Master's thesis

Faculty of Electrical Engineering
Department of Measurement

FPGA Based Robotic Motion Control System

Bc. Martin Meloun

June 2014
Supervisor: Ing. Pavel Píša Ph.D.

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
Katedra měření

Akademický rok **2012-2013**

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Martin Meloun**

Studijní program: **Kybernetika a robotika**
Obor: **Senzory a přístrojová technika**

Název tématu česky: **Systém pro řízení pohybu robota využívající FPGA**

Název tématu anglicky: **FPGA Based Robotic Motion Control System**

Pokyny pro vypracování:

Seznamte se s již řešenými projekty systémů pro řízení polohy na bázi knihovny PXMC a proveďte jejich rozšíření, integraci a portaci na perspektivní kombinace FPGA a CPU (jak diskrétní, tak implementované na FPGA).

a) Zpracujte přehled použitelných kombinací FPGA a CPU pro danou aplikaci, zaměřte se i na výkonnost a cenovou dostupnost jednotlivých řešení s diskrétními a syntetizovanými CPU.

b) Implementujte ve VHDL rozhraní pro přístup k syntetizovaným periferiím v FPGA Spartan 6 připojeném k mikrokontroléru LPC1788 na desce LX_CPU1 a navrhněte potřebné periferie pro řízení 4 až 6 stejnosměrných motorů. K otestování funkce lze využít mechaniku robota BlueBot.

c) Navrhněte rozšíření periferií pro řízení bezkartáčových motorů.

d) Podle možností navrhněte další možná (budoucí) rozšíření systému, například portaci do prostředí operačního systému reálného času, využití sběrnice CAN nebo komunikace ETHERNET pro řízení robota atd.

Seznam odborné literatury:

[1] Volnei A. Pedroni: Digital Electronics and Design with VHDL, MORGAN KAUFMANN 2008, ISBN: 0123742706

[2] Enoch O. Hwang: Digital Logic and Microprocessor Design with VHDL, Thomson 2006, ISBN: 0-534-46593-5

[3] Skup Konrad R.: Motion Control for Mobile Robots: Basics and Concepts of PXMC Library for Brushless DC Motors, 2008, ISBN: 978-3639086140

[4] Burian V.: Využití programovatelného pole pro řízení bezkartáčových motorů, bakalářská práce ČVUT FEL 2011

[5] LPC17xx User manual, Rev. 2, NXP Semiconductors, 19 August 2010

[6] Knihovna PXMC, http://pxmc.org/, GIT http://www.pikron.com/pxmc/git/pxmc.git, 2001 – 2012, PiKRON s.r.o.

Vedoucí diplomové práce: Ing. Pavel Píša, Ph.D. (K13135)
Datum zadání diplomové práce: 14. prosince 2012
Platnost zadání do[1]: 30. června 2014

L.S.

Prof. Ing. Vladimír Haasz, CSc.
vedoucí katedry

Prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 14.12.2012

[1] Platnost zadání je omezena na dobu tří následujících semestrů.

Acknowledgement

I would like to express my gratitude to Ing. Pavel Píša Ph.D. for supervision, useful comments, remarks and support for this master thesis. Furthermore I would like to thank Ing. Tomáš Pajdla Ph.D. for permission to publish my research report supervised by him as an appendix of this thesis.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with methodical instructions about ethical principles for writing academic thesis.

Abstrakt

Diplomová práce se zabývá zpracováním přehledu použitelných kombinací CPU a FPGA pro centralizované řízení až 4 PMSM motorů a dále návrhem samotných periferií v FPGA pro samotné řízení. Prvně je zpracován přehled možných kombinací CPU a FPGA, i s možností použití jenom FPGA a v něm syntetizovaný procesor, a to vzhledem k výkonu a ceně. Dále je popsáno oživení FPGA na zvolené platformě a komunikace mezi CPU a FPGA. Poté pokračuje odvozením periferií pro řízení PMSM motorů. Samotné motory pak řídí výkonový modul, který je propojen s řídícím modulem přes vlastní typ sběrnice, pro kterou byla navržena odpovídající periferie. Diplomová práce využívá předchozích projektů systémů pro řízení motorů, konkrétně PXMC knihovny a sysless frameworku jako podklad pro embedded aplikaci na CPU, oboje od firmy PiKRON . Cílem práce je připravit platformu k produktizaci.

Klíčová slova

CPU, FPGA, LPC1788, Spartan6, LX_CPU1, PMSM motor, syntetizovatelný procesor, robot, řízení polohy, Tumbl, PXMC, sysless

Abstract

This master thesis presents a platform research about possible CPU and FPGA combinations for centralized control system for up to 4 PMSM motors, alongside with design of the FPGA peripherals needed for the actual controller. At the beginning a platform overview is presented, including the option to use a synthesized controller inside FPGA, with performance and cost of each platform in mind. Then there is a description on how to configure FPGA on the chosen platform and how to communicate between CPU and FPGA. Then peripherals for PMSM motor controller are derived. The motors themselves are controlled by power stage module, which is connected with the control module using a custom bus, for which a relevant peripheral was designed. This thesis bases on previous PMSM motor control systems, namely PXMC library and sysless framework as a base for an embedded application on CPU, both developed by PiKRON. The goal of this thesis is to ready the platform for productization.

Keywords

CPU, FPGA, LPC1788, Spartan6, LX_CPU1, PMSM motor, synthesized controller, robot, motion control, Tumbl, PXMC, sysless

Contents

1. Introduction **1**
1.1. PMSM Motor Controller . . . 2
1.2. Planning . . . 2

2. Platform **4**
2.1. Choosing A Platform . . . 4
2.2. Sysless Framework . . . 6
2.3. Synthesized CPU Cores . . . 7
2.4. The Platform . . . 8
2.5. Spartan-6 Wiring And Configuration . . . 11
2.6. Slave Memory Controller . . . 12
2.7. Power Stage Module . . . 14

3. FPGA Peripherals Analysis **15**
3.1. Parallelism . . . 15
3.2. Selecting A Synthesized Processor . . . 18
3.3. IRC Peripheral . . . 18
3.4. Power Stage Module Communication . . . 20

4. Tumbl Co-processor Core **21**
4.1. Overview . . . 21
4.2. Minimalization Of The Core . . . 22
4.3. Co-processor Modifications . . . 23
4.4. Core Modifications . . . 24
4.4.1. Conditional Execution . . . 25
4.4.2. Branching With Link . . . 26
4.4.3. Count Leading Zeroes . . . 28
4.4.4. Instruction Binary Encoding Changes . . . 28
4.4.5. Assembler Changes . . . 28
4.5. Processor Pipeline . . . 29
4.5.1. Instruction Fetch . . . 29
4.5.2. Instruction Decode . . . 30
4.5.3. Execution . . . 30
4.5.4. Memory And Writeback . . . 31
4.5.5. General Purpose Register File . . . 31
4.5.6. Core Component . . . 32
4.5.7. Top Module . . . 32
4.6. Division . . . 32
4.7. External Memory Interface . . . 32
4.8. C lanugage support . . . 33
4.9. Pipeline Balance . . . 33

5. Tumbl Co-processor Implementation **34**
5.1. Enumerations . . . 34
5.2. Records . . . 37
5.3. Entities . . . 42
5.3.1. Instruction Fetch . . . 42
5.3.2. Instruction Decode . . . 43

5.3.3. Execution . . . 43
5.3.4. Memory And Writeback . . . 44
5.3.5. Core Component . . . 44
5.4. Platform Entities . . . 45
5.4.1. General Purpose Registers File . . . 46
5.5. Top Module . . . 46

6. Other FPGA Peripherals 48
6.1. IRC Co-processor . . . 48
6.1.1. Operation . . . 48
6.2. LX Master . . . 49
6.2.1. Operation . . . 50

7. FPGA Simulation 52
7.1. ModelSIM Setup . . . 52
7.2. Tumbl Simulations . . . 52
7.2.1. Processor Core . . . 54
7.2.2. Cycle Counting . . . 54
7.2.3. Tumbl External Memory Interface Collisions . . . 57
7.3. IRC Co-processor . . . 57
7.4. LX Master Transmission . . . 57
7.5. Master CPU Online Debugging . . . 59

8. Conclusion 60

Appendices

A. Inverse Kinematics For A General 6R Manipulator 62
A.1. Introduction . . . 65
A.2. Inverse Kinematics Task . . . 66
A.2.1. Formulation . . . 66
A.2.2. Raghaven and Roth Solution . . . 66
A.2.3. Modified Manocha and Canny Optimization . . . 69
Elimination of c_1, s_1, c_2, s_2 . . . 69
Sylvester Dialytic Elimination Method . . . 70
Solving x_4 and c_5, s_5 . . . 72
A.2.4. Solving remaining variables . . . 73
A.3. Implementation . . . 73
A.3.1. Symbolic Preprocessing . . . 73
A.3.2. Numerical Substitution . . . 74
A.3.3. Verification . . . 77
A.3.4. Conclusion . . . 79
A.4. Documentation . . . 80
A.4.1. Platform . . . 80
A.4.2. Numerical Optimization . . . 80
A.4.3. Doxygen . . . 81

B. Tumbl Technical Reference Manual 82
B.1. General Purpose Registers . . . 82
B.2. Condition Evaluation . . . 82
B.3. Special Registers . . . 83

B.4. Interrupts 83
B.5. Instruction Set 83

C. Tumbl C lanugage support **88**
C.1. binutils 88
C.2. gcc 88
C.3. newlib 89

D. Memory Map **90**
D.1. Master CPU Memory Map 90
D.2. Tumbl Memory Map 91

Bibliography **93**

Abbreviations

MCU	Micro-processor unit
CPU	Central processor unit
FPGA	Field-programmable gate array
PMSM	Permanent Magnet Synchronous Motors, formerly called BLDC (brush-less DC motors)
BRAM	Xilinx FPGA primitive, dual-port block RAM
DSP48	Xilinx FPGA primitive, DSP48 slice consists of a multiplier followed by an adder,

Chapter 1
Introduction

Electrical motors are widely used in many machines, starting from trivial tasks in toys to task demanding high accuracy level, such as space vehicles or factory robotic manipulators. PMSM motors, formerly called BLDC motors, are a popular type of electrical motors for their performance and durability but bearing a disadvantage of requiring a sophisticated controller implemented in a microprocessor. The microprocessor main task is to implement a control loop to substitute a brush commutator in a standard DC motor. This task was implemented in **PXMC** library [9] for a voltage driven controller [8], distributed under GPL license. Many applications use more than one motor. However today's microprocessors don't have the needed peripherals to control more than one motor which needs one IRC input, 3 PWM outputs and a A/D converter (to be multiplexed for all axes) per motor. One way to control more than one PMSM motor is to use a distributed system with multiple microprocessors and one central processor unit. In this case you have to face synchronization challenges and increased cost of the components. The other option is to use FPGA to implement peripherals for more PMSM motors in a centralized system, lifting off all synchronization issues and being more budget friendly. This effort was started in [7]. This thesis first aims to find out the best platform for a control board of such centralized control system supporting up to 4 PMSM motors, with later possible extension up to 8 PMSM motors. The control board is connected to a power stage module, designed by PiKRON, through FPGA using a custom bus. The platform thus consists of a CPU, handling common tasks such as communication through USB, Ethernet etc. and possibly running an operating system, and FPGA to extend its peripherals to accommodate the need for multiple IRC inputs and a peripheral for the custom bus to the power stage module. The PWMs are sent through the custom bus to the power stage module which in return sends current or voltage readings.

Figure 1.1. Platform overview

A PMSM controller for 4 motors however turns out to be too demanding task for the CPU. Such controller has a slow control loop, which sets up the speed reference, and a fast control loop, which substitutes a brush commutator, for example by using block commutation. The slow control loop isn't a demanding task and may stay in the CPU. The demanding task is the fast control loop and has to be offloaded to FPGA. This is the main goal of this thesis. To design FPGA peripherals for a PMSM controller, alongside peripherals for 4 IRC inputs with easy later extension to 8 IRC inputs and a communication peripheral. The

PMSM controller is described in section 1.1. At the end we intend to have a platform with necessary FPGA peripherals designed and tested (either on the board or in simulation) and making the platform ready for productization.

1.1 PMSM Motor Controller

Aside from voltage driven controller from **PXMC** library, a more advanced current driven controller can be used with an advantage of being able to control based on momentum reference. The fast control loop of a current driven controller is shown on figure 1.2.

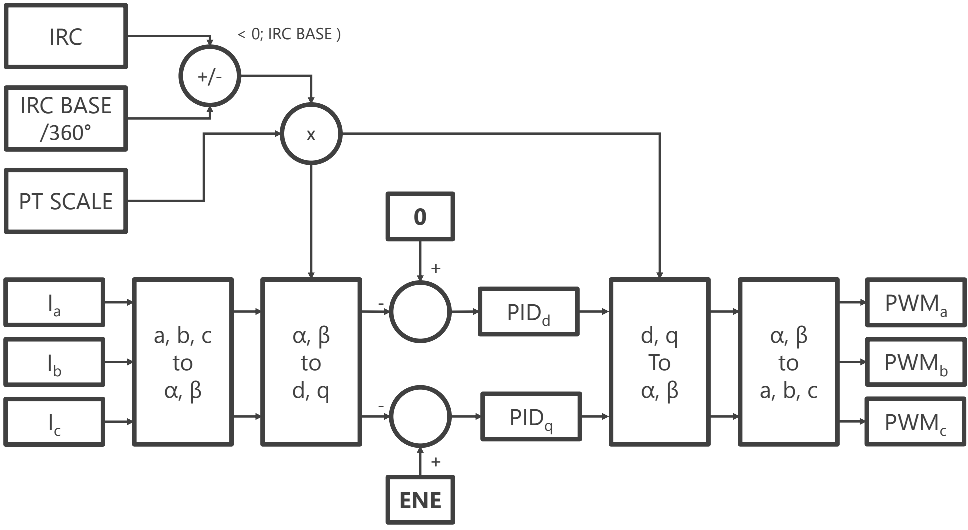

Figure 1.2. Fast control loop of a current driven PMSM motor controller

The concept is based on control theory used in [11]. An overview of the controller: first, it converts from 3-phase currents I_a, I_b, I_c to vector coordinates of stationary reference frame α, β using Park transformation. These are further converted to vector coordinates of rotating reference frame d, q using Clarke transformation, resulting in I_d (direct-axis current) and I_q (quadrature-axis current), orthogonal to each other. The property of I_d and I_q is that they are DC variables, not changing in time based on the current phase of the rotor. These two currents are then regulated by PID controllers based on the reference of the momentum, resulting in two voltages U_d and U_q, which are then converted back to 3-phase system voltages U_a, U_b, U_c using inverse transformations. They are sent to power stage module as PWMs.

