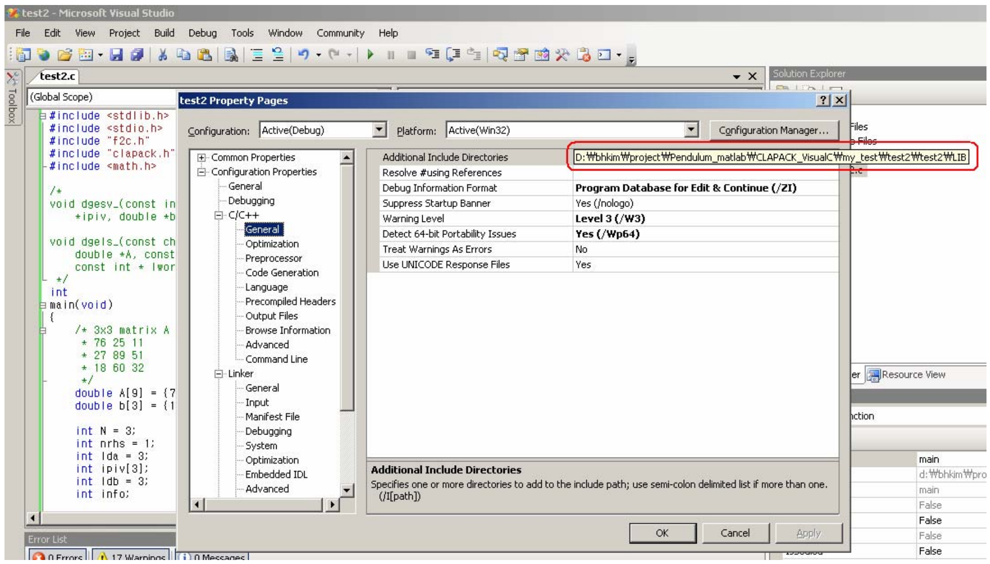


Figure 79. Screen shot for C/C++ general configuration setup.

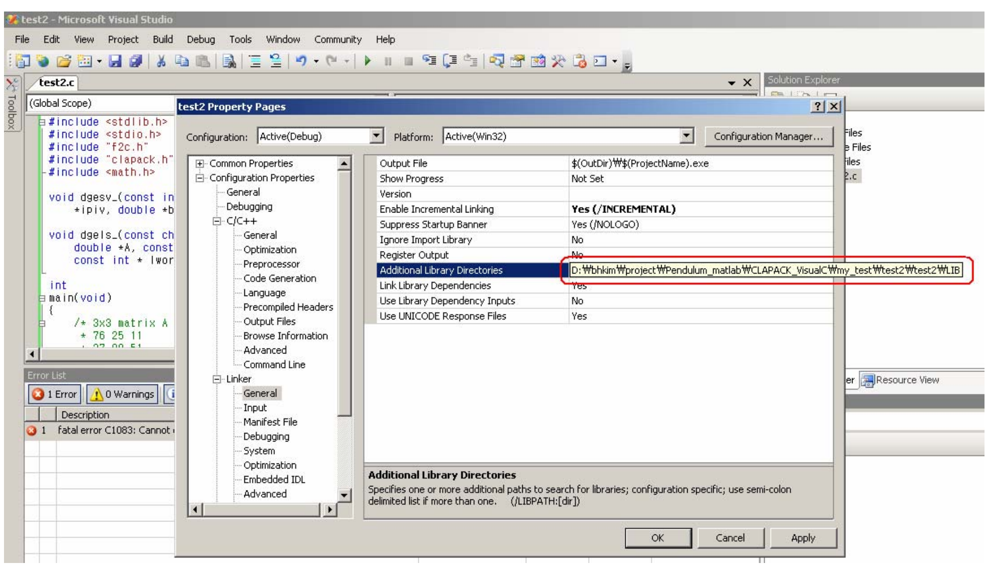


Figure 80. Screen shot for Linker setup.

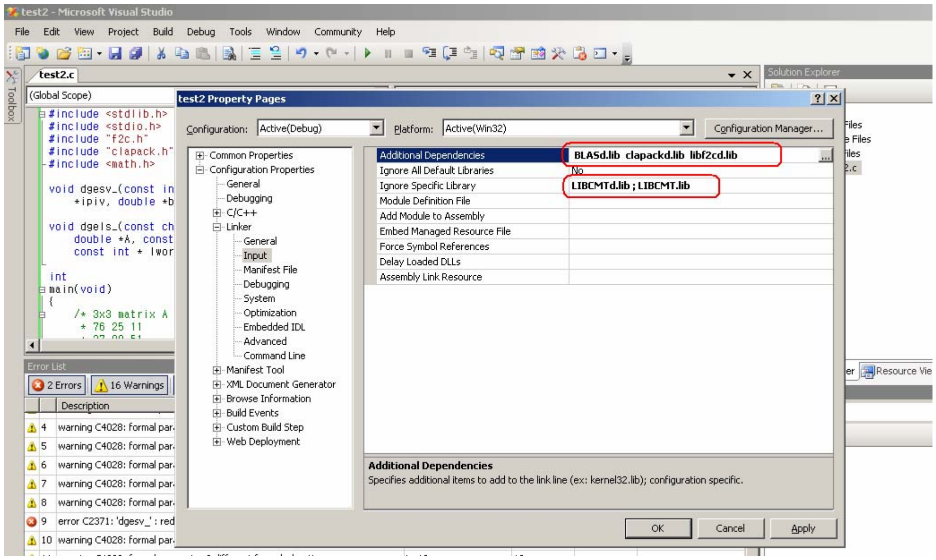


Figure 81. Screen shot for Linker input setup.