1.2 Planning

Appendix A presents my research report for general 6R (6 rotating joints) robotic manipulator inverse kinematics task, which is still a non-trivial task today. The inverse kinematics task is solved algebraically as a set of polynomial equations with focus on numerical accuracy. It can be used with the product to extend support of trajectory planning for 6R

manipulators of general geometry, which is still a very uncommon feature.

Chapter 2
Platform

This chapter describes analysis leading to selection of core components for a developed motion control platform. Chapter starts with analysis of available components and concepts which is followed by their actual use and interconnection used on circuit board.

2.1 Choosing A Platform

The motion control application requires processing power and more motor interfacing specific peripherals: incremental encoder (IRC) signals processing, multi-phase power stages PWM and current control etc.. Then processing power is required for digital signal processing (DSP). The control loops for winding current, speed and positional control as well ass trajectory planing demands different minimal sampling frequencies for smooth operation. There exist many DSP based systems on chip for single or two axis control but chips for integrated systems for complex multiple axes controllers are seldom. Such system can be build by combining general purpose CPU and discrete IRC and PWM peripheral chips. But such design is inflexible and does not allow to integrate application specific functions (i.e. exact position triggered outputs control, etc.). With that in mind, the other option is to combine CPU and FPGA. We start with looking on possible combinations based on their performance and cost:

- **Synthesized CPU in FPGA chip:** Further develop ideas from [7] and use a synthesized CPU. The advantages of this solution is saving board space by having to use only one chip, however the smallest Spartan-6 chips (non-BGA package) do not have an SDRAM memory controller. This means we have to use a larger, more expensive, Spartan-6 chip in a BGA package which in contrary raises the on the board design (generally more layers than boards with no BGA packaged chip on them) and also soldering side. Synthesized CPU generally offers less performance than a dedicated CPU chip. All additional controllers for peripherals such as USB or ethernet etc., need to be implemented in FPGA. A PHY for such peripherals is usually needed with a dedicated CPU chip as well.

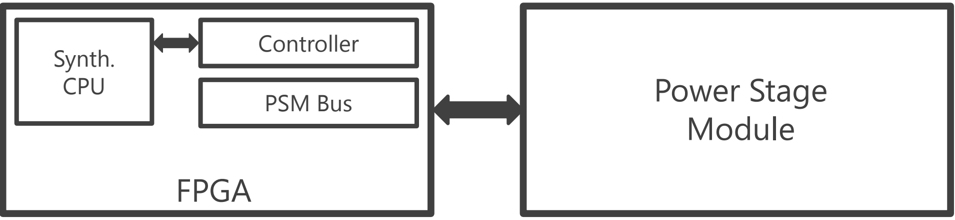

Figure 2.1. Synthesized CPU in FPGA chip

- **Dedicated CPU chip interconnected with FPGA:** Use a dedicated master CPU to handle the common tasks (USB, Ethernet, other communication peripherals) and interconnect it with the FPGA through its external memory interface. An example master CPU would be ARM Cortex-M3 preferably in a non-BGA package for budget friendliness. The `sysless` framework, see section 2.2 for more information, already supports **LPC17xx** ARM Cortex-M3 processors from previous projects, thus making it a good choice from both software support and budget perspective.

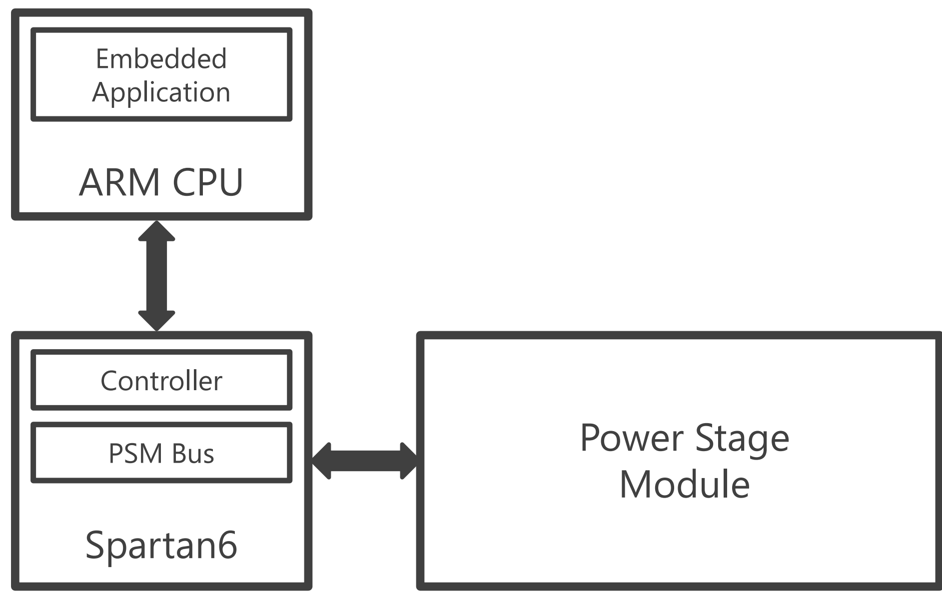

Figure 2.2. Dedicated CPU chip interconnected with FPGA:

- **Xilinx Zynq:** Xilinx Zynq is a dual-core ARM Cortex-A9 processor with FPGA in a single chip with the smallest ones being for a reasonable price. The second core could be used for the fast control loop (for Zynq PMSM motor controller would cease being a heavy duty task, since it operates on 667 MHz (the smallest one), more than 10 times higher than more budget friendly solution). There would be additional costs due to BGA packaging on board and soldering side. This would seem a good choice for a more high-end platform which it would be relatively budget friendly for. Still expected price for the board and components would still be still three to five times higher than with ARM Cortex-M3 with Spartan-6 in non-BGA package.

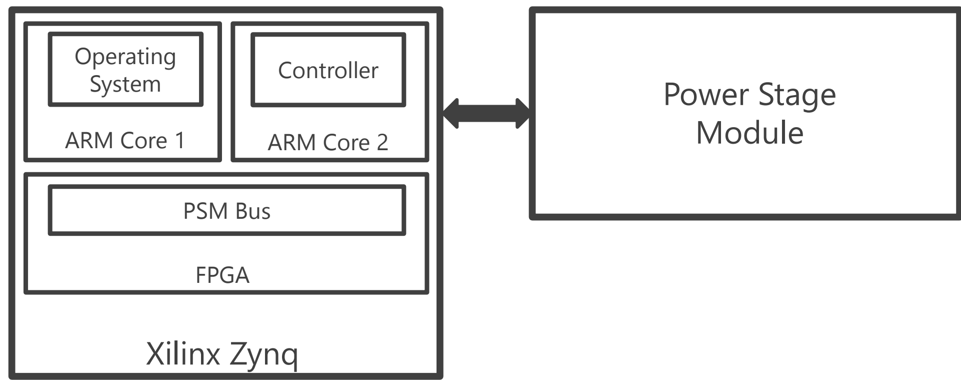

Figure 2.3. Xilinx Zynq

Given our low-cost intention, we chose the second option, concretely ARM Cortex-M3 core **LPC1788** and Spartan-6 FPGA, **XC6SLX9-2TQG144**, which is the largest Spartan-6 in non-BGA package, providing us with sufficient amount of logic cells. Both can be upgraded to ARM Cortex-M4F core **LPC4088** and a Spartan-6 with a better speed grade **XC6SLX9-3TQG144**. As for the other two options, using synthesized core in a single FPGA chip has been abandoned in favor of Xilinx Zynq on both performance and price end. Xilinx XST from Xilinx ISE will be used for synthesis. Table 2.1 shows reference pricing of several components:

Component	**Reference Price[17]**
XC6SLX9-2TQG144 Spartan-6 FPGA (small, non-BGA, speed grade 2)	$15.69
XC6SLX9-3TQG144 Spartan-6 FPGA (small, non-BGA, speed grade 3)	$17.22
XC6SLX75-2CSG484C Spartan-6 FPGA (medium, BGA, speed grade 2)	$94.56
LPC1788FBD208 ARM Cortex-M3 CPU chip	$13.09
LPC4088FBD208 ARM Cortex-M4F CPU chip	$13.53
XC7Z010-1CLG225C Xilinx Zynq (2x ARM Cortex-A9 CPU, medium size FPGA)	$54.86

Table 2.1. Pricing of relevant components

The first development kit for the platform contains **LPC1788** and **XC6SLX9-2TQG144** chips and was named **LX_CPU1**.

2.2 Sysless Framework

sysless framework [18], or System-less framework, is PiKRON's and its partners solution to have common infrastructure for building bare metal applications for multiple MCU architectures. It uses **ulboot** as a bootloader, able to boot from flash or USB, making development more comfortable, and written in order to be easily reflashable by toggling a hardware

jumper. This framework also contains a set of tools for the host machine to upload images of the software and generally to communicate through USB or serial link. For the chosen platform, **sysless** framework received a few patches:

- **Platform Support:** Support for **LPC1788** MCU, which contains some changes against other processors in its family, and board definition files for **LX_CPU1**.
- **Toolchain Support:** Support for **arm-eabi** toolchain instead of obsolete **arm-elf**.
- **Improved USB:** Support for using USB to send commands to PXMC library command processor and to send custom calls, used for Spartan-6 FPGA configuration and communication.

Sysless framework works as a state machine, periodically polling each of its component (peripheral handlers, such as USB, and virtual components, such as PXMC controller).

2.3 Synthesized CPU Cores

Part of the platform study were synthesized CPU cores. Generally it doesn't pay off to use synthesized CPU cores in an FPGA chip due to lower performance and price per gate than their ASIC counterparts. They are used in cases requiring an FPGA for other reason and offer sufficient performance or as co-processors. In [7] two cores were studied:

- **OpenMSP[14]:** This is a minimal MSP430 Texas Instruments processor core. It is well documented 16-bit Harward architecture CPU with hardware multiplier.
- **Plasma MIPS[15]:** A more robust 32-bit core, yet it's declared to run only on 25 MHz, which is a red flag. It's Von Neumann architecture, supports hardware multiplier and barrel shifts. It additionally supports restricting access to memory space and caches with external memory support.

Both of the above cores have performance issues. OpenMSP is 16-bit, thus slowing down operations with numbers above 16-bit such as division. Plasma MIPS declares only 25 MHz as maximum frequency therefore being too slow. Thus additionally to that, another other two synthesizable cores were studied:

- **MicroBlaze[10]:** Closed source 32-bit CPU core provided by Xilinx and optimized for their FPGA with MMU support, caching support and extensible through FSL for multi-cycle cores. It is supported by many operating systems (both normal and realtime). Supports hardware division, taking up to 32 cycles. However it's requires license fee for Xilinx EDK and been designed and optimized for larger cores, while minimal **MicroBlaze** core would take roughly similar amount of LUTs as the other cores, generally configured for usage up to 3 times more LUT usage.
- **MB-Lite+[13]:** An open-source implementation of **MicroBlaze** processor, supporting basic set of instructions including barrel shifter and hardware multiplier, able to run about at 60 MHz on Spartan-6 (when synthesized without any additional components). It has a 4 stage pipeline and was created in mind with primitives found on Xilinx FPGA and consumes similar amout of LUTs like OpenMSP. It supports JTAG debugging, FSL bus and Wishbone.

- **SecretBlaze:[16]** A more robust **MicroBlaze** processor implementation, supporting caching, external memories etc; making it more suitable to run a real-time operating system. Then it supports multi-cycle instructions, used for multiplying for a shorter critical path and allowing to run on higher frequency.

Out of these, **MB-Lite+** seems as a good start point for a minimal CPU core while **Secret-Blaze** would be a good start point for running a more robust core on a smaller Spartan-6. Full **MicroBlaze** becomes an option if we're using larger Spartan-6 in a BGA package. We would want to use hardware multiplier and barrel shift, with having some support for division (doesn't necessarily have to be hardware) and keep it about 1500 LUTs for a minimal CPU core and 2500 LUTs for a more robust core (for a reference, **XC6SLX9-2TQG144** Spartan-6 has 5720 LUTs).

2.4 The Platform

We assumed **LX_CPU*n*** naming scheme for our chosen platform, with *n* being the revision. This section describes the connected peripherals on it. **LX_CPU1** is the initial prototype followed by **LX_CPU2**[22], which is just a revision of the prototype. Both platforms consists of:

- **LPC1788** as ARM Cortex-M3 master central processor unit containing small internal flash and SRAM memories
- **XC6SLX9-2TQG144** as Spartan-6 FPGA, connected through external memory controller for both SelectMap configuration and normal opertaion
- 32 MB SDRAM connected to master CPU
- 4 MB NOR flash memory connected to master CPU
- 4 IRC differential signals receivers for 5V I/O on signals side, connected to Spartan-6 FPGA
- General purpose 3.3 V CMOS I/Os for both ARM master CPU and Spartan-6 FPGA
- Ethernet PHY connected to master CPU
- USB PHY for a slave and host controller connected to master CPU
- One dedicated I2C port on master CPU
- One dedicated LCD display port connected to master CPU
- One dedicated UART port on master CPU
- One dedicated RS-232 port with RS-485 and uLan extension connected to CPU
- One dedicated CAN port on master CPU
- One dedicated SPI port on master CPU

- One configurable port for TTL I/O, or SPI, or UART on TTL levels on master CPU
- 50 MHz clock source, another clock source configurable by master CPU

The board can be powered from USB or dedicated power supply, depending on a jumper configuration. **LX_CPU2** revision fixes wiring for ethernet PHY as a major fix and adds a missing pull-up resistor for Spartan-6 FPGA, and changes the position of SD and USB host connectors. Photo of the board can be seen on 2.4 with board layout on 2.5.

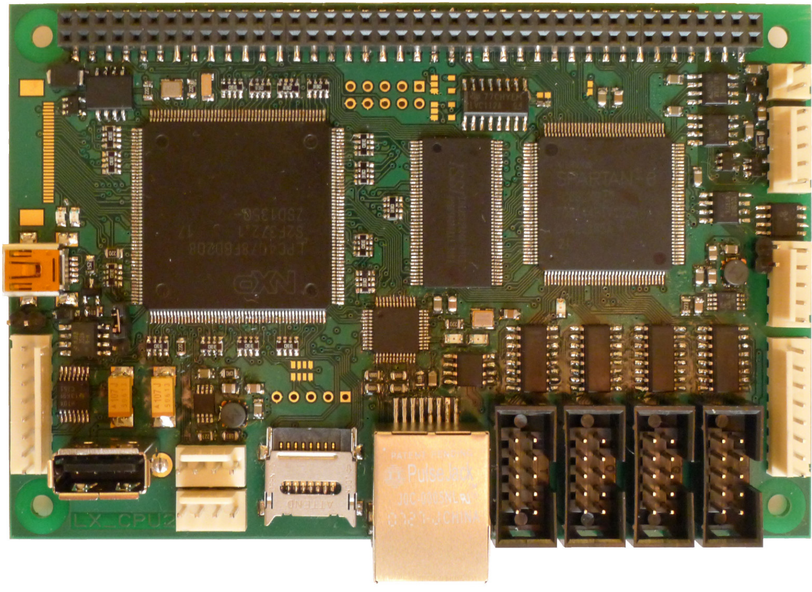

Figure 2.4. Top view of the **LX_CPU2** board

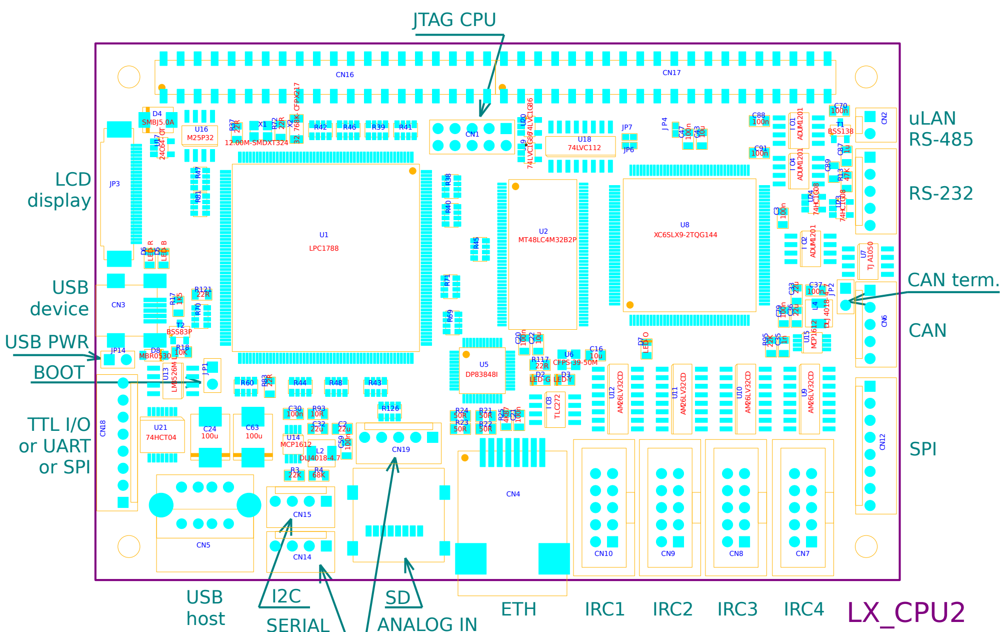

Figure 2.5. Layout of the **LX_CPU2** board

2.5 Spartan-6 Wiring And Configuration

Spartan-6 is wired on master CPU external memory controller for chip selects 0 and 1, pins **CS_XC0** and **CS_XC1**, and using **BLS[3:0]** pins for writing (4 pins for active byte selection), **RD** pin for reading, **DATA[31:0]** pins for writing a 32-bit word and **ADDRESS[15:0]** pins for 16-bit addressing per 32-bit word, giving us 18-bit address space. Additional pins wired from master CPU to Spartan-6 are a clock output (configured to the same clock as is the CPU clock), a pin used to reset Spartan-6 and initiate configuration, another pin for toggling low power state (toggled also on configuration failure) and a pin to assert data transmission when configuring Spartan-6. A pin is wired from Spartan-6 back to the master CPU as a response during configuartion steps and another one to signal successful end of the configuration.

Master CPU communicates with Spartan-6 using memory transactions. All signals are low-level active. At first, chip select is asserted. After a small defined delay, either is address is asserted when reading or address and data are asserted when writing. Finally either **RD** is asserted when reading, or **BLS[3:0]** is asserted when writing. The signal are then kept asserted for a configurable delay. The delays between assertion of chip select and asserting address or data, or any other step. See [19] for details about read and write cycles.

On power up, master CPU initiates configuration and utilizes the writing cycle to send the configuration bitstream. This is achieved by common data signals interconnection for FPGA configuration phase and their reuse for operational system mode for peripherals / slave access data bus. Spartan-6 is configured using SelectMap interface [20] with master CPU as the master entity. This is either 8-bit or 16-bit line. Data are sent when **RDWR_B** pin is asserted and received by Spartan-6 on positive edge of **CCLK** signal. This signal is wired to **BLS[0]**, which is asserted in both 8-bit and 16-bit write. The configuration line is only active when **CSI_B** is asserted. It is wired to **CS_XC0**, in order prevent interference from transactions with other peripherals using the external memory controller, such as reading the configuration bitstream from SDRAM memory. An overview of steps used for FPGA configuration follows. An overview of the actual configuration of the FPGA:

- Setup external memory controller on master CPU: given small amount of data to transfer, use largest possible delays for the transaction, make sure any buffering is off and that the memory is strongly ordered (i.e. the transactions on the bus come in the same order as are in the processor code)
- Setup directions of the general purpose I/O pins: On master CPU side **XC_INIT_PIN**, **XC_DONE_PIN** are inputs and **XC_SUSPEND_PIN**, **XC_PROGRAM_PIN**, **XC_RDWR_PIN** are outputs.
- Assert **XC_PROGRAM_PIN** to initiate programming and **XC_SUSPEND_PIN** to make sure it's not suspended.
- Hold **XC_PROGRAM_PIN** for at least 500 ns, then release it and wait for Spartan-6 to be ready for programming state/condition, signalled by asserting **XC_INIT_PIN**. In case of timeout, release **XC_SUSPEND_PIN** and exit with an error.
- We're now ready to begin data transmission. Assert **XC_RDWR_PIN** and write a

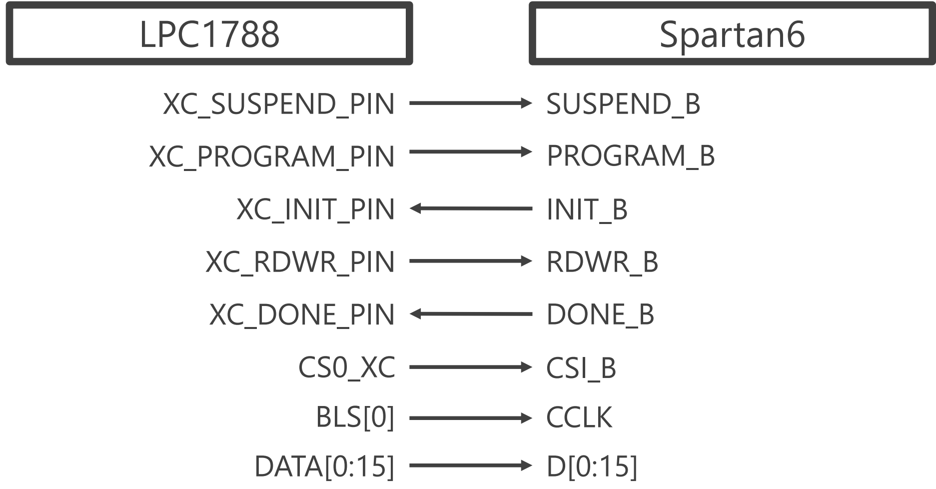

Figure 2.6. Configuration signal mapping

16-bit word on any address mapped to the external memory controller for any chip select with correct configuration done in the first step. The data to write is the generated bitstream from Xilinx tools. It will contain transfer mode, device ID check and CRC. If it was generated for 8-bit transfer, then write it as 8-bit word.

- Periodically through the data transmission (once every 128 writing cycles) it's a good idea to check if **XC_INIT_PIN** has been released by Spartan-6, flagging an error.
- Release **XC_RDWR_PIN** when you sent all data and wait for Spartan-6 to assert **XC_DONE_PIN**. In case of timeout, release **XC_SUSPEND_PIN** and exit with an error. In case of success, make at least 8 writing transactions again (could be any data) to issue start-up clocks.
- At this stage Spartan-6 will release most of the configuration pins for general purpose use (except **XC_SUSPEND_PIN**, **XC_PROGRAM_PIN** and **XC_DONE_PIN**). It's a good idea to issue reset signal and wire it to **XC_INIT_PIN**. Make sure the FPGA design in Spartan-6 is using **XC_INIT_PIN** pin as it's reset input and assert it for several cycles.
- Finally reconfigure external memory controller to its normal running state (such as delays, possibly not needing strong ordering).

2.6 Slave Memory Controller

In order to transfer data between Spartan-6 and master CPU we need to create a slave memory controller. The subsystems interconnection is shown on figure 2.7. On the left-hand side of the slave controller, there are **ADDRESS[15:0]**, **BLS[3:0]** and **RD** as inputs and **DATA[31:0]** as tri-state input / output and connected to master CPU on the other side. On the right-hand side, there are slave peripherals in FPGA, each with **I_CE**, sig-

nal for chip enabling, **I_BLS[3:0]** asserted alongside **I_CE** if we are writing, optionally **I_ADDRESS[*n*:0]** if the peripheral has its own address space and **I_DATA_I[31:0]** as input data. There are no tri-state signals inside FPGA and thus chip enabling as well as data output of the peripherals is multiplexed. The multiplexer decodes It provides **I_ADDRESS[15:0]** and asserts **I_CE** for the corresponding module. In a read cycle it will then transfer output data from the enabled peripheral back to **I_DATA_O[31:0]**. All internal Spartan-6 data transactions are limited to a single cycle duration for performance and inability to set master CPU external memory controller for variant delays based on the address of the transaction. Spartan-6 runs the slave memory controller on 50 MHz while master CPU runs on 72 MHz asynchronously. Due to that all driving signals **BLS[3:0]**, **RD** and **ADDRESS[15:0]** have to be filtered, to prevent incorrect sampling but requiring master CPU to use a longer delay before releasing the signals.

We also need to know what is the minimum sufficient delay before master CPU can release the signals to ensure that the transactions are completed properly. This is started by measuring the delay for reading. It is implemented using a pair of read-only registers holding constants, where bitwise and of those constants is zero. The measurement is a series of readings of the read-only registers, cycling between them. Thus every bit **DATA[31:0]** has to flip in all transactions. We start with the minimum delay and keep increasing it until the series of transactions is completed successfully. If the maximum possible delay is exceeded, we assume that either there is a problem with hardware or with the slave controller in Spartan-6. Then the delay for writing is measured with a similar process using two read-write registers. Two values (again bitwise and of those is zero) are chosen. The measurement is a series of writings, where one constant is written to the first register and immediately after that the other constant is written to the second register. Then the registers are read. Since reading is already measured and considered working properly, if we exceed the maximum possible delay, we know there is a problem with writing (again either in the hardware or with the slave controller).

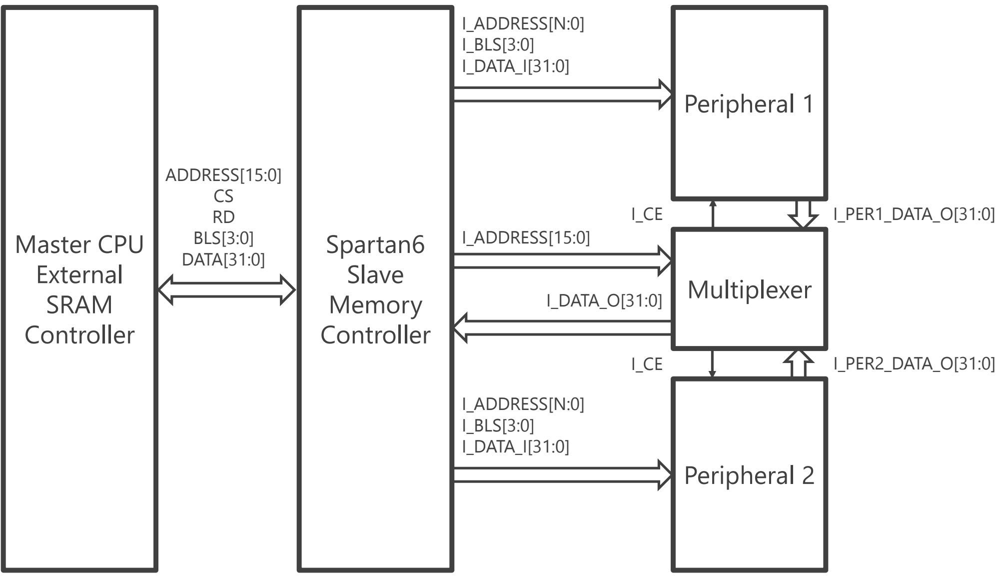

Figure 2.7. Spartan-6 slave memory controller entity

Lastly, memory reading (not writing) can happen rapidly, by keeping **RD** asserted but changing just **ADDRESS[15:0]**. This is how **ADDRESS[15:0]** can be a controlling signal and requires filtering.

2.7 Power Stage Module

Power stage module, called **LX_PWR**, is the module powering up to 4 PMSM motors and connected using a custom bus. This is a single-master, multi-slave bus, meaning that more power stage modules can be connected to a single control board. The bus in described in section 3.4. The control board is sending PWM settings for each axis (as well as enabling and disabling the bridges) and receives back current in each axis of each motor. IRCs is wired separately.

Chapter 3
FPGA Peripherals Analysis

In section 1.1 we described a PMSM controller whose fast control loop needs to be implemented in FPGA. Alongside that we need to implement a peripheral to communicate with power stage module and to read up to 4 IRC inputs.