Appendix G. Source code to implement LQR control using VisualC

Source code to implement LQR control using CLAPACK library (version 3.1.1 for windows) and Microsoft Visual Studio 2005 software is presented in this appendix. The screen capture of results is shown in Appendix H.

```
//=============================================================
// Objective : to implement LQR control using CLAPACK library
//
//   Version 1.1 (5/7/08)
//   -- try with single precision
//   Version 1.0 (5/4/08)
//   -- I used 'dgees' subroutine in CLAPACK to run schur decomposition
//
#include <stdlib.h>
#include <stdio.h>

#include "blaswrap.h"
#include "f2c.h"
```

```
#include "clapack.h"
//#include <math.h>

///////////////////////////////////////////
#define N_  8

#define NMAX N_
#define NxN (N_*N_)
#define NA_ (N_/2)    // size of row of A matrix
#define NAxNA (NA_*NA_)

#define LWORKB  (3*N_*3*N_)
#define IWORKB   (3*N_*3*N_)

#define TRUE 1
#define FALSE 0

#define Allocate(n,t) (t*)malloc((n)*sizeof(t))

///////////////////////////////////////////////////
//  subroutine declaration
//
void printm(real *a, int row, int col);
void printm_int(integer *a, int row, int col);
void matrix_mult(real *a, int a_row, int a_col, real *b, int b_row, int b_col, real
*c);
real* MatrixCopy(real *m, int row, int col);
real* gauss_elim(real *A, int A_row, int A_col, real *b, int b_size);
int MatrixInvert(real *A, real *invA, int row, int col);

///////////////////////////////////////////////////
//  main routine
//
//
int main(void)
{

   /* 3x3 matrix A
       [ 76 25 11;
         27 89 51;
         18 60 32]    */
       //static real  M[NxN] = {
                                 76, 27, 18, 25, 89, 60, 11, 51, 32};
```

```
	// Hamiltonian matrix 'M'
   static real  M[NxN] =   {
0.0000,  0.0000,  0.0000,  0.0000,  -2000.0000,  0.0000,  0.0000,  0.0000,  1.0000,  -
0.5549,  0.0000,  -5.0441,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  5.5612,  0.0000,
139.6469, 0.0000, 0.0000, -50.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000,
0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  -1.0000,  0.0000,
0.0000,  0.0000,  -11.0831,  0.0000,  -100.7555,  0.0000,  0.5549,  -5.5612,  0.0000,
0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, -1.0000, 0.0000, -100.7555,
0.0000, -915.9592, 0.0000, 5.0441, -139.6469, 0.0000 };

	//static real  M[N_*N_] =  {
0.35,0.09,  -0.44,  0.25,  0.45,0.07,  -0.33,  -0.32,  -0.14,-0.54,-0.03,  -0.13,  -0.17,
0.35, 0.17,0.11};

	static real wr[N_];
	static real wi[N_];
	static real work[LWORKB]; // [(2+NxN)*NMAX];
	static real us[NMAX*N_];

	static integer iwork[IWORKB];

	static real us11[NAxNA], us21[NAxNA];
	static real inv_us11[NAxNA];
	static real P[NAxNA] =  {
				0.0, };

	static real Beye[NAxNA] =  {
				 1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1};
	static real B_tr[NA_] =   {
				  0, 3.3291, 0, 30.2648};
	static real K[NA_];

   int N = N_;
	int NA = NA_;
	int Nunit = 1;
	int lda = NMAX;

	/* Local variables */
	static integer i, j;
	static integer  info, sdim;
	static integer ldus = NMAX;
	static logical bwork[N_];
```

```
        static integer lwork = 3*N_; //  (2+NMAX)*NMAX;
        static integer liwork = 3*N_; //  (2+NMAX)*NMAX;


        extern logical select();


        printf(" \n original M is ... \n");
        printm(M, N, N);


        // 'dgees' subroutine in CLAPACK
        // input:  'M'  -- Hamiltonian matrix
        // output: 'M'  -- quasitriangular Schur matrix T
        //        'us' -- unitary matrix U
        sgees_("V", "S", (L_fp)select, &N, M, &lda, &sdim, wr, wi, us, &ldus, work,
&lwork, bwork, &info);

        for (i=0;i<NA;i++)
                for(j=0;j<NA;j++)
                        us11[i+j*NA] = us[i+j*N_];       //     U11 = US(1:N,1:N);
        for (i=0;i<NA;i++)
                for(j=0;j<NA;j++)
                        us21[i+j*NA]   =   us[i+(j*N_+NA)];        //        U21   =
US((N+1):N*2,1:N);

        MatrixInvert(us11, inv_us11, NA, NA);     // inv(U11) :   gauss elimination

        // P matrix is the solution of riccati equation
        matrix_mult(us21, NA, NA, inv_us11, NA, NA, P);     //   P = U21 * inv(U11);

        // K is the state feedback gain vector for LQR control
        matrix_mult(B_tr, Nunit, NA, P, NA, NA, K);       //   K = inv(R)*B'*P

    if(info == 0) /* succeed */
        {
                printf("\n success !! \n");
                printf(" \nQuasitriangular Schur matrix T is ... \n");
                printm(M, N, N);

                printf(" \nUnitary matrix U is ... \n");
                printm(us, N, N);
```

```
                printf("\n U11 ..\n");
                printm(us11, NA, NA);

                printf("\n U21 ..\n");
                printm(us21, NA, NA);

                printf(" \n P matrix is ... \n");
                printm(P, NA, NA);

                printf("\n K ..\n");
                printm(K, Nunit, NA);

        }
    else
                fprintf(stderr, " error :  %d\n", info);

        printf("\n Press any key to exit !! ");
        getch();

    return info;
}



logical select(real *ar, real *ai)
{
        logical d, select_;
        if (*ar < 0 )     // check if the real part is negative  !!! This is used in
reordering schur matrix
                d = TRUE;
        else
                d = FALSE;

        select_ = d;
        return select_;
}


void printm(real *a, int row, int col)
{
        int i, j;
```

```
        printf("[");      // for MATLAB
        for (i=0;i<row;i++)
        {
                for(j=0;j<col;j++)
                {
                        printf(" % 8.3f",a[i+j*row]);
                }
                printf(";\n");   // for MATLAB
        }
        printf("]\n");        // for MATLAB
}

void printm_int(integer *a, int row, int col)
{
        int i, j;

        printf("[");       // for MATLAB
        for (i=0;i<row;i++)
        {
                for(j=0;j<col;j++)
                {
                        printf("% 6d  ",a[i+j*row]);
                }
                printf(";\n");    // for MATLAB
        }
        printf("]\n");       // for MATLAB
}


//----------------------------------------------------------------------
// perform matrix multiply,  c = a * b
//
//    a matrix has dimension of a_row and a_col
//        b matrix has dimension of b_row and c_col
//        c matrix has dimension of a_row and b_col
//        and a_col and b_row should be the same.
//
//        input : a matrix,  b matrix
//        output : c matrix
//
void matrix_mult(real *a, int a_row, int a_col, real *b, int b_row, int b_col, real
*c)
{
```

```
        int i, j, k;
        real temp=0.0;

        for (i=0;i<a_row;i++)
        {
                for(j=0;j<b_col;j++)
                {
                        temp = 0.0;
                        for(k=0;k<a_col;k++)
                                temp += a[i+k*a_row]*b[k+j*b_row];

                        c[i+j*a_row] = temp;
                }
        }
}

//----------------------------------------------------------------------
//  Gauss Elimination method
//
//    input A, b --  A is a matrix (A_row x A_col),  b is a vector (b_size)
//    output : x -- x has dimension of A_row x 1
//
real* gauss_elim(real *A, int A_row, int A_col, real *b, int b_size)
{
        real *a, *m;    // matrix
        real *x;       // vector

        int i,j,k,n;

        a = MatrixCopy(A, A_row, A_col);  // allocate memory for 'a' matrix in the
'MatrixCopy' subroutine

        m = Allocate(A_row*A_col, real);
        x = Allocate(A_row, real);

        n = A_row;

        for(k = 0;k<n-1;k++){
                for(i = k+1;i<n;i++){
                        m[i+k*A_row] = a[i+k*A_row]/a[k+k*A_row];
                        for(j=k+1;j<n;j++)
                                a[i+j*A_row]         =         a[i+j*A_row]         -
```

```
m[i+k*A_row]*a[k+j*A_row];
                        b[i] = b[i] - m[i+k*A_row]*b[k];
                }
        }

        // back-substitution for upper triangular matrix 'a'

        x[n-1] = b[n-1]/a[(n-1)+(n-1)*A_row];

        for(i=0;i<n-1;i++){
                x[n-2-i] = b[n-2-i];
                for(j=n-i-1;j<n;j++)
                        x[n-2-i] = x[n-2-i] - a[(n-2-i)+j*A_row]*x[j];
                x[n-2-i] = x[n-2-i]/a[(n-2-i)+(n-2-i)*A_row];
        }

        free(a);
        free(m);
        return x;
}

//----------------------------------------------------------------------
// MatrixInvert: Performs matrix inversion using gaussian elimination.
//
int MatrixInvert(real *A, real *invA, int row, int col)
{
        real *b, *x;    // vectors having dimension of row x 1
        int i,k;

        if(row != col)    // matrix must be square in order to invert
                return -1;

        for(i=0;i<col;i++){
                b = Allocate(row, real);
                for(k=0;k<row;k++)  b[k]  =  0;             //  'b'  is  vector  having
dimension of row x 1
                b[i] = 1;

                x = gauss_elim(A, row, col, b, row);    //    A*x = b,  A and b are
inputs and x is solution

                // output of gauss_elim is vector 'x' and
                // this 'x' is each column of inverse matrix of 'A'
```

```
                for( k=0;k<row;k++) invA[k+i*row] = x[k];

                free(b);
        }

        return 0;
}



// -------------------------------------------------------------------
// copy all the data from source matrix 'm' to destination matrix 'n'
//
real* MatrixCopy(real *m, int row, int col)
{

        real *n;     // target matrix
        int i,j;

        n = Allocate(row*col, real);

        for(i=0;i<row;i++)
                for(j=0;j<col;j++)
                        n[i+j*row] = m[i+j*row];

        return n;
}
```