3.1 Parallelism

When designing FPGA cores, the first thing to determine is how much parallel the design should be, i.e. how much tasks each part of the chip should do. The fastest options, however very demanding in terms of the chip size, is to dedicate each part of the task to a different part of the chip. These are then connected sequentially into stages, each stage doing its part of the task and passing the data to the next stage in one or more clock cycles, depending on the design. This is designed in mind that when each stage completes, the previous stage provides new data every time for it (in our case it would be data for another PMSM motor) and thus the pipeline doesn't have to idle and maximizes the utilization of the chip. Thus in ideal state, we would be able to run the controller on frequency by dividing the source clock (50 MHz) with number of clock cycles needed for the slowest stage. This implies that the maximum performance is achieved with balanced pipeline, when each stage needs exactly the same amount of clock cycles as the other ones. Let's see if we can design a pipeline like that, best is to start with resource usage: In the fastest

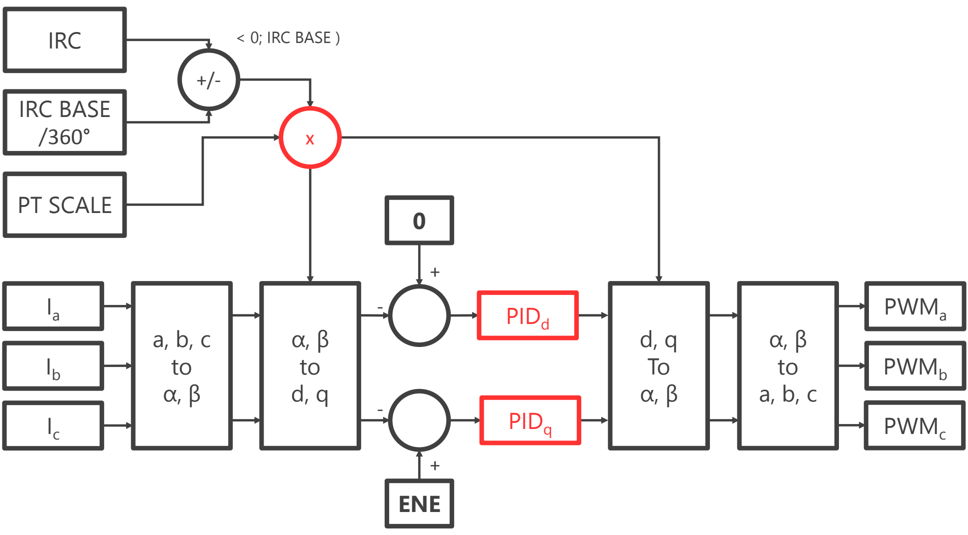

Figure 3.1. Resource usage analysis of the controller

scenario, each motor has it's own controller. Let's see how much resources would need the highlighted parts in figure 3.1. Assuming 32-bit numbers, each multiplication needs

3 **DSP48** slices (because it is 18-bit multiplier). So a single PID controller would need 9 **DSP48** slices and 3 more would be need for multiplications prior PID controller entry. This is 21 **DSP48** slices per motor, meaning that we need astonishing 84 **DSP48** slices with 4 PMSM motors. And there are only 16 **DSP48** slices available on our Spartan-6. To reduce that, there could be just a single PMSM controller for all motors, running four times per iteration. However integral part of PID requires an accumulator and derivative part of the PID needs previous values of the deviation, in other words, they have state variables. Therefore a context switching is necessary. This is shown on figure 3.2. This

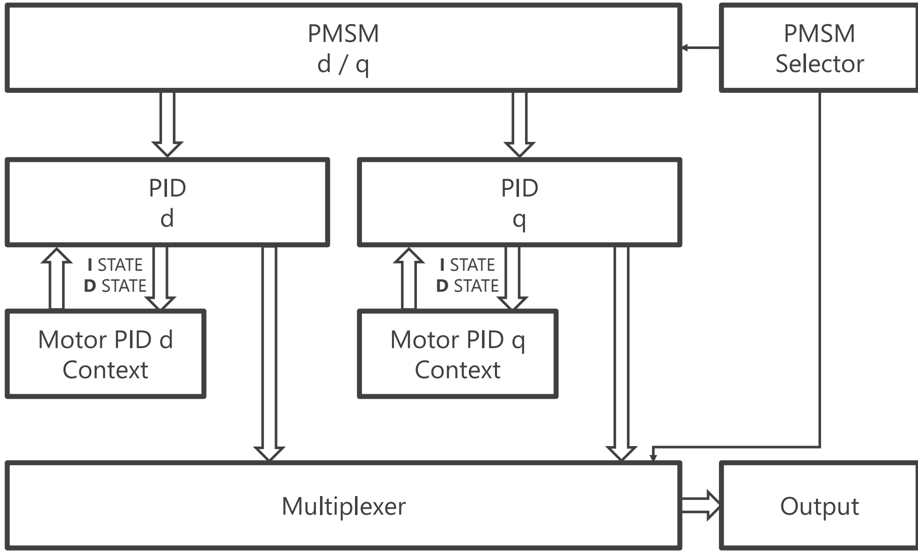

Figure 3.2. One PID in the controller

would save 63 **DSP48** slices. This means that we still need 21 **DSP48** slices, which is still too much. So lets consider using just one PID for both d and q axes, running twice and needing twice more cycles to complete. The PID controller part would change slightly, see figure 3.3. This would save another 9 **DSP48** slices, meaning that we need 12 **DSP48**

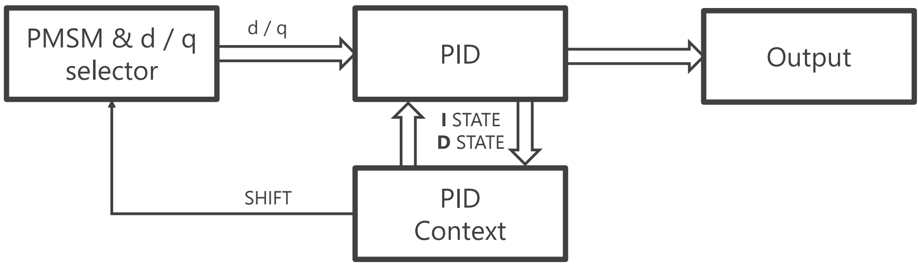

Figure 3.3. One controller for all motors

slices, which are finally available. Let's take a look on the other parts of the controller: Which means we are faced with more multiplications and even division. This is a full stop

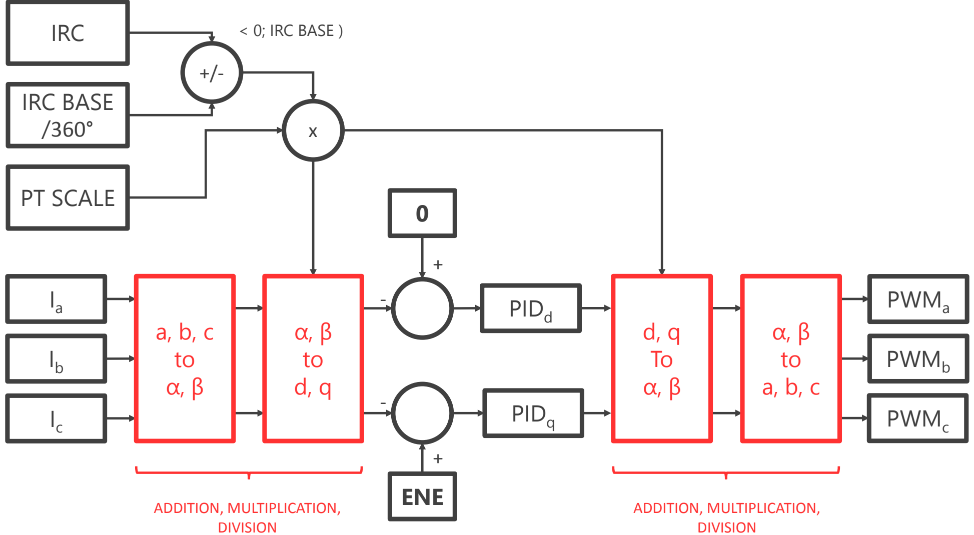

Figure 3.4. Further resource usage analysis of the controller

even for a partially parallel design due to lack of resources. Therefore we primarily need to save resources and use a much more sequential design. Consider using just a single 32-bit multiplier multiplier, using 3 **DSP48** slices, and let's try to wire it to make it used by all parts of the pipeline. And let's try to do the same for adding because that occurs frequently as well. With that, there has to be an entity commanding these two, that is accepting an instruction and decoding it, another entity passing the actual values and finally one more entity to pass the value to next stage. It's visualized on figure 3.5: In other words, for the

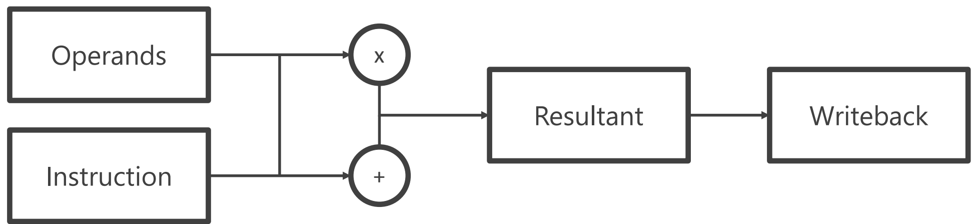

Figure 3.5. Pipeline with single ALU for all operations

first entity, we need an instruction fetch and decoding, then reading operands for the second entity, then do an actual operation and then write it somewhere. So we get something very close to a 5-stage RISC processor pipeline. 5-stage RISC processor pipeline consists of instruction fetch, instruction decode (during which registers are read), execution, memory and writeback. Oftenly, writeback is merged with memory access, making it into a 4-stage pipeline (note that it's oftenly still called 5-stage pipeline). Now we need to check if this is going to have sufficient performance. We have 4 PMSM motors, 50 MHz frequency and required 20 kHz frequency for control for each motor, giving us 80 kHz frequency in total.

Figure 3.6. 5-stage RISC pipeline

50 MHz / 80 kHz is 625 cycles for controlling a single PMSM motor.

3.2 Selecting A Synthesized Processor

If we need to use a processor structure, best way is to start from an existing processor and adapt it for our needs. In section 2.3 we conducted a study for synthesized processor cores with the intention of using them as a master CPU. Let's reconsider their use again, with the intention of using a minimal CPU core serving as a co-processor. We also know some requirements obvious from above - hardware multiplier, which is supported by all cores presented there, however we want a multiplier with 32-bit result, ruling out **OpenMSP**. **Plasma MIPS** was red flagged for low frequency, 25 MHz, halving available cycles for out control here - and thus is not a good option either. **MicroBlaze** would be ruled out here for being optimized for larger cores, which means we have two options left: **SecretBlaze** and **MB-Lite+**. In that section we concluded that **MB-Lite+** would rather be a candidate for a minimal CPU core over **SecretBlaze**. And so we chose **MB-Lite+** as the starting point.

From the first moment it became obvious we would need to modify the core for various reasons, described in the next chapter, and so decided to call our core **Tumbl** (as that's what **MB-Lite+** is sometimes referred as). The first test was made immediately - to check how many instructions a simple PID controller would need (with pre-computed tables etc.), which ended up to around 60 - 70 cycles for 3 axes and without any code to obtain the current from sensors (expected to require no more than a hundred of cycles). This in total would leave another 400 cycles for additional measures, such as anti-windup effect, thresholds for maximum current provided to the motor. With possible later extension to 8 PMSM motors, there would be still about 100 cycles for additional operations. For a reference, some additional operations above bare controller will be necessary, such as prevention from windup effect and for security (checking for limits on both required current to move the motor for a collision check, or simply check the positional limit switches in case of a robotic arm). Chapter 7 talks about FPGA simulations, which were used to find out the amount of cycles needed to run the core of a PID controller.

3.3 IRC Peripheral

Aside from the main controlling peripheral, we also need to capture the current motor position using incremental encoder. IRC signals are shown on figure 3.7. Two main signals for encoding, **A**, **B**, which are shifted by a quarter of signal period, to be able to encode increment and decrement. **IDX** is used to find base value, generally in log. 1 on zero value, periodically occurring when motor fully rotates and **MARK** signal to know which way to

find nearest **IDX** (**MARK** is not accurate, you will need to use **IDX** afterwards to find the actual base or zero value). All these are already handled in delivered quad counter entity.

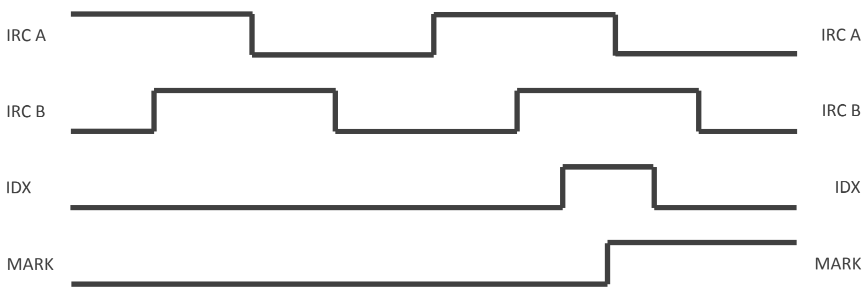

Figure 3.7. IRC signals

The only problem arises that the quad counter, counting the pulses on **A** and **B**, needs fair size of the chip due to need to implement sufficiently large counter (we decided to keep 32-bit counter, minimum could be around 20-bit counter, depending on the motor resolution), and that for all 4 IRCs (or all 8 IRCs if extended further). Like with the controller task, IRC handling doesn't require to run on 50 MHz frequency, generally 1 MHz is sufficient (this depends on maximum speed of the motors and their resolution). Thus there is again the idea of using a processor structure, yet specialized as the task here is very simple.

Let's reduce the counter to 8-bit, and try to find a way how to extend it to 32-bit, when using a single accumulator for all IRCs. Consider Q is the 32-bit accumulated value, C is the 8-bit counter on the IRC, and $s(X, n)$ as a signed extension of value X to n bits. You can outline the process as:

$$\begin{aligned} Q_s &= s(C, 32) - Q \\ Q_s &= s(Qs[7:0], 32) \\ Q &= Q + Q_s \end{aligned} \tag{3.1}$$

where we introduce Qs as intermediate result to be able to distinguish between increment or decrement, effectively being able to increment by 127 pulses or decrement by -128 pulses in a single iteration. The equations can be slightly altered to

$$\begin{aligned} Q_s &= s(C, 32) + \overline{Q} + 1 \\ Q_s &= s(Q_s[7:0], 32) \\ Q &= Q + Q_s \end{aligned} \tag{3.2}$$

to simply use an addition or written as a one-liner

$$Q = Q + s((s(C, 32) + \overline{Q} + 1)[7:0], 32) \tag{3.3}$$

which basically says to: subtract Q from C, which is signed extended to 32-bits and store just the lowest 8 bits into Q_s, then sign extend it and add back to Q. Now we also need to capture the value of index. When **IDX** is asserted, IRC quad counter will save the lower 8 bits and that **IDX** was asserted. Then the operation to count the index is identical, just C is the value when **IDX** was asserted. We only have to reset it, so it doesn't add it again in next iteration.