Appendix H. Output screen capture of implementing LQR control

This appendix is the screen capture of the result of source code in Appendix G, which is to implement LQR control using CLAPACK library. At the end of this output, K matrix can be found and it shows the same result of MATLAB LQR control routine.

```
 Hamiltonian matrix M is ...
```

```
[    0.000    1.000    0.000    0.000    0.000    0.000    0.000    0.000;
    0.000   -0.555    5.561    0.000    0.000  -11.083    0.000 -100.756;
    0.000    0.000    0.000    1.000    0.000    0.000    0.000    0.000;
    0.000   -5.044  139.647    0.000    0.000 -100.756    0.000 -915.959;
 -2000.000    0.000    0.000    0.000    0.000    0.000    0.000    0.000;
    0.000    0.000    0.000    0.000   -1.000    0.555    0.000    5.044;
    0.000    0.000  -50.000    0.000    0.000   -5.561    0.000 -139.647;
    0.000    0.000    0.000    0.000    0.000    0.000   -1.000    0.000;
]

Quasitriangular Schur matrix T is ...
[  -13.962   20.177  -19.145   32.795  252.125  713.183  550.325  952.272;
   -3.864  -13.962   17.642   33.838  127.017 -365.777  -45.575  1541.543;
    0.000    0.000   -5.976   19.486   13.361  -26.575   85.268  721.343;
    0.000    0.000   -0.662   -5.976  -17.610   15.553  -19.374 -230.326;
    0.000    0.000    0.000    0.000   13.962   27.087   15.008    3.117;
    0.000    0.000    0.000    0.000   -2.878   13.962    4.774  -34.176;
    0.000    0.000    0.000    0.000    0.000    0.000    5.976   39.148;
    0.000    0.000    0.000    0.000    0.000    0.000   -0.329    5.976;
]

Unitary matrix U is ...
[   -0.005    0.000    0.000    0.025    0.119    0.050    0.148    0.980;
    0.072   -0.108    0.084   -0.308   -0.210    0.507   -0.752    0.122;
   -0.050   -0.037    0.104   -0.153    0.961    0.081   -0.154   -0.094;
    0.836   -0.490   -0.216    0.069    0.069   -0.030    0.057   -0.013;
   -0.541   -0.751   -0.360    0.115   -0.004    0.016   -0.020   -0.003;
   -0.020   -0.071   -0.139   -0.925   -0.078   -0.244    0.231    0.011;
   -0.021   -0.420    0.885   -0.039   -0.090   -0.045    0.167   -0.011;
    0.005   -0.023    0.056    0.073   -0.003   -0.820   -0.551    0.123;
]

 U11 is ..
[   -0.005    0.000    0.000    0.025;
    0.072   -0.108    0.084   -0.308;
   -0.050   -0.037    0.104   -0.153;
    0.836   -0.490   -0.216    0.069;
]

 U21 is ..
[   -0.541   -0.751   -0.360    0.115;
   -0.020   -0.071   -0.139   -0.925;
   -0.021   -0.420    0.885   -0.039;
```

```
    0.005   -0.023    0.056    0.073;
]

 P matrix is ...
[  696.468  121.252 -135.535  -14.815;
  121.252   28.315  -32.382   -3.635;
 -135.535  -32.382   44.905    4.592;
  -14.815   -3.635    4.592    0.500;
]

 K is ..
[  -44.725  -15.741   31.185    3.031;
]

 Press any key to exit !!
```

Appendix I.   Atmel source code

Source code to implement LQR control using schur decomposition method is presented in this appendix. This program is developed with the target board of CEREBOT II from Digilent Inc, which contains Atmel ATmega64 AVR processor with 64KB flash, 4KB SRAM, and 2KB EPROM.

```
/*****************************************************************/
/*  test p/g for CEREBOT II from Digilent Inc.                   */
/*                                                               */
/*  function: Implement LQR control using schur decomposition   */
/*****************************************************************/
#include <inttypes.h>
#include <stdio.h>
#include <avr/interrupt.h>
#include <compat/deprecated.h>
```

```
#include "blaswrap.h"
#include "f2c.h"
#include "clapack.h"

/////////////////////////////////

#define MYBPS      9600UL
#define SYSCLK 8000000UL                       //  fosc = 8MHz

////////////////////////////////////////////
#define N_  8

#define NMAX N_
#define NxN (N_*N_)
#define NA_ (N_/2)   // size of row of A matrix
#define NAxNA (NA_*NA_)

#define LWORKB  (3*N_*3*N_)
#define IWORKB   (3*N_*3*N_)

#define TRUE 1
#define FALSE 0

#define Allocate(n,t) (t*)malloc((n)*sizeof(t))

//extern int sgees_(char *jobvs, char *sort, L_fp select, int *n, real *a, int
*lda, int *sdim, real *wr, real *wi, real *vs, int *ldvs, real *work, int *lwork,
logical *bwork, int *info);

////////////////////////////////////////////////////
//  subroutine declaration
//
void printm(real *a, int row, int col);
void printm_int(integer *a, int row, int col);
void matrix_mult(real *a, int a_row, int a_col, real *b, int b_row, int b_col, real
*c);
real* MatrixCopy(real *m, int row, int col);
real* gauss_elim(real *A, int A_row, int A_col, real *b, int b_size);
int MatrixInvert(real *A, real *invA, int row, int col);

/////////////////////////////
int uart_putchar1(char c)       // serial port 1
```

```
{
  if (c == '\n')
    uart_putchar1('\r');
  loop_until_bit_is_set(UCSR1A, UDRE1);
  UDR1 = c;
  return 0;
}

char uart_getchar1(void)       // serial port 0
{
  //while ( !(UCSR0A & (1<<RXC0)) ) ;
  loop_until_bit_is_set(UCSR1A, RXC1);
  return UDR1;
}

/////////////////////////////////////////////////////////
void delay_ms(uint16_t delay) {

        uint16_t i;
        while(delay > 0){
                for( i = 0; i < 390; i ++){
                        ;;
                }
                delay -= 1;
        }
}

void delay_ms2(uint16_t delay) {
        uint16_t i;
        while(delay > 0){
                for( i = 0; i < 39; i ++){
                        ;;
                }
                delay -= 1;
        }
}

////////////////////////////////////////////////

int main(void)
{
// variables for cerebot ////////////////////
        uint16_t baud;
```

```
// variables for clapack //////////////////////


        static real  M[NxN] = {
0.0000,  0.0000,  0.0000,  0.0000,  -2000.0000,  0.0000,  0.0000,  0.0000,  1.0000,  -
0.5549,  0.0000,  -5.0441,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  5.5612,  0.0000,
139.6469, 0.0000, 0.0000, -50.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000,
0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  -1.0000,  0.0000,
0.0000,  0.0000,  -11.0831,  0.0000,  -100.7555,  0.0000,  0.5549,  -5.5612,  0.0000,
0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, -1.0000, 0.0000, -100.7555,
0.0000, -915.9592, 0.0000, 5.0441, -139.6469, 0.0000 };

                static real wr[N_];
        static real wi[N_];
        static real work[LWORKB]; // [(2+NxN)*NMAX];
        static real us[NMAX*N_];

        static integer iwork[IWORKB];

        static real us11[NAxNA], us21[NAxNA];
        static real inv_us11[NAxNA];
        static real P[NAxNA] = {
                            0.0, };

        static real Beye[NAxNA] = {
                          1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1};
        static real B_tr[NA_] =
                          {
                            0, 3.3291, 0, 30.2648};
        static real K[NA_];

        int N = N_;
        int NA = NA_;
        int Nunit = 1;
        int lda = NMAX;

        /* Local variables */
        static integer i, j;
        static integer  info, sdim;
        static integer ldus = NMAX;
        static logical bwork[N_];
```

```
        static integer lwork = 3*N_; //  (2+NMAX)*NMAX;
        static integer liwork = 3*N_; //  (2+NMAX)*NMAX;


        // extern logical select();
        extern logical select(real *ar, real *ai);


////////////////////////////////////////////////
        delay_ms(20);   // wait until the board is ready

//  serial communication setup
//////////////////////////////////////////////////////////

        baud = (SYSCLK / (16 * MYBPS)) - 1;

  UBRR1H = (unsigned char)(baud >> 8);
  UBRR1L = (unsigned char)baud;

  UCSR1B = (1<<RXEN1) | (1<<TXEN1);
  UCSR1C = (3<<UCSZ10);
  sei ();

///////////////////////////////////////////////////////////////////////////
/*=======================================================*/
        printf("\r\n\r\n\r\n BKKIM>> test v4  .. ");
        printf("\r\n press any key to start .. ");

// start of routine of LQR control using clapack /////////////////////
        // 'sgees_' subroutine in CLAPACK
        // input:  'M'  -- Hamiltonian matrix
        // output: 'M'  -- quasitriangular Schur matrix T
        //        'us' -- unitary matrix U
        sgees_("V", "S", (L_fp)select, (integer *)&N, M, (integer *)&lda, &sdim, wr,
wi, us, &ldus, work, &lwork, bwork, &info);

        for (i=0;i<NA;i++)
                for(j=0;j<NA;j++)
                        us11[i+j*NA] = us[i+j*N_];      //    U11 = US(1:N,1:N);
        for (i=0;i<NA;i++)
                for(j=0;j<NA;j++)
                        us21[i+j*NA]   =   us[i+(j*N_+NA)];        //        U21   =
US((N+1):N*2,1:N);
```

```
        return 0;
}

logical select(real *ar, real *ai)
{
        logical d, select_;
        if (*ar < 0 )     // check if the real part is negative  !!! This is used in
reordering schur matrix
                d = TRUE;
        else
                d = FALSE;

        select_ = d;
        return select_;
}
```