A requirement for this to work is that there is no overflow in the 8-bit counter, which is

running on full 50 MHz. If IRC is supposed to catch a pulse at 1 MHz frequency, it can fill up 127 pulses in 127 microseconds, or with around 8 kHz. So it's likely we can further reduce the 8-bit counter to save more chip space. The IRC peripheral, called **IRC co-processor**, is detailed in section 6.1.

3.4 Power Stage Module Communication

The communication between the control board and the high performance board is using custom bus, type single-master and multi-slave, where the target slave is determined from the first data in the protocol. The protocol itself is not yet fully finalized (on both control board and power stage module) and thus for this thesis we only designed master to slave communication peripheral. The bus uses three pins: **MOSI**, "master out, slave in", transmits the actual data; **SYNC** is in log. 0 when there is data transmission, otherwise in log. 1, **CLK** are the reference clocks passed from mater to slaves for synchronous timing. Each transmission ends with 8-bit CRC to validate the transaction. Timing diagram of the transaction is on 3.8:

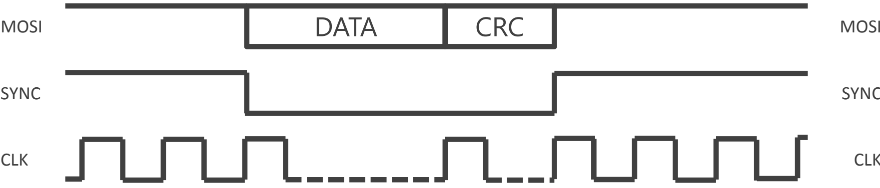

Figure 3.8. Power stage module bus transaction

At the beginning, **SYNC** is asserted along with **MOSI** with the first data bit, data transaction starts with LSB. With each clock, data is shifted by one bit. At the end of the data transmission, CRC follows immediately with **SYNC** still being asserted. **SYNC** is only released after one the last bit of CRC is sent and has to remain in log. 1 for at least one cycle before new transmission can begin. The whole transmission cycle is periodical, with frequency around 20 kHz.

The data for transmission are stored in dual-port BRAM in a specified format. First we need to be able to store two buffers, so that we can set them up without having to potentially interrupt transmission. Second, the messages may have variant length. After transmitting one message, the transmitting side needs to be navigated to the address of the next message or signalled it just transmitted the final message. The word size is 16 bits and we won't likely need more than 256 words (which can be easily extended if needed) therefore with two buffers we need 1 kiB memory (on Spartan-6 this will use one 9 kib BRAM). The communication peripheral, in direction from master to slaves, is called **LX Master** and is detailed in section 6.2.

Chapter 4
Tumbl Co-processor Core

This chapter describes Tumbl co-processor core in detail and does summarize the features and structure of the core along with it state vectors. However it's necessary to know the structure of it's predecessors, namely **MicroBlaze** in [10], **MB-Lite** in [12], and **MB-Lite+** in [13]. This chapter will not describe parts and peripherals of the core that were not altered from its predecessor and expect familiarity with **MicroBlaze** assembler.

4.1 Overview

Tumbl processor is a RISC Harward architecture core with 4-stage pipeline, divided into instruction fetch, instruction decode, execution and pipeline. All instructions except branching take 1 cycle whereas branching takes 2 or 3 cycles depending on whether delay slot was used or not. The processor core is derived from MB-Lite+ (which is derived from **MicroBlaze**) but has been mainly simplified by removing unnecessary peripherals and modified to be used as a co-processor. The core is scalar, meaning that waiting at any stage of the pipeline automatically halts entire core. The instruction set is a reduced set of **MicroBlaze** instructions but several new instructions were added and their binary encoding has been altered in some cases. The core flow is visualized on the following figure:

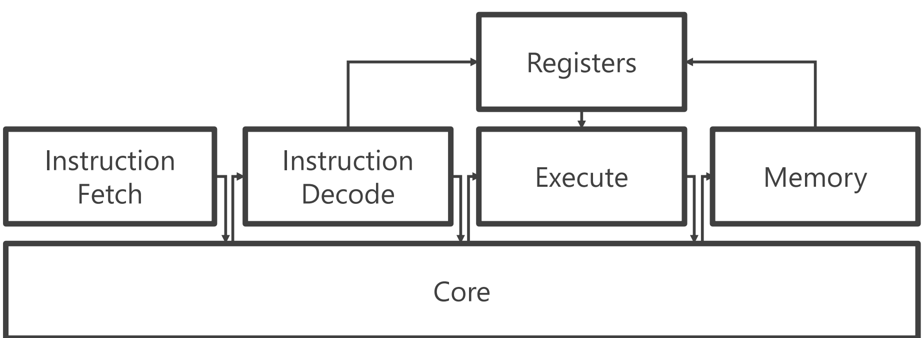

Figure 4.1. Tumbl co-processor pipeline

The following is an overview of changes of **MB-Lite+**:

- **FSL:** Fast-Simplex link is an interface to connect additional cores to **MicroBlaze** in a way that doesn't interfere with the processor's critical path and is commanded with special instructions. The CPU writes on the bus in the first cycle and then either in the next cycles or several cycles later reads the result from the core. This means it can execute other instructions should the core need many cycles to finish but all has to be optimized manually. In our case on Tumbl CPU however, we don't need such link as we simplify the core as much as we can. Therefore FSL support has been removed completely.

- **Wishbone:** A popular bus connected to the processor as external memory interface used by many cores. Unlike with FSL, a core requiring a lot of cycles to execute the request will halt the processor. Regardless, as with FSL, we simplify the core as much as possible and thus Wishbone bus has been removed and the address space is used for generic external memory interface.

- **JTAG:** JTAG is a debugging interface standard for ASIC CPUs to get information about the state of CPU peripherals. On-chip debugging is done by master CPU in our application and we can simulate the behavior of the Tumbl processor in a simulator, eliminating the need for JTAG. Therefore JTAG has been omitted.

- **External memory interface:** Originally **MB-Lite+** includes a "data memory bus selector", an interface to optimize slow cores connected to external memory. Not only this is a duplication towards FSL (and worse, because you can't control the waiting on slow cores in any way) but our application is not going to support more than one-cycle read / write operations. External memory bus has been trimmed of this "selector" and it's interface has been changed to match Master CPU's interface so they can share it.

- **Instruction encoding:** Since we are not using all **MicroBlaze** instructions we changed binary encoding of several instructions, namely `CMP`, `CMPU` to allow more variants for the instructions.

- **Delay slot:** Some branching in **MicroBlaze** exists only in a variant with a delay slot, such as branching with link. Tumbl adds a variant of such branching without delay slot. With that in mind, instructions for returning from interrupt and subroutine have also a variant without delay slot and have been fixed to jump on the correct address whether the branching was with delay slot or not.

- **Conditional execution instructions:** New instructions `IT` – if-then, `ITT` – if-then-then and `ITE` – if-then-else have been added to optimize code fragments and do not require branching in order to skip one or two instructions. Up to 100 % performance gain possible in specific cases.

- **Assembler changes:** This doesn't belong to the core directly but conditional branching instructions' assembler syntax was changed to have operands similar to conditional execution instructions.

- **Synchronization:** New `HALT` instruction was added causing the CPU to halt and wait for further commands from the master CPU.

- **Bugfixes:** Several errors in **MB-Lite+** implementation were fixed, such as machine status register being implemented as a latch.

4.2 Minimalization Of The Core

Minimalization of the core is basically removing unnecessary peripherals that are very unlikely to be used in applications of the core (our own or possible other applications) and then defining several LUT expensive parts of the core as a generic option. Our Spartan-6 has only 5720 LUTs, from which Tumbl has roughly 800 - 1200 LUTs in our application with

current configuration. Same goes for DSP blocks, where we use 3 out of 16 for hardware multiplier. It's hard to estimate the actual number of LUTs used for the core, as it depends on it's configuration and wiring of other peripherals. As explained in the overview FSL, JTAG and Wishbone peripherals have been removed because they won't be needed.

4.3 Co-processor Modifications

Generally co-processor modifications can be described as the means of a master unit to control the co-processor and feedback from the co-processor for synchronization. Then you also need to provide means of master unit to be able to debug the co-processor. The following figure illustrates the wiring of Tumbl on master CPU memory bus and peripherals on Tumbl external memory bus.

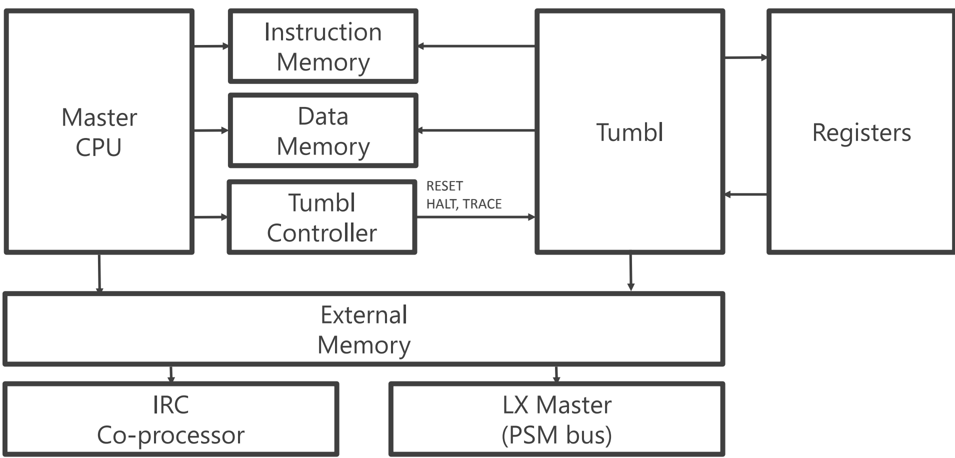

Figure 4.2. Tumbl core interconnection

We're using dual-port BRAMs for Tumbl's instruction and data memory, one port goes to master CPU and the other port to Tumbl. Thus master CPU is responsible for uploading firmware for Tumbl and the data memory can be used as registers accessible both to master CPU and Tumbl through a fast and independent interfaces. These can be used for direct data transfer between the two cores. Master CPU has two registers, one to set reset, interrupt, halt and tracing for debugging, the other one to toggle step for tracing. See appendix D for a detailed registers overview. For synchronization purposes, Tumbl was equipped with `HALT` instruction with 5-bit exit code, causing the core to halt on that instruction until the halt state is reset. Tumbl core cannot resume itself without the help of external unit if it was halted. The exit code is an output of the core alongside with the halt state, thus this can be either wired to interrupt the master unit or wait for the master unit until it polls Tumbl core etc. based on the exit code. Alternatively you can wire it to a timer unit which would automatically resume it after a certain period has passed. `HALT` instruction is detailed in section B.5 and the signals for tracing and halting the core are described in section 5.5.

Tumbl co-processor core does support interrupts but it isn't designed for scheduling interrupts. Thus there is no interrupt status register nor a way to provide "reason" for the interrupt, which right now can be an enity wired to it's external memory or master CPU.

Interrupt support is only present to deal with an unexpected state and should be forwarded to master CPU.

External memory interface is wired in a manner so it is accessible for both master CPU and Tumbl. This was done in order to provide means of debugging the output of Tumbl core and allowing master CPU to take over Tumbl to deal with unexpected scenarios requiring immediate intervention from master CPU. On the external memory interface Tumbl is seen as a slave unit, thus in a collision between master CPU and Tumbl in using the bus it will wait a cycle for master CPU transaction to finish. External memory interface is designed so all transactions are taking only a single cycle, while the interface itself can support multiple cycle read and write opeartions for Tumbl side (at a cost of performance penalty) master CPU would have to access it under a different chip select with timings set for slower transactions. External memory interface signals were changed to mirror master CPU signals wired to the FPGA so it can be shared. Address space was reduced just to 64 kB as it's more than sufficient when we are not using external RAM memory for Tumbl code. External memory interface is detailed in section 4.7.

4.4 Core Modifications

Initially we were not satisfied with the core design as timing on Tumbl core is critical and there are certain patterns that have a performance penalty that could be fairly easily eliminated. From previous analysis we are aiming for up to 4 PMSM motors each to be serviced at 20 kHz frequency, with planned extension to 8 PMSM motors. That gives us 160 kHz frequency in total and with 50 MHz clock on Tumbl we have 312 cycles to service each DC motor. If we were able to use 72 MHz clock we would have up to 450 cycles. However when synthesizing **MB-Lite** core on Spartan-6 with barrel shift enabled and multiplier enabled, the maximum frequency with plain processor core was around 60 MHz, ruling out 72 MHz frequency as a possibility with the current design as hardware multiplier is necessary (frequent multiplications occur with PID compensators). Thus the aim is to look on likely used code fragment and try to optimize them to achieve more performance. We will also need some form of division, which is described in section 4.6, for which we need to count the leading zeroes, resulting in a new instruction. This ended up with modifying three things:

- **Conditional execution**: Code snippets like **if-then-else** end up with a severe performance penalty.
- **Count leading zeroes**: A new instruction, counting the leading zeros of the passed unsigned value.
- **Branching with link**: All branching with link instructions, and return from subroutine or interrupt instructions are only in a variant with a delay slot.

Then additionally to that, aside from `HALT` introduced above for synchronization, `CMP` and `CMPU` instructions encoding was changed to allow `CMPI` and `CMPUI` variants with immediate value as source operand b. See chapter 4 of [10] to get familiar with **MicroBlaze** assembler.