Appendix J. Procedure of square-root balanced truncation model reduction

Model reduction is to remove states from the original system and to compute reduced-order system while maintaining similar characteristics from the original system. Square-root balanced truncation model reduction is used in this study, which is implemented in Matlab and has function name of 'BALMR'.

In balanced truncation model reduction, *k*th order reduced model $G_M(s)$ is computed from a possibly non-minimal and not necessarily stable, *n*th order system $G(s)$ such that

$$\begin{aligned}
&G_M(s) = C_M(Is - A_M)^{-1}B_M + D_M \\
&G(s) = C(Is - A)^{-1}B + D \\
&\|G(jw) - G_M(jw)\|_\infty \le totbnd, \\
&totbnd = 2\sum_{i=k+1}^{n} svh(i)
\end{aligned} \tag{39}$$

The n-vector svh contains the Hankel singular values of the stable and antistable parts of

$G(jw)$, which are the square-roots of eigenvalues of their reachability and observability grammians. Since the original system is unstable, $G(s)$ will be splitted into the sum of stable and antistable parts.

The Matlab procedure of 'BALMR' balanced-truncation model reduction is as follows.

1. Split the system into stable and antistable part. Use Matlab function 'stabproj'.

$$G(s) = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \quad \text{: original system} \tag{40}$$

, where the matrix symbol $\left[\begin{array}{c|c} & \\ \hline & \end{array}\right]$ denotes state space realization of the given system.

$$G(s) = G_L(s) + G_R(s) = \left[\begin{array}{c|c} A_L & B_L \\ \hline C_L & D_L \end{array}\right] + \left[\begin{array}{c|c} A_R & B_R \\ \hline C_R & D_R \end{array}\right] \tag{41}$$

: stable system and antistable system

Where, $V^T A V = \begin{bmatrix} A_L & A_{12} \\ 0 & A_R \end{bmatrix}$, V is unitary transformation matrix in a block ordered schur form.

$$\begin{bmatrix} x_L \\ x_R \end{bmatrix} = V^T x \tag{42}$$

, where x_L is state variable for the stable system and x_R is state variable for the antistable system.

2. Find hankel singular values and grammians P,Q for each subsystem. Use Matlab function 'hksv'.

HSV_L , P_L , Q_L
HSV_R , P_R , Q_R

3. Find smallest hankel singular value which will be used for model reduction.

4. Run balanced-trunctation model reduction for each system

$$G_F(s) = G_{LF}(s) + G_{RF}(s) = \left[\begin{array}{c|c} A_{LF} & B_{LF} \\ \hline C_{LF} & D_{LF} \end{array}\right] + \left[\begin{array}{c|c} A_{RF} & B_{RF} \\ \hline C_{RF} & D_{RF} \end{array}\right] \tag{43}$$

where $\left[\begin{array}{c|c} A_{LF} & B_{LF} \\ \hline C_{LF} & D_{LF} \end{array}\right] = \left[\begin{array}{c|c} S_{L,L}{}^T A_L S_{L,R} & S_{L,L}{}^T B_L \\ \hline C_L S_{L,R} & D_L \end{array}\right]$

$$\left[\begin{array}{c|c} A_{RF} & B_{RF} \\ \hline C_{RF} & D_{RF} \end{array}\right] = \left[\begin{array}{c|c} S_{R,L}{}^T A_R S_{R,R} & S_{R,L}{}^T B_R \\ \hline C_R S_{R,R} & D_R \end{array}\right]$$

, where $S_{L,L}$ means the basis for the left eigenspace for the stable system. $S_{L,R}$ means the basis for the right eigenspace for the stable system. $S_{R,L}$ means the basis for the left eigenspace for the antistable system. $S_{R,R}$ means the basis for the right eigenspace for the antistable system. Refer to [68] for further detail.

In order to find the transformation matrix for the state variable, equation for C_{LF} or C_{RF} can be considered. Compare the output equation for $G_L(s)$ with $G_{LF}(s)$ assuming D_L as zero matrix.

$$\begin{aligned}
&C_L S_{L,R} x_{LF} = C_L x_L \\
&C_L{}^T C_L S_{L,R} x_{LF} = C_L{}^T C_L x_L \\
&S_{L,R} x_{LF} = x_L \\
&S_{L,R}{}^T S_{L,R} x_{LF} = S_{L,R}{}^T x_L \quad \text{(since } S_{L,R} \text{ may not be square if order is reduced.)} \\
&x_{LF} = \left(S_{L,R}{}^T S_{L,R}\right)^{-1} S_{L,R}{}^T x_L
\end{aligned} \tag{44}$$

$$x_{RF} = \left(S_{R,R}{}^{T} S_{R,R}\right)^{-1} S_{R,R}{}^{T} x_R \quad \text{with similar manner.}$$

Here x_{LF} and x_{RF} represent the state variables for the reduced-model system. The new state variable of x_{LF} or x_{RF} has reduced rank compared to old state variable of x_L or x_R, which means that the new state variable is a projection from the old state variable.

5. Add two systems together

$$G_M(s) = \left[\begin{array}{c|c} A_M & B_M \\ \hline C_M & D_M \end{array}\right] = G_F(s) = \left[\begin{array}{cc|c} A_{LF} & 0 & B_{LF} \\ 0 & A_{RF} & B_{RF} \\ \hline C_{LF} & C_{RF} & D_F \end{array}\right] \tag{45}$$

$$x_M = \begin{bmatrix} x_{LF} \\ x_{RF} \end{bmatrix} = T\,x$$

, where x_M denotes the final state variable for the combined system of stable and antistable systems and T represents transformation matrix between x_M and x.