4.4.1 Conditional Execution

Conditional execution primarily improves performance on branching over one or two instructions as branching takes 3 cycles in total or two cycles with delay slot. Assume the following C pseudocode.

```
if  (R10 < 0x128)
   R11 = 0x10;
else
   R11 = 0x20;
```

Such code would be translated to the following assembler code.

```
   ADDI R19, R0, 0x128
   CMP  R12, R19, R0
   BLTI R12, lesser
   BRID common
   ADDI R11, R0, 0x20
lesser:
   ADDI R11, R0, 0x10
common:
    ...
```

Now let's walk through it. Each instruction takes one cycle. When `R4 < 0x128` the branching take extra 2 cycles to flush the pieline and skips two instructions. So in that case this snippet takes 6 cycles. In the other case one of the jumps acts as `NOP` and the other jump takes extra cycle as it's with delay slot. One instruction is skipped so we get 6 cycles again. We chose to use approach similar to the one on ARM processors in Thumb mode and created three instructions.

```
; If-Then instruction - evaluates Cnd(Ra, Rb) and
IT Cnd, Ra, Rb
; evaluates this instruction if true, otherwise acts as NOP
ADDI R10, R0, #0x1
```

```
; If-Then-Then instruction - evaluates Cnd(Ra, Rb) and
ITT Cnd, Ra, Rb
; evaluates this instruction if true, otherwise acts as NOP
ADDI R10, R0, #0x1
; evaluates this instruction if true, otherwise acts as NOP
ADDI R11, R0, #0x2
```

```
; If-Then-Else instruction - evaluates Cnd(Ra, Rb) and
ITE Cnd, Ra, Rb
; evaluates this instruction if true, otherwise acts as NOP
ADDI R10, R0, #0x1
; evaluates this instruction if true, otherwise acts as NOP
ADDI R10, R0, #0x2
```

The flow is illustrated in the tables above. `IT` instruction, if-then, will execute the following instruction only if the condition it evaluates is true, otherwise flushes the next instruction

with NOP. Similarly, `ITT` instruction, if-then-then, will execute the two following instructions only if the condition is true, whereas `ITE` instruction, if-then-else, if the condition is true, it will execute the following instruction and flush the instruction afterwards. In the other case it will flush the first instruction and execute the instruction following it and thus works as "if condition, then the next instruction, else the following instruction". All `IT`, `ITT`, `ITE` instructions have variants for unsigned comparison and allowing `Imm` operand b instead of `Rb`. So in total there is 12 new instructions: `IT`, `ITT`, `ITE`, `ITU`, `ITTU`, `ITEU`, `ITI`, `ITTI`, `ITEI`, `ITUI`, `ITTUI`, `ITEUI` where `U` suffix means unsigned comparison, `I` suffix means `Imm` as source operand b and `UI` suffix means both. Conditional execution has one

exception, which is when `IMM` instruction is one of the conditionally executed. Because `IMM` instruction is implicitly added by **binutils**, the conditional execution core will see an `IMM` instruction and the following instruction as one. For example the following snippet for the assembler to compile

```
    ITE   LT, R12, 0x128
    ADDI R11, R0, 0x20000000
    ADDI R11, R0, 0x10000000
```

will in fact be

```
    ITE   LT, R12 0x128
;   ---
    IMM   0x2000
    ADDI R11, R0, 0x0
;   ---
    IMM   0x1000
    ADDI R11, R0, 0x0
```

but either `ADDI R5, R0, #0x20000000` or `ADDI R5, R0, #0x10000000` will be executed. Now if we use the new assembler on the original pseudocode

```
    ITEI LT, R12, #0x128
    ADDI R11, R0, #0x10
    ADDI R11, R0, #0x20
```

we can see we're able to do it in 3 cycles and get a 100 % performance boost. We also need to use just two registers instead of three. See B.5 to see conditional execution instructions encoding.

4.4.2 Branching With Link

Branching with link on **MicroBlaze** does not have a variant with no delay slot and the reason for it most likely is that all variants of branching with link instructions will pass the current `PC` in execution stage. Branching with link is used for subroutines.

```
sub:
    ADD    R3, R5, R6
    RTSD   R15, 0x8
    NOP
...
main:
    ADDI   R5, R0, 0x10
```

```
    ADDI  R6, R0, R19
    BRLID R15, sub
    NOP
```

In C language ABI, `R15` is used as link register. So at **BRLID R15, sub**, current `PC` is stored into `R15`, then `NOP` is executed and then finally we are in "sub" subroutine. Now `RTSD`, return from subroutine with delay slot, has to jump on `PC + 8` because instruction at `PC + 4` was executed in the delay slot. So assume you want to add variant to **MicroBlaze** with no delay slot (originally **MicroBlaze** had only branching with delay slot) and keep compatibility with currently compiled code. So you'd have to pass `PC - 4` when branching which would complicate the core a bit. So what we do here is to fix the `PC` passing in order to create branching with link without delay slot. Simply if you branch with link with no delay slot `PC` is passed whereas if you branch with link with a delay slot, `PC + 4` is passed, which in the core is forwarded from decode stage. We had to change C ABI a bit however but the code above would now be a little simpler.

```
sub:
    ADD   R3, R5, R6
    RTS   R15, 0x4
...
main:
    ADDI  R5, R0, 0x10
    ADDI  R6, R0, R11
    BRLI  R15, sub
```

In fact the correct way here would be to utilize the delay slots here, it would save two cycles.

```
sub:
    RTSD  R15, 0x4
    ADD   R3, R5, R6
...
main:
    ADDI  R5, R0, 0x10
    BRLID R15, sub
    ADDI  R6, R0, R11
```

Notice that `RTSD` still passes `#0x4`. We also added a version with no delay slot for returning instructions. Thus in total the following instructions were created: `RTS`, `RTI`, `BRL`, `BRLI`, `BRAL`, `BRALI`.

While from performance point of view it's better to use delay slot where possible the problem comes that in the core it's not part of the processor's state. It's really just that the instruction is already loaded in the pipeline and not getting flushed. Therefore you cannot branch in a delay slot (would not make sense anyway) or use delay slot branching with conditional execution instructions introduced above (which actually served as a motivation add these), other limitations are not being able to interrupt the core at the moment or improper behavior if `IMM` instruction is in delay slot (this would be propagated for the instruction where we branched to and omitted from instruction located after `IMM` instruction. Then many times delay slot has to be filled with `NOP` because the actual branching is in a sequence with the previous instruction.

4.4.3 Count Leading Zeroes

A simple instruction, cheap in hardware, providing an easy and effective way of counting leading zeroes. Instruction is labeled as CLZ and $31 - \mathrm{ceil}(\log2(n))$, in which we assume $\mathrm{ceil}(\log2(0)) = -1$. An example: CLZ returns 32 for value 0, 31 for value 1, 30 for value 2 and 21 for value 1024 (bit 10 is 1). This can be used with more advanced division algorithms.

4.4.4 Instruction Binary Encoding Changes

Given that many **MicroBlaze** instructions were removed, binary encoding of CMP and CMPU instructions was changed to add CMPI and CMPUI variants, where source operand b is an immediate value. Originally CMP was encoded as a special case of RSUB, see table 4.4.4. Since we have a few more empty identifiers in **0-5** bits we changed it to the following with

Directive	0-5	6-10	10-15	16-20	21-31
RSUB Rd, Ra, Rb	000001	Rd	Ra	Rb	00000000000
CMP Rd, Ra, Rb	000001	Rd	Ra	Rb	00000000001
CMPU Rd, Ra, Rb	000001	Rd	Ra	Rb	00000000011

Table 4.1. Original CMP and CMPU binary encoding

source operand b as a register, see table 4.4.4. And then we set **bit 3** to indicate immediate

Directive	0-5	6-10	10-15	16-20	21-31
CMP Rd, Ra, Rb	010010	Rd	Ra	Rb	00000000000
CMPU Rd, Ra, Rb	010010	Rd	Ra	Rb	00000000000

Table 4.2. New CMP and CMPU binary encoding

value as source operand b, see table 4.4.4. See section B.5 for the nomenclature used and

Directive	0-5	6-10	10-15	16-31
CMPI Rd, Ra, Imm	010010	Rd	Ra	Imm
CMPUI Rd, Ra, Imm	010010	Rd	Ra	Imm

Table 4.3. New CMPI and CMPUI binary encoding

the semantics of the instructions.

4.4.5 Assembler Changes

Conditional branch instructions syntax has been adapted to the syntax of conditional execution instructions. This is summarized in table 4.4.5. See section B.5 for the nomenclature used and the semantics of the instructions.

Old Syntax	New Syntax
`BEQ{I}{D} Ra, Rb/Imm`	`BRC{I}{D} EQ, Ra, Rb/Imm`
`BNE{I}{D} Ra, Rb/Imm`	`BRC{I}{D} NE, Ra, Rb/Imm`
`BLT{I}{D} Ra, Rb/Imm`	`BRC{I}{D} LT, Ra, Rb/Imm`
`BLE{I}{D} Ra, Rb/Imm`	`BRC{I}{D} LE, Ra, Rb/Imm`
`BGT{I}{D} Ra, Rb/Imm`	`BRC{I}{D} GT, Ra, Rb/Imm`
`BGE{I}{D} Ra, Rb/Imm`	`BRC{I}{D} GE, Ra, Rb/Imm`

Table 4.4. Conditional branching assembler changes

4.5 Processor Pipeline

Tumbl processor pipeline is divided into 4 stages, each runs for a single cycle. It's visualized on the following figure 4.1. The stages are:

- **Instruction fetch:** Loads an instruction from instruction memory and stores the current program counter.
- **Instruction decode:** Decodes an instruction binary into context for execution, such as ALU operation, where to take source operands from etc.
- **Execution:** Evaluates an instruction based on its context description.
- **Memory and Writeback:** Performs all memory operations - reading from and writing to memory (whether internal data memory or external memory) or registers.

Apart from that, the core contains two other components.

- **General Purpose Registers File:** A storage for registers, allowing up to reading 3 registers at once and storage for 31 registers, each 32-bit width (R0 is always zero)
- **Core Component:** This is the interconnect holding the pipeline components state.
- **Top Module:** Top level interconnect providing interconnection of the core to other components and wires instruction and data memory alongside with registers.

The following subsections overview pipeline, detailed documentation for the structures and signals is in chapter 5.

4.5.1 Instruction Fetch

Instruction fetch is the component of the pipeline responsible for `PC` register. It schedules reading from instruction memory using the current `PC` as an address. If we are branching, sets the next `PC` from branching input otherwise increments `PC` by 4. Note that no check is performed whether the read operation is valid or not. The width of program counter is fixed to 32 bits regardless of the actual address space. Instruction fetch also passes the current program counter to instruction decode.

4.5.2 Instruction Decode

Decode is nothing but a parser of the bytecode. It creates a context for execution, which consists of:

- **Program counter:** Program counter for the currently parsed instruction is passed to decode to process branching etc.
- **Operands:** Which register is the destination operand and which registers are source operands or the immediate value for source operand b.
- **ALU context:** What kind of operation ALU should do (add, subtract, multiply, etc.) also the type ALU operands (register or immediate value).
- **Conditional execution:** Whether we are doing conditional execution (if-then type instructions) and what is the condition (this is also used for conditional branching).
- **Memory operation:** Type of memory operation, whether we're writing to a register or data memory etc. and size of the writing. Then also reading from data memory or external memory.
- **Co-processor synchronization:** Whether we should halt the core on next instruction.

4.5.3 Execution

Execution is the stage of pipeline which performs the actual operation in a single cycle. Execution is able to perform:

- **Addition:** Sums values of two operands with support of using and storing carry bit. Note that subtraction is solved as adding negated value and one.
- **Comparison:** Compares two operands. The result is either stored to a destination register for `CMP` and derived instructions or used immediately for `IT`, `ITT`, `ITE` and derived instructions.
- **Logical operations:** Performs `AND`, `OR`, `XOR` operations (and negates source operand b for `ANDN`).
- **Simple shift:** Shifts to the right by one for `SRA`, `SRL`, `SRC` instructions.
- **Barrel shift:** Performs parametrized shift to left or right for barrel shift instructions. Configurable by a generic.
- **Special purpose register access:** Read and write to special registers, in our case only it's only carry and interrupt enable bit for machine status register.
- **Extensions:** Register extensions for `SEXT8` and `SEXT16` instructions.

- **Multiplication:** Hardware multiplication for `MUL` and derived instructions. This part has the longest critical path. Configurable by a generic.

- **Count leading zeroes:** Hardware implementation of `CLZ` instruction.

Aside from instruction context, execution input is also values of up to three registers from general purpose register file. It outputs context to memory and writeback. There is also a hazard context where execution forwards data back to itself. It deals when in two subsequent instructions the first instruction destination register is used as a source operand in the second instruction. There is also a similar hazard where such sequence happens within 2 cycles. Example of the hazardous situation is:

```
ADDIK R10, R0, 0x100
ADDIK R11, R0, 0x200
MUL   R12, R10, R11
```

`MUL` instruction needs both `R10` and `R11` value. `R10` is being written at the moment to general purpose register file, as memory and writeback has just processed it. So it has to be taken from the writing port right away. However `R11` is not even yet processed by memory and writeback. So it has to be forwarded back from the execution itself so it can be used right away in next cycle. However note that when you want to load data from memory and then use it in next instruction then you cannot forward them as it doesn't previously come from execution. In this case the core stalls for a cycle for the data to load. This applies both to data memory and external memory.

4.5.4 Memory And Writeback

Memory and writeback performs writing to registers, data memory and external memory. Addresses with first n bits low (configurable) are considered data memory, the other addresses are considered external memory. Data memory is intended to consist only from block ram memory, all registers and other modules are meant to be part of the external memory. This entity is also responsible for read operations from data and external memory.

4.5.5 General Purpose Register File

General purpose register file is just a storage for 32 32-bit registers, which needs a 9 kiB dual-port BRAM per bank. As up to three simultaneous reads can occur at once, there are 3 register banks, with shared writing on one port and the other port open for reading. To handle hazard situations, described in subsection 4.5.3, it checks if the register to be read is being written to at the moment and decides whether to read it from the reading side or whether to forward it from the writing side. This is accomplished with dual-port BRAM in write-first mode. General purpose register file is seen more as a custom component of the core and interconnected using the top module, because it is directly dependent on Xilinx BRAM. General purpose register file also holds state of the processor and is wired to secure that when reset signal is asserted, `R0` register is set to 0.

4.5.6 Core Component

Core component is mainly an interconnect between the stages of the pipeline and a driver for clock signal. It stores state variables, such as machine status register, pipeline flushing whether partial, which occurs during conditional execution or branching and schedules interrupts, and full, which occurs during reset. Synchronization and debugging is also handled by the core component, providing a single clock when being traced upon request or releasing the core from halting (whether it occurs from inside Tumbl or externally).