Here is an example by numerical values when reduced order is three,

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -0.5549 & 5.5612 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -5.0441 & 139.6469 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 3.3291 \\ 0 \\ 30.2648 \end{bmatrix} \tag{46}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$A_L = \begin{bmatrix} \text{-11.9212} & \text{-0.1343} \\ 0 & \text{-0.3538} \end{bmatrix} \quad B_L = \begin{bmatrix} 15.8702 \\ \text{-6.3581} \end{bmatrix}$$
$$C_L = \begin{bmatrix} \text{-0.0034} & 0.9427 \\ \text{-0.0835} & \text{-0.0130} \end{bmatrix} \quad D_L = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{47}$$

$$A_R = \begin{bmatrix} 11.6539 & 0.6924 \\ 1.1155 & 0.0663 \end{bmatrix} \quad B_R = \begin{bmatrix} 2.3268 \\ 2.2326 \end{bmatrix}$$
$$C_R = \begin{bmatrix} \text{-0.2662} & 2.9864 \\ 0.5053 & 0.0300 \end{bmatrix} \quad D_R = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{48}$$

$$V = \begin{bmatrix} \text{-0.0034} & 0.9427 & \text{-0.0198} & 0.3331 \\ 0.0409 & \text{-0.3331} & \text{-0.0907} & 0.9376 \\ \text{-0.0835} & \text{-0.0130} & 0.9919 & 0.0949 \\ 0.9957 & 0.0158 & 0.0869 & \text{-0.0294} \end{bmatrix} \qquad \begin{bmatrix} x_L \\ x_R \end{bmatrix} = V^T x \tag{49}$$

$$HSV_L = \begin{bmatrix} 8.4716 \\ 0.0522 \end{bmatrix} \quad P_L = \begin{bmatrix} 10.6633 & \text{-8.8451} \\ \text{-8.8451} & 57.1294 \end{bmatrix} \quad Q_L = \begin{bmatrix} 0.0003 & \text{-0.0002} \\ \text{-0.0002} & 1.2562 \end{bmatrix}$$
$$HSV_R = \begin{bmatrix} \text{1.6688e+15} \\ 0.1113 \end{bmatrix} \quad P_R = \begin{bmatrix} \text{0.0040e+015} & \text{-0.0668e+015} \\ \text{-0.0668e+015} & \text{1.1242e+015} \end{bmatrix} \tag{50}$$
$$Q_R = \begin{bmatrix} \text{0.0230e+015} & \text{-0.2401e+015} \\ \text{-0.2401e+015} & \text{2.5084e+015} \end{bmatrix}$$

Sorted hankel singular values are as follows,

$$HSV = \begin{bmatrix} \text{1.6688e+15} & 8.4716 & 0.1113 & 0.0522 \end{bmatrix} \tag{51}$$

$$A_{LF} = \text{-0.3541} \qquad B_{LF} = 2.4494$$
$$C_{LF} = \begin{bmatrix} \text{-2.4494} \\ 0.0002 \end{bmatrix} \quad D_{LF} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{52}$$

$$
\begin{aligned}
A_{RF} &= A_R \quad B_{RF} = B_R \\
C_{RF} &= C_R \quad D_{RF} = D_R
\end{aligned}
\tag{53}
$$

, where $S_{L,L} = \begin{bmatrix} 0.0001 \\ -0.3851 \end{bmatrix}$, $S_{L,R} = \begin{bmatrix} 0.4021 \\ -2.5969 \end{bmatrix}$, $S_{R,L} = S_{R,R} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$$
totbnd = 0.1044
$$

$$
A_M = \begin{bmatrix} -0.3541 & 0 & 0 \\ 0 & 11.6539 & 0.6924 \\ 0 & 1.1155 & 0.0663 \end{bmatrix} \quad B_M = \begin{bmatrix} 2.4494 \\ 2.3268 \\ 2.2326 \end{bmatrix}
$$
$$
C_M = \begin{bmatrix} -2.4494 & -0.2662 & 2.9864 \\ 0.0002 & 0.5053 & 0.0300 \end{bmatrix} \quad D_M = \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\tag{54}
$$

$$
x_{M\,ord3} = \begin{bmatrix} 0.0582 & -0.3761 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_L \\ x_R \end{bmatrix} = \begin{bmatrix} -0.3547 & 0.1276 & 3.5937\text{e-}5 & 0.0520 \\ -0.0198 & -0.0907 & 0.9919 & 0.0869 \\ 0.3331 & 0.9376 & 0.0949 & -0.0294 \end{bmatrix} x
\tag{55}
$$

When the reduced order is two, a final state variable can be obtained with the following transformation matrix.

$$
\begin{aligned}
x_{M\,ord2} &= \begin{bmatrix} 0.0582 & -0.3761 & 0 & 0 \\ 0 & 0 & 0.0726 & -1.2213 \end{bmatrix} \begin{bmatrix} x_L \\ x_R \end{bmatrix} \\
&= \begin{bmatrix} -0.3547 & 0.1276 & 0 & 0.0520 \\ -0.4082 & -1.1518 & -0.0440 & 0.0422 \end{bmatrix} x
\end{aligned}
\tag{56}
$$

Appendix K. Source code for LEGO self-balancing robot

```
// LEGO two-wheeled balancing robot
//  k1, k2, k3, k4 feedback gains are specified below

const tSensors GyroSensor        = (tSensors) S1;   //gyro sensor//
```

```
#define GyroScale 4
#define half_h 2    // Increment used in Runge-Kutta integration
#define t_scale 500

task main ()
{
  float a=0.2, aa=0.001;   // aa is the filter time-constant to take care of the
gyro drift
  float theta_bias = 0;

  // This is the feedback loop gain
  float k1 = 0.2;     // position feedback gain
  float k2 = 90;      // velocity feedback gain
  float k3 = 8;      // tilt feedback gain
  float k4 = 10.0;    // angular velocity feedback gain
  long f1=0, f2=0, Gyro_value = 0;
  int pwr=0, batt=0;
  float GyroBias = 593.8;  // This is the bias that I had to apply to my gyro
sensor.  You may need to find your own gyro bias
  float k_pwr;  // This is the gain that is computed adaptively based on the
battery voltage (as the battery is drained, the gain is increased)

  int i=0;

  float theta=0, theta_old=0, t_old=0;
  float x=0, x_old=0, x_dot=0;

  // This causes the motors to stop when they are set to zero
  bFloatDuringInactiveMotorPWM = false;
  theta_old = 0;
  t_old= nSysTime;
  f1=0;
  x_old=0;


  ClearTimer(T1);
  // Find the gyro bias associated w/ the balanced position
  // Hold the robot in the balanced position for 3 sec to find the gyro bias
  GyroBias = 0;
  while (time1[T1] < 3000) {
```

```
        // filter the sensor output
        Gyro_value = SensorValue(GyroSensor);

        wait1Msec(100);
        GyroBias = (1-a)*GyroBias + a*Gyro_value;
  }
  // play a sound when the training is over
  PlaySound(soundBlip);

  //nxtDisplayTextLine(4, "G_B: %d", GyroBias);

  // I ended up hard-coding the bias after measuring it a few times.
  // comment out the following line if you want the Gyro bias to be measured
adaptively

  GyroBias = 593.8;

        // Measure the battery voltage and compensate for it by adjusting the
gain (k_pwr)
  batt=nAvgBatteryLevel;
  k_pwr = 0.7 + (0.7-1.1)/(8816-8196)*(batt-8816);

  nMotorEncoder[motorC] = 0;
  nMotorEncoder[motorA] = 0;

  while(true) {

    i++;
    // Runge-Kutta integration (http://en.wikipedia.org/wiki/Runge-kutta)
    wait1Msec(half_h);
    f2 = (SensorValue(GyroSensor)-GyroBias)/GyroScale; // f(tn)
    wait1Msec(half_h);
    theta = theta_old + (f1+2*f2)*(nSysTime-t_old)/t_scale;
        theta_old = theta;
        x = nMotorEncoder[motorC];
        // compute the linear velocity
        x_dot = x-x_old;
    f1 = (SensorValue(GyroSensor)-GyroBias)/GyroScale; // f(tn+h/2)
    t_old = nSysTime;
        x_old = x;
```

```
        // Compute the long-term average of tilt
    theta_bias = theta_bias*(1-aa) + theta*aa;

    pwr = k1*x + k2*(x_dot) + k3*(theta-theta_bias) + k4*f1;

    motor[motorA] = (k_pwr*pwr );
    motor[motorC] = (k_pwr*pwr );

  }
}
```

Appendix L. Specification of LEGO Mindstorms NXT

1. Processors

- Main processor: Atmel 32-bit ARM processor, AT91SAM7S256, 256 KB FLASH, 64 KB RAM, 48 MHz
- Co-processor: Atmel 8-bit AVR processor, ATmega48, 4 KB FLASH, 512 Byte RAM, 8 MHz

2. Interface

- 4 input ports 6-wire interface supporting digital and analog interface
- 3 output ports 6-wire interface supporting input from motor encoders
- 4 button user-interface Rubber buttons

3. Communication

- Bluetooth wireless communication (CSR BlueCoreTM 4 v2.0 +EDR System)
- USB 2.0 communication Full speed port (12 Mbit/s)

4. Programming Language

- RobotC Software [15]

Appendix M. Partial reconfiguration implementation using PlanAhead

Implementation of partial reconfiguration using PlanAhead software is explained in this appendix.

Software and hardware environments are as follows.

Software conditions:

- ISE 9.2i Service Pack 4 with PR overlay
- PlanAhead 10.1

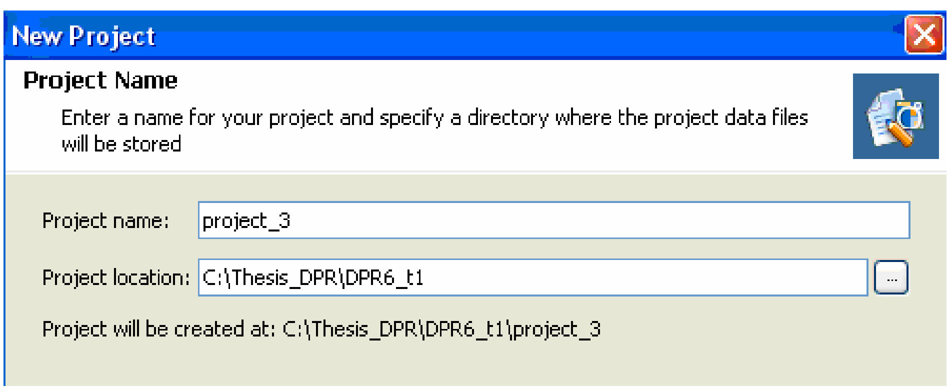
Hardware Environments:

- Digilent FX12 Development Board with Xilinx Virtex-4 FX12 FPGA
- JTAG Programming cable

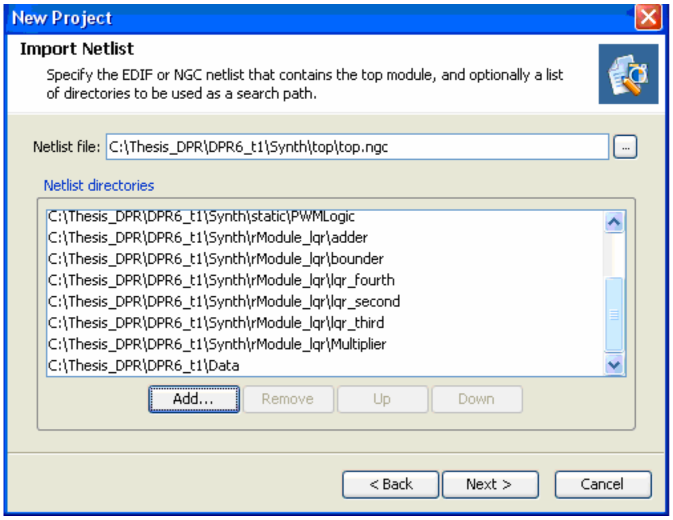
Specific procedure of partial reconfiguration is as follows.

1. Start PlanAhead program

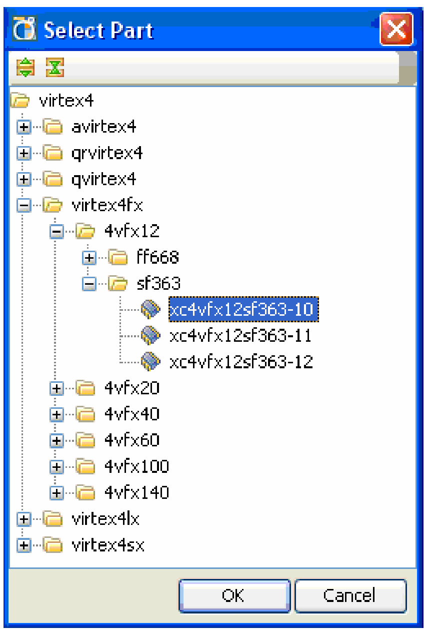
Click on Create A New Project



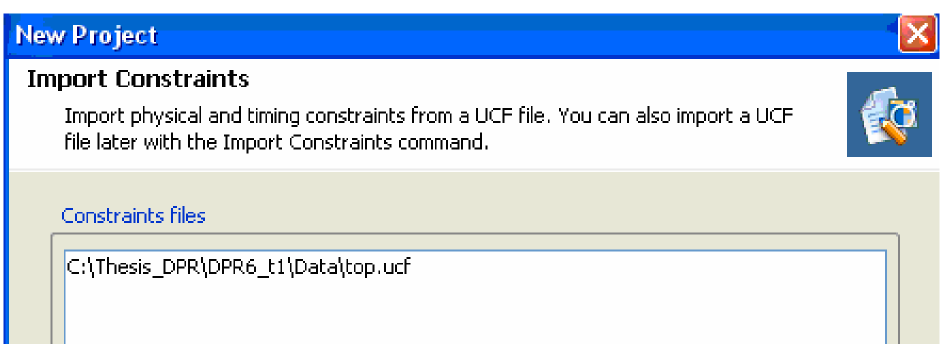
Choose top-level ngc file and add directories of static logic and reconfigurable modules.



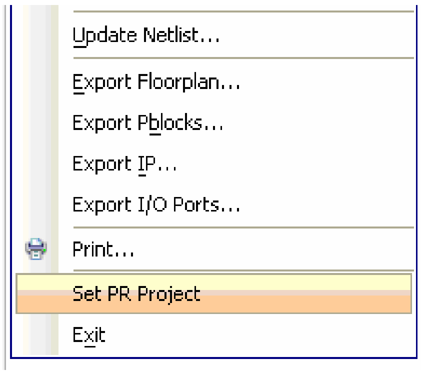
Select specific Virtex-4 FPGA for Digilent FX12 card.



Import ucf constraints from ‘Data’ directory.