4.5.7 Top Module

Finally top module packs it all together and provides instruction and data memory, implemented as dual-port block RAM. The second port is accessible for master CPU. Top module also wires general purpose register file and provides signals, such as whether Tumbl is halted, being traced or its current program counter.

4.6 Division

Integer division is provided through calculating inverted number using a lookup table, then multiplying and shifting the result. The lookup table can be defined for n bit width divider with m bit width remainder, were $m > n$ is recommended for a reliable result. The table is then pre-calculated as:

$$d = \text{floor}((1 << n)/r) \tag{4.1}$$

where r is the number to be inverted and d is the resulting table entry. Then store d as r^{th} element in a table. When wanting to divide online, lookup d and calculate the inverse number y as:

$$y = \text{floor}((d * r) >> m) \tag{4.2}$$

For 12-bit divider and 16-bit remainder, you get 98.7 % success rate when dividing a 12-bit number by a 12-bit divider. In the other cases the result differs only by 1. This lookup table would require 8 kiB of memory. Depending on how wide the division needs to be, Tumbl data memory would have to be adjusted to accommodate both division lookup table and general purpose data memory.

4.7 External Memory Interface

Tumbl external memory interface provides access to peripherals. It is mapped on addresses above data memory and shared with master CPU, where master CPU has priority. In case of a collision, Tumbl stalls until master CPU doesn't release the bus. The wiring is shown on figure 4.3. In addition other peripherals can issue interrupt through external memory interface.

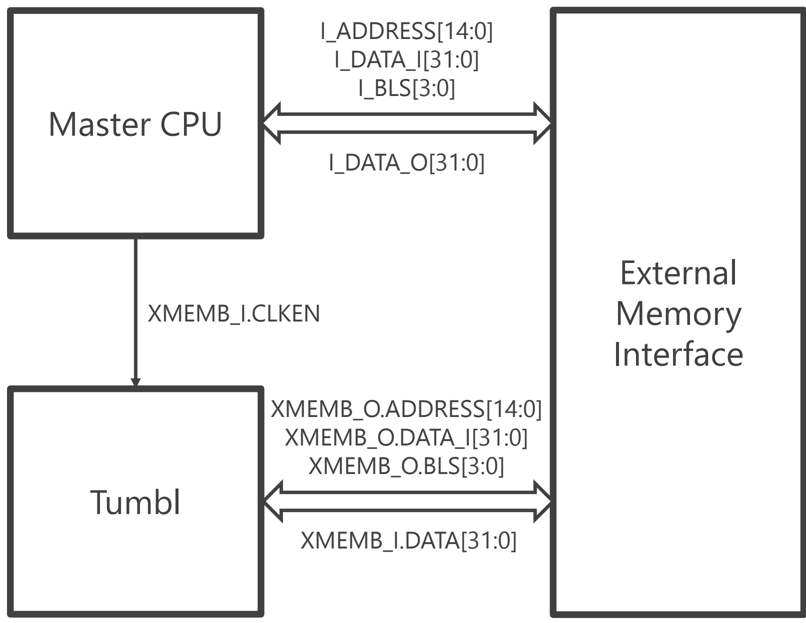

Figure 4.3. Tumbl external memory interface

4.8 C lanugage support

Custom processor core needs custom **binutils** to translate assembler code to bytes and **gcc** to compile C code into assembler code, alongside with port for **libgcc** to provide functions not supported by platform hardware and **newlib** as a basic C library. **MicroBlaze** is nowadays supported by upstream of all **binutils**, **gcc** and **newlib** and support for our processor is made from recent stable versions: **binutils 2.23.2**, **gcc 4.7.3** and **newlib 2.0.0**. See appendix C for building the toolchain.

4.9 Pipeline Balance

We made a small study regarding balance of the pipeline to determine critical path in various configurations. It was determined that without hardware multiplier and without barrel shift, our design is currently able to run on about 79 MHz or 82 MHz with compatibility mode, or 50 MHz with hardware multiplier on (regardless of the other configurations). In an attempt to make 2 cycle multiplication (with particular modification of other elements of the core), it was determined that the core would be able to run around 60 MHz. This effort was however abandoned for the moment due to heavy core modifications needed to handle all kinds of hazardous situations and mainly due to 50 MHz being sufficient for this application. It however shows that pipeline is not very well balanced and that the problem should be acknowledged later on when the processor core is to be optimized.

Chapter 5
Tumbl Co-processor Implementation

This chapter details FPGA implementation of Tumbl co-processor, the actual structure of the pipeline entities and all state and interconnection vectors. Tumbl co-processor is implemented in VHDL and with inferred operations and entities, maintaining compatibility for other than Xilinx platforms. We first describe enumerations and records, then follow up with the actual entities for each pipeline stage.

5.1 Enumerations

- **ALU_ACTION_Type:** Type specifying action for ALU in execution:

```
TYPE ALU_ACTION_Type IS (
          A_NOP,
          A_ADD,
          A_CMP,
          A_CMPU,
          A_OR,
          A_AND,
          A_XOR,
          A_SHIFT,
          A_SEXT8,
          A_SEXT16,
          A_MFS,
          A_MTS,
          A_MUL,
          A_BSLL,
          A_BSRL,
          A_BSRA,
          A_CLZ
);
```

ALU operation switch. See instruction set for mathematical representation of the relevant instructions in appendix B.

- **ALU_IN1_Type:** Source operand a for ALU:

```
TYPE ALU_IN1_Type IS (
          ALU_IN_REGA,
          ALU_IN_NOT_REGA,
          ALU_IN_PC,
          ALU_IN_ZERO
);
```

Which is either a value in a register (ALU_IN_REGA), or negated value in a register (ALU_IN_NOT_REGA), or current program counter value (ALU_IN_PC) otherwise unused (ALU_IN_ZERO).

- **ALU_IN2_Type:** Source operand b for ALU:

```
TYPE ALU_IN2_Type IS (
        ALU_IN_REGB,
        ALU_IN_NOT_REGB,
        ALU_IN_IMM,
        ALU_IN_NOT_IMM
);
```

Which is either a value in a register (ALU_IN_REGB), or negated value in a register (ALU_IN_NOT_REGB), or immediate value (ALU_IN_IMM), or negated immediate value (ALU_IN_NOT_IMM).

- **ALU_CIN_Type:** Carry bit wiring for instructions using it as an input:

```
TYPE ALU_CIN_Type IS (
        CIN_ZERO,
        CIN_ONE,
        FROM_MSR,
        FROM_IN1
);
```

In the execution stage, instructions varying just with usage of carry bit (such as ADD and ADDC) and fixed value are merged into a single small "core" and only their input wiring is changed. The four options are CIN_ZERO for forcing 0, CIN_ONE for forcing 1, FROM_MSR for using the actual carry bit from MSR and FROM_IN1 for using the most significant bit as carry bit.

- **MSR_ACTION_Type:** Output from execution stored to MSR:

```
TYPE MSR_ACTION_Type IS (
        UPDATE_CARRY,
        KEEP_CARRY
);
```

From section B.3 we know that MSR has two bits: carry bit and interrupt enable bit. The meaning of interrupt enable bit inside MSR is only whether we are servicing interrupt at the moment and thus such state is fully implicit and rather controller by instruction fetch rather than execution. Thus execution only controls carry bit and has two operations, whether to keep its current state (KEEP_CARRY) or update it (UPDATE_CARRY). How is carry bit calculated is detailed in the technical reference manual.

- **BRANCH_ACTION_Type:** What kind of branching should execution do:

```
TYPE BRANCH_ACTION_Type IS (
        NO_BR,
        BR,
        BRL
);
```

Which either is NO_BR for no branching, BR for branching, this is also used for return from subroutine or interrupt and BRL for branching with link, storing the current PC or PC + 4, depending on whether delay slot is used, into the destination register.

- **COND_Type:** Condition type for conditional instructions, see table B.2 for explanation.

```
TYPE COND_Type IS (
        COND_ALL,
        COND_EQ,
        COND_NE,
        COND_LT,
        COND_LE,
        COND_GT,
        COND_GE
);
```

Condition type COND_ALL is the default value for not doing any condition checks. The other possibilities are only used for BRC, IT, ITT, ITE and derived instructions.

- **IT_ACTION_Type:** If-then opreation type:

```
TYPE IT_ACTION_Type IS (
        NO_IT,
        IT,
        ITT,
        ITE
);
```

Which either is NO_IT for no conditional execution, IT for if-then conditional execution, ITT for if-then-then and ITE for if-then-else. See section 4.4.1 for conditional execution explanation.

- **WRB_ACTION_Type:** Write back to register file type, it is input to memory and writeback:

```
TYPE WRB_ACTION_Type IS (
        NO_WRB,
        WRB_EX,
        WRB_MEM
);
```

This is **NO_WRB** for instructions with no destination register, or **WRB_EX** for the value to be written to a register is a result of operation in execution, or **WRB_MEM** if the value to be written to a register is loaded from data or external memory.

- **MEM_ACTION_Type:** Memory and writeback input for reading or writing from data or external memory

```
TYPE MEM_ACTION_Type IS (
        NO_MEM,
        WR_MEM,
        RD_MEM
);
```

NO_MEM is for no memory action, WR_MEM is for writing and RD_MEM is for reading.

- **TRANSFER_SIZE_Type:** Size of the transaction for loading and storing instructions:

```
TYPE TRANSFER_SIZE_Type IS (
        WORD,
        HALFWORD,
        BYTE
);
```

Obviously this is WORD for 32-bit transation, HALFWORD for lower 16 bits, BYTE for least significant 8 bits. Used with SW, SH, SB for storing to memory and LW, LHU, LBU for loading from memory.

- **SAVE_REG_Type:** Type to store information about possible hazard state, when a instruction uses the resultant of the previous instruction as a source operand:

```
TYPE SAVE_REG_Type IS (
        NO_SAVE,
        SAVE_RA,
        SAVE_RB
);
```

This will command the execution to take use previous instruction destination operand as a source operand a (SAVE_RA) or source operand b (SAVE_RB), or otherwise obtain source operands normally (NO_SAVE).

5.2 Records

- **ID2EX_Type:** Interconnection between instruction decode and execution, with general purpose register file being wired between them:

```
TYPE ID2EX_Type IS RECORD
        program_counter   : STD_LOGIC_VECTOR (31 DOWNTO 0);
        rdix_rA           : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
        rdix_rB           : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
        curr_rD           : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
        alu_Action        : ALU_ACTION_Type;
        alu_Op1           : ALU_IN1_Type;
        alu_Op2           : ALU_IN2_Type;
        alu_Cin           : ALU_CIN_Type;
        IMM16             : STD_LOGIC_VECTOR (15 DOWNTO 0);
        IMM_Lock          : STD_LOGIC;
        msr_Action        : MSR_ACTION_Type;
        branch_Action     : BRANCH_ACTION_Type;
        it_Action         : IT_ACTION_Type;
        mem_Action        : MEM_ACTION_Type;
        transfer_Size     : TRANSFER_SIZE_Type;
        wrb_Action        : WRB_ACTION_Type;
```

```
        condition            : COND_Type;
        halt                 : STD_LOGIC;
END RECORD;
```

Summary of the signals:

- **program_counter:** Program counter of the instruction being parsed.
- **rdix_rA:** Register number for operand Ra - wired to execution to check for hazard state.
- **rdix_rB:** Register number for operand Rb - wired to execution to check for hazard state.
- **curr_rD:** Register number for operand Rd - certain instructions like SW, SH, SB use 3 source operands, because the value is stored on address in Rd
- **alu_Action:** Operation type for ALU in execution.
- **alu_Op1:** First ALU operand type, see section 5.1 for explanation.
- **alu_Op2:** Second ALU operand type, see section 5.1 for explanation.
- **alu_Cin:** Carry bit source, see section 5.1 for possible inputs.
- **branch_Action:** Whether this instruction is a branching instruction or not.
- **it_Action:** Whether this instruction is a conditional execution instruction or not.
- **mem_Action:** Memory and writeback operation.
- **condition:** Conditional operand, if this instruction is conditional.
- **halt:** Halt the core upon execution of this instruction.

- **ID2GPRF_Type:** Interconnection between instruction decode and general purpose register file:

```
TYPE ID2GPRF_Type IS RECORD
        rdix_rA : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
        rdix_rB : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
        rdix_rD : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
END RECORD;
```

Here we only need to know which registers to read: **rdix_rA** for operand Ra, **rdix_rB** for operand Rb, **rdix_rD** for operand Rd.

- **ID2CTRL_Type:** Interconnection between instruction decode and core component:

```
TYPE ID2CTRL_Type IS RECORD
        delayBit : STD_LOGIC;
        int_busy : STD_LOGIC;
END RECORD;
```

delayBit is set when the decoded instruction is branching with delay slot, including returns from subroutines or an interrupt. **int_busy** is set when the state of MSR (which is fed back from core component) has interrupts disabled and unset when we are returning from an interrupt using RTID instruction.

- **INT_CTRL_Type:** General interconnection for interrupt handling:

```
TYPE INT_CTRL_Type IS RECORD
        setup_int  : STD_LOGIC;
        rti_target : STD_LOGIC_VECTOR (31 DOWNTO 0);
        int_busy   : STD_LOGIC;
END RECORD;
```

int_busy has the same meaning as in **ID2CTRL_Type**. **rti_target** is the address to which we are supposed to return after the interrupt service routine is finished. **setup_int** is used to propagate to peripherals, namely instruction fetch and decode, to setup an interrupt.

- **GPRF2EX_Type:** Data transmission between general purpose register file and execution:

```
TYPE GPRF2EX_Type IS RECORD
        data_rA : STD_LOGIC_VECTOR (31 DOWNTO 0);
        data_rB : STD_LOGIC_VECTOR (31 DOWNTO 0);
        data_rD : STD_LOGIC_VECTOR (31 DOWNTO 0);
END RECORD;
```

data_rA is the value of operand Ra, **data_rB** is the value of operand Rb and **data_rD** is the value of operand Rd.

- **IMM_LOCK_Type:** Provides state for immediate value instructions:

```
TYPE IMM_LOCK_Type IS RECORD
        locked   : STD_LOGIC;
        IMM_hi16 : STD_LOGIC_VECTOR (15 DOWNTO 0);
END RECORD;
```

IMM_hi16 is set by a previous IMM instruction and is valid for a one cycle. **locked** is set depending on whether the previous instruction was IMM or not.

- **MSR_Type:** Machine status register type:

```
TYPE MSR_Type IS RECORD
        IE   : STD_LOGIC;
        C    : STD_LOGIC;
END RECORD;
```

In Tumbl, there is only support for interrupts, **IE** holds whether interrupts are enabled (or rather, if we are not servicing interrupt request), **C** is current carry bit.

- **EX2IF_Type:** Execution to instruction fetch interconnect:

```
TYPE EX2IF_Type IS RECORD
        take_branch   : STD_LOGIC;
        branch_target : STD_LOGIC_VECTOR (31 DOWNTO 0);
END RECORD;
```

Basically tells instruction fetch whether to take branch and what is the destination address, by setting **take_branch** to 1 when branching and **branch_target** to the destination address. This is set when any kind of branching occurs.

- **EX2CTRL_Type:** Execution to core component interconnect:

```
TYPE EX2CTRL_Type IS RECORD
        flush_first         : STD_LOGIC;
        flush_second        : STD_LOGIC;
        ignore_state        : STD_LOGIC;
END RECORD;
```

Provides 3 bits, all used for conditional execution. **flush_first** and **flush_second** tells core component whether to flush execution in next cycle or cycle after that (i.e. IT will set **flush_first** to 1 when its condition is met). If there is subsequent conditional execution instruction, while we are executing one (such as two IT in a sequence), then **ignore_state** is set to 1 to signal core component to ignore current conditional execution state and directly adopt the new one.

- **HALT_Type:** General halt state interconnect, for halting from within Tumbl:

```
TYPE HALT_Type IS RECORD
        halt            : STD_LOGIC;
        halt_code       : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
END RECORD;
```

Where **halt** is whether we are halted (or requesting to halt) and **halt_code** is the code passed from HALT instruction.

- **EX2MEM_Type:** Execution to memory and writeback interconnect:

```
TYPE EX2MEM_Type IS RECORD
        mem_Action          : MEM_ACTION_Type;
        wrb_Action          : WRB_ACTION_Type;
        exeq_result         : STD_LOGIC_VECTOR (31 DOWNTO 0);
        data_rD             : STD_LOGIC_VECTOR (31 DOWNTO 0);
        byte_Enable         : STD_LOGIC_VECTOR ( 3 DOWNTO 0);
        wrix_rD             : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
END RECORD;
```

This is a bit more complicated interconnect, serving both for memory operations and writeback. For memory operations, **mem_Action** states what operation happens on the data or external memory and **wrb_Action** states what happens with registers. Instructions LW, LH, LB and derived will use both, as you are loading from memory and saving to a register. Otherwise one of the actions is always set to no operation type. **exeq_result** is the result from execution unit. **data_rD** is populated for memory load operations to be written to a register, again for instructions LW, LH, LB and derived. When writing to a memory, **byte_Enable** is used for whether we are writing a word, half-word or a byte. With writeback to a register, **wrix_rD** states which register is to be written to.

- **WRB_Type:** Writeback type, interconnect between memory and general purpose register file:

```
TYPE WRB_Type IS RECORD
        wrb_Action  : WRB_ACTION_Type;
        wrix_rD     : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
        data_rD     : STD_LOGIC_VECTOR (31 DOWNTO 0);
END RECORD;
```

wrb_Action is writeback operation, **wrix_rD** is register to be written to and **data_rD** is the actual data to write.

- **HAZARD_WRB_Type:** Hazard writeback type, which execution forwards to itself, with help of memory and writeback component:

```
TYPE HAZARD_WRB_Type IS RECORD
        hazard  : STD_LOGIC;
        save_rX : SAVE_REG_Type;
        data_rX : STD_LOGIC_VECTOR (31 DOWNTO 0);
        data_rD : STD_LOGIC_VECTOR (31 DOWNTO 0);
END RECORD;
```

hazard means there is a hazard situation, as resultant of the previous instruction is going to be used in the current instruction. This is set by execution by checking what memory and writeback is doing. **save_rX** states with operand is saved, either `Ra` or `Rb`, otherwise none if no hazard occurs. Now two separate data storages: **data_rX** for hazards over `Ra` or `Rb` and **data_rD** for hazard over `Rd`. Example of a hazard over `Rd`:

```
ADDI    Rd, Ra, Imm
SWI     Rd, Ra, Imm
```

Here, `Rd` value is calculated in previous instruction and immediately after used as an address to memory.

- **MEM_REG_Type:** Subset of **EX2MEM_Type**, holds the necessary information by one cycle, when we cannot writeback right away:

```
TYPE MEM_REG_Type IS RECORD
        wrb_Action   : WRB_ACTION_Type;
        exeq_result  : STD_LOGIC_VECTOR (31 DOWNTO 0);
        byte_Enable  : STD_LOGIC_VECTOR ( 3 DOWNTO 0);
        wrix_rD      : STD_LOGIC_VECTOR ( 4 DOWNTO 0);
END RECORD;
```

See **EX2MEM_Type** for explanation of the members.

- **MEM2CTRL_Type:** Memory and writeback interconnect with core component:

```
TYPE MEM2CTRL_Type IS RECORD
        clken : STD_LOGIC;
        int   : STD_LOGIC;
END RECORD;
```

clken states whether core component should enable clock in the next cycle or is held by slower memory operations. **int** is an interrupt request from external memory interface.

- **CORE2DMEMB_Type:** External memory interface type, for transaction from core to peripherals:

```
TYPE CORE2DMEMB_Type IS RECORD
        rd    : STD_LOGIC;
        addr  : STD_LOGIC_VECTOR (14 DOWNTO 0);
        bls   : STD_LOGIC_VECTOR ( 3 DOWNTO 0);
        data  : STD_LOGIC_VECTOR (31 DOWNTO 0);
END RECORD;
```

Matches **LPC1788** interface, address was limited to 64 kiB. **rd** is asserted for read operation, **bls** is asserted for write operation (per byte). **addr** and **data** are the address and the data.

- **DMEMB2CORE_Type:** External memory interface type, transaction from peripherals back to the core and interrupt support:

```
TYPE DMEMB2CORE_Type IS RECORD
        clken : STD_LOGIC;
        data  : STD_LOGIC_VECTOR (31 DOWNTO 0);
        int   : STD_LOGIC;
END RECORD;
```

clken signal is used for slow memory transactions, in our case it's only used when master CPU is actually using the bus. Interrupt support for peripherals is not used (**int** signal, unrelated to memory transaction). **data** is loaded with the actual data during read transaction and unused otherwise.

5.3 Entities

Entities for the pipeline stages are documented in the form of input ports, output ports, their types and with description of their operation. Three common generics:

- **USE_HW_MUL_g:** Whether to add support for hardware multiplier instructions.
- **USE_BARREL_g:** Whether to add support for barrel shifting instructions.
- **COMPATIBILITY_MODE_g:** Whether to be compatible with **MicroBlaze** code and not use any enhancements made (such as conditional execution).
- **IMEM_ABITS_g:** Instruction memory bus size for 32-bit words
- **DMEM_ABITS_g:** Data memory bus size for 32-bit words

5.3.1 Instruction Fetch

Input ports:

```
prog_cntr_i  :  IN STD_LOGIC_VECTOR (31 DOWNTO 0)
inc_pc_i     :  IN STD_LOGIC
EX2IF_i      :  IN EX2IF_Type
```

Output ports:

```
IF2ID_o        : OUT IF2ID_Type
```

Increments program counter each cycle, driven by **inc_pc_i**. If requested, branches to a new address. Program counter within **IF2ID_o** is to be wired as a address to an instruction memory, which is enabled and set to read operation permanently when the core is not halted.

5.3.2 Instruction Decode

Generics:

```
USE_HW_MUL_g : BOOLEAN := TRUE
USE_BARREL_g : BOOLEAN := TRUE
COMPATIBILITY_MODE_g : BOOLEAN := FALSE
```

Input ports:

```
IF2ID_i        :  IN IF2ID_Type
imem_data_i    :  IN STD_LOGIC_VECTOR (31 DOWNTO 0)
INT_CTRL_i     :  IN INT_CTRL_Type
```

Output ports:

```
ID2GPRF_o      : OUT ID2GPRF_Type
ID2EX_o        : OUT ID2EX_Type
ID2CTRL_o      : OUT ID2CTRL_Type
```

Decodes instruction binary into ALU action, source operands, carry bit source, MSR action etc., all are part of **ID2EX_o** interconnect. Additionally sets reading addresses on general purpose register file and informs core component about coming delay slot bit or if we are leaving interrupt service routine using RTI instruction.

5.3.3 Execution

Generics:

```
USE_HW_MUL_g : BOOLEAN := TRUE
USE_BARREL_g : BOOLEAN := TRUE
COMPATIBILITY_MODE_g : BOOLEAN := FALSE
```

Input ports:

```
IF2ID_i          :  IN IF2ID_Type
ID2EX_i          :  IN ID2EX_Type
delayBit_i       :  IN STD_LOGIC
GPRF2EX_i        :  IN GPRF2EX_Type
EX_WRB_i         :  IN WRB_Type
MEM_WRB_i        :  IN WRB_Type
HAZARD_WRB_i     :  IN HAZARD_WRB_Type
IMM_LOCK_i       :  IN IMM_LOCK_Type
MSR_i            :  IN MSR_Type
```

Output ports:

```
EX2IF_o          : OUT  EX2IF_Type
EX2CTRL_o        : OUT  EX2CTRL_Type
HALT_o           : OUT  HALT_Type
EX_WRB_o         : OUT  WRB_Type
HAZARD_WRB_o     : OUT  HAZARD_WRB_Type;
IMM_LOCK_o       : OUT  IMM_LOCK_Type
MSR_o            : OUT  MSR_Type
EX2MEM_o         : OUT  EX2MEM_Type
```

The main stage of the pipeline, doing the actual arithmetic and logic operation. It is flushed during conditional execution, in case the current instruction is supposed to be skipped. Passes memory and writeback information to following stage. **IMM_LOCK_i** and **HAZARD_WRB_i** are directly fed back from the outputs once cycle ago, because they are valid in the next instruction.

5.3.4 Memory And Writeback

Input ports:

```
EX2MEM_i     :  IN  EX2MEM_Type
DMEMB_i      :  IN  DMEMB2CORE_Type
MEM_REG_i    :  IN  MEM_REG_Type
```

Output ports:

```
DMEMB_o      : OUT  CORE2DMEMB_Type
MEM_REG_o    : OUT  MEM_REG_Type
MEM_WRB_o    : OUT  WRB_Type
MEM2CTRL_o   : OUT  MEM2CTRL_Type
```

Memory and writeback is the last stage in the pipeline. It is responsible to write and read data from memory and write back to registers. In some cases it can be seen as 2 stages in the pipeline, because in case of loading data from memory and saving it to a register takes two cycles. However any other operation takes just 1 cycle (excluding waiting for external memory interface to be ready). Wires back two important signals to core component in **MEM2CTRL_o**: interrupt request from a peripheral and clock enabling signal: while it's only intended for slow memory transactions it can be used to halt the core at any time.

5.3.5 Core Component

Generics:

```
IMEM_ABITS_g            :  positive := 9
COMPATIBILITY_MODE_g    :  BOOLEAN := FALSE
```

Input ports:

```
clk_i           :  IN  STD_LOGIC
rst_i           :  IN  STD_LOGIC
halt_i          :  IN  STD_LOGIC
int_i           :  IN  STD_LOGIC
trace_i         :  IN  STD_LOGIC
trace_kick_i    :  IN  STD_LOGIC
```

```
IF2ID_REG_i      :  IN  IF2ID_Type
ID2EX_REG_i      :  IN  ID2EX_Type
EX2IF_REG_i      :  IN  EX2IF_Type
EX2CTRL_REG_i    :  IN  EX2CTRL_Type
exeq_halt_i      :  IN  STD_LOGIC
EX2MEM_REG_i     :  IN  EX2MEM_Type
MEM_REG_i        :  IN  MEM_REG_Type
ID2CTRL_i        :  IN  ID2CTRL_Type
EX_WRB_i         :  IN  WRB_Type
HAZARD_WRB_i     :  IN  HAZARD_WRB_Type
IMM_LOCK_i       :  IN  IMM_LOCK_Type
MSR_i            :  IN  MSR_Type
MEM2CTRL_i       :  IN  MEM2CTRL_Type
```

Output ports:

```
core_clken_o     : OUT STD_LOGIC
imem_addr_o      : OUT STD_LOGIC_VECTOR ((IMEM_ABITS_g-1) DOWNTO 0)
imem_clken_o     : OUT STD_LOGIC
pc_ctrl_o        : OUT STD_LOGIC
IF2ID_REG_o      : OUT IF2ID_Type
ID2EX_REG_o      : OUT ID2EX_Type
delay_bit_o      : OUT STD_LOGIC
gprf_clken_o     : OUT STD_LOGIC
EX2IF_REG_o      : OUT EX2IF_Type
EX2MEM_REG_o     : OUT EX2MEM_Type
MEM_REG_o        : OUT MEM_REG_Type
INT_CTRL_o       : OUT INT_CTRL_Type
EX_WRB_o         : OUT WRB_Type
HAZARD_WRB_o     : OUT HAZARD_WRB_Type
IMM_LOCK_o       : OUT IMM_LOCK_Type
MSR_o            : OUT MSR_Type
```

Core component holds state variables and is the only component in the pipeline that works with clock signals and feeds back previous states of the pipeline as they request it (such as when dealing with hazards). It stores special machine status register. From the control signals, **exeq_halt_i** is asserted when execution stage is executing `HALT` instruction, causing the core to disable clock for the pipeline. It can be resumed by externally asserting **trace_kick_i**. Similarly asserting **trace_i** will cause the core to enable clock only when **trace_kick_i** is asserted. **halt_i** is seen as external halt and when asserted the core will disable clock regardless of other signals. Asserting **int_i** will cause an interrupt, when it's enabled in machine status register.

5.4 Platform Entities

The following entities are specific to a platform.

5.4.1 General Purpose Registers File

Input ports:

```
clk_i        :  in  std_logic
rst_i        :  in  std_logic
clken_i      :  in  std_logic
ID2GPRF_i    :  in  ID2GPRF_Type
MEM_WRB_i    :  in  WRB_Type
```

Output ports:

```
GPRF2EX_o    :  out  GPRF2EX_Type