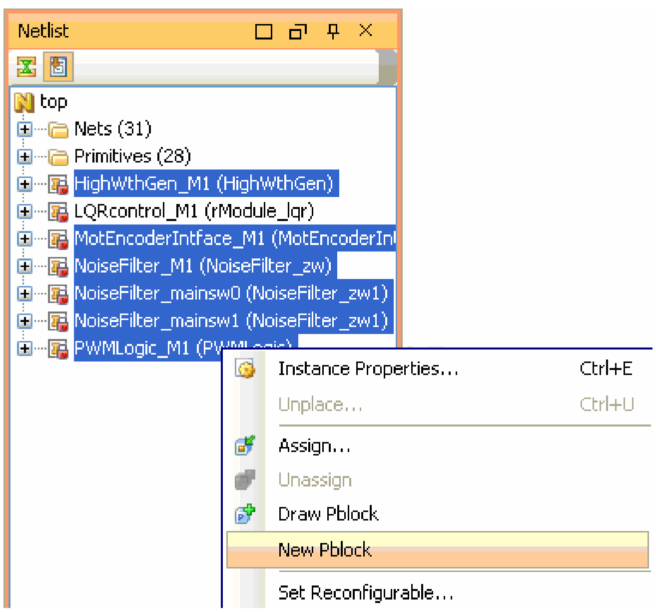


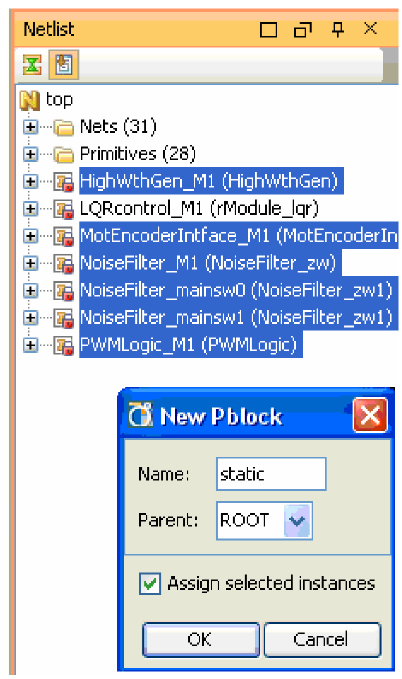
Select File → Set PR Project from the menu to make an active partially reconfigurable project.



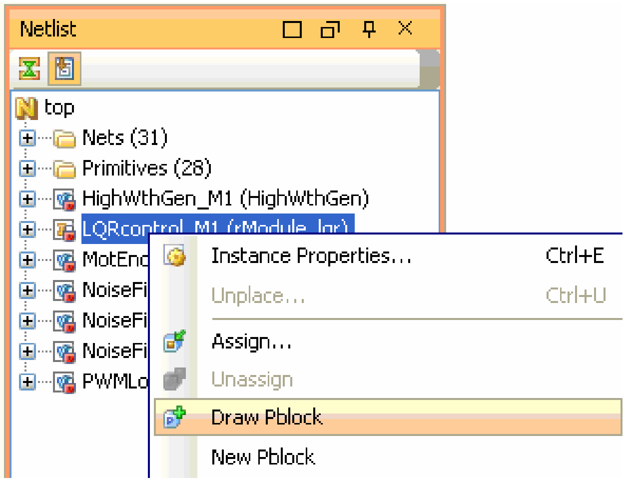
2. Create static and reconfigurable Pblocks

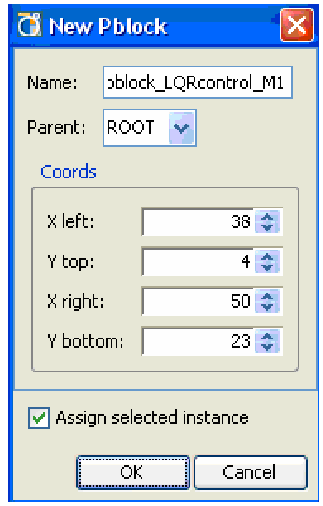
Select all static modules and create static Pblock.





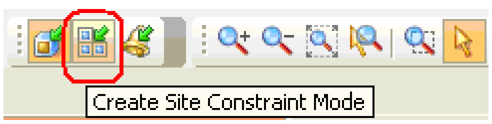
Select reconfigurable module and draw a rectangular Pblock in floorplan window.



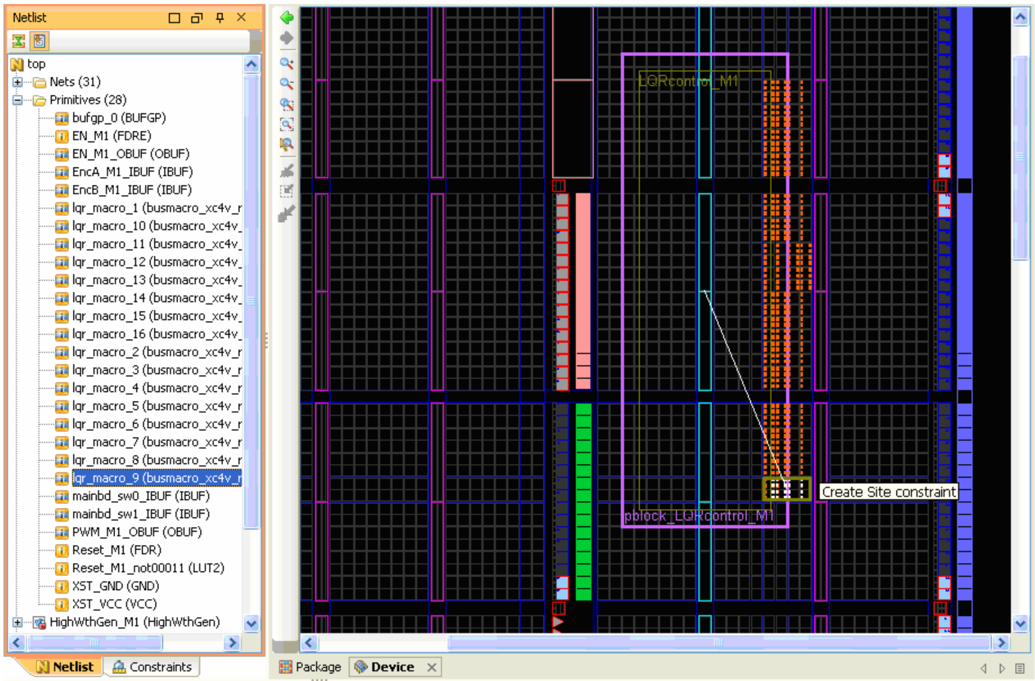


3. Place components

Select Create Site Constraint Mode.

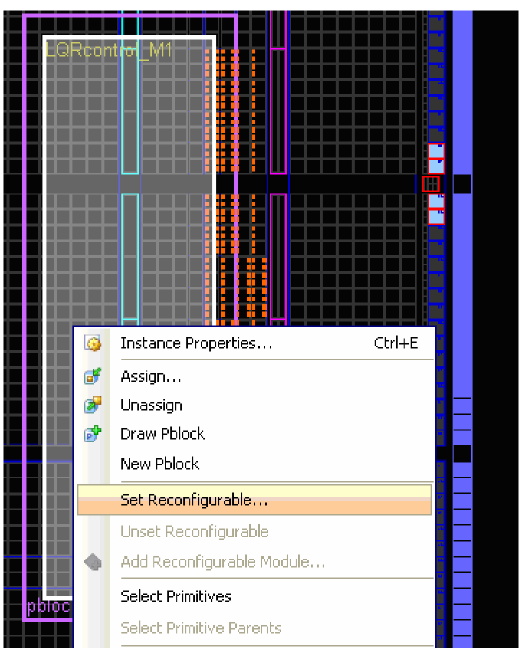


Select bus macro instances from top-level Primitives folder and place them on the right side of reconfigurable Pblock. Then place bufg instance at BUFGCTRL_X0Y0.

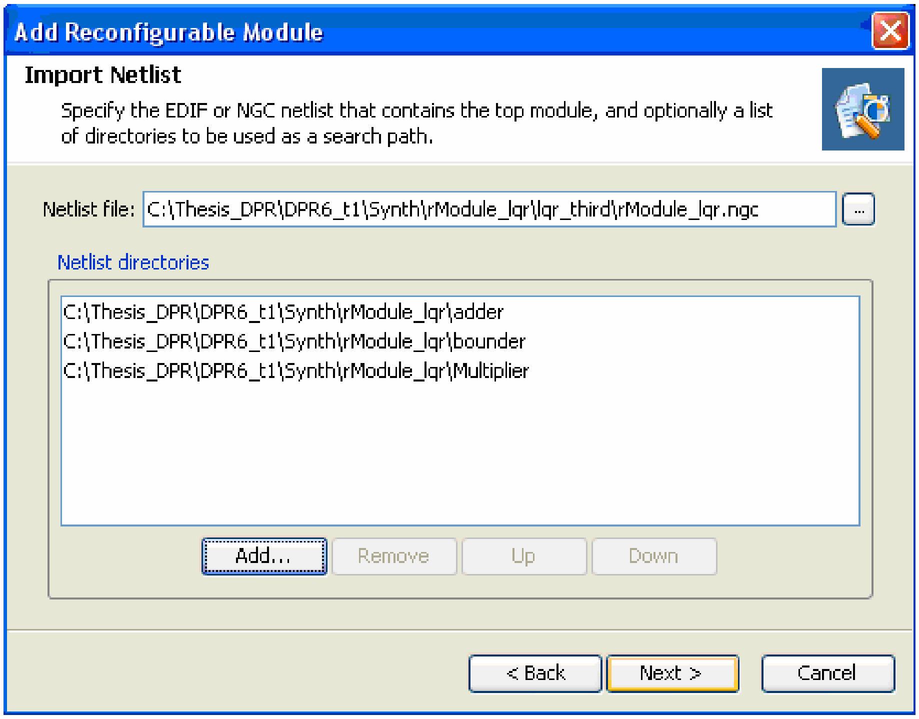


4. Set and add reconfigurable modules

Select reconfigurable Pblock and select ‘Set Reconfigurable’.

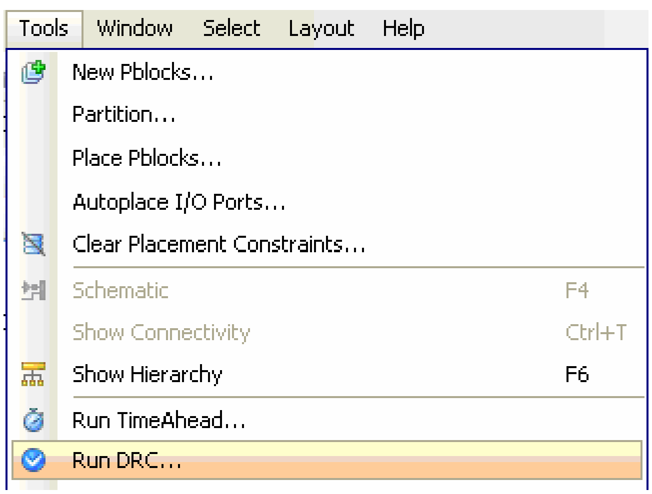


Add reconfigurable modules of lqr_third and lqr_second.



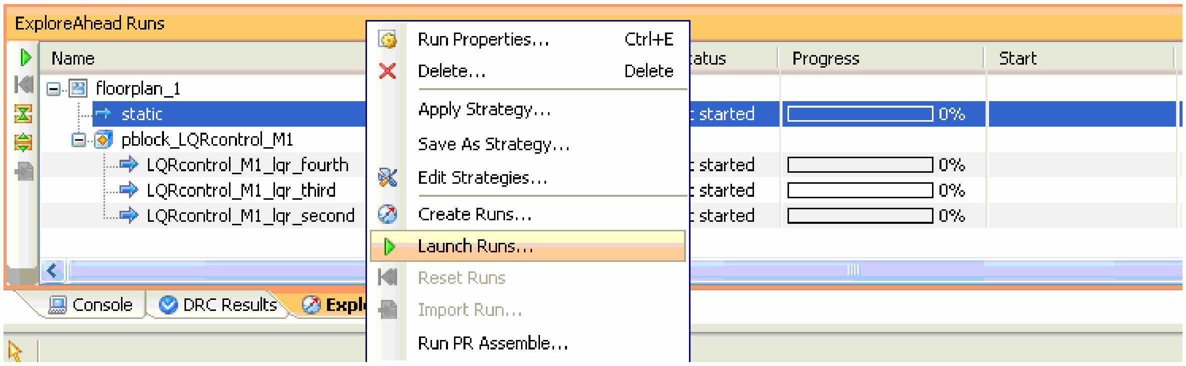
5. DRC check and view results

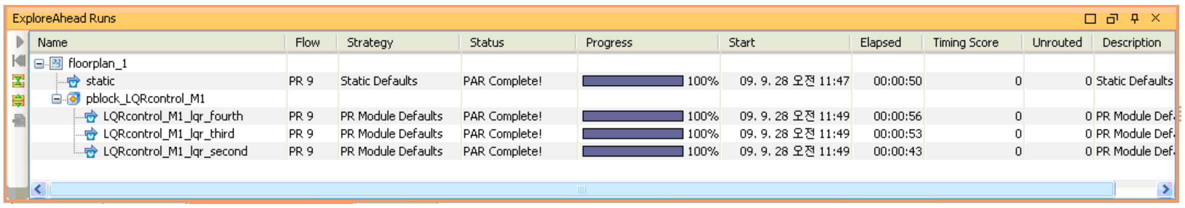
Select Tools → Run DRC in PlanAhead menu.



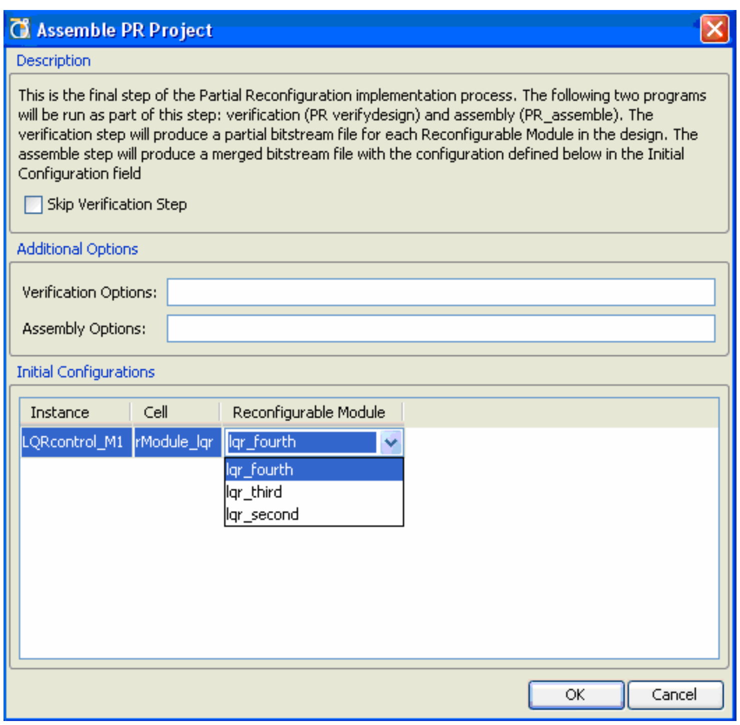
6. Implementation of partial reconfiguration and Assemble the design

Execute Initial Budgeting and static implementation by clicking 'Launch Runs'. Then, PR implementation on each reconfigurable module.





Run PR Assemble by right-clicking one of the Reconfigurable modules. Then, select initial Reconfigurable module which will be included in the static full bitstream.



In Windows Explorer, open the following directory and find generated partial and full bitstreams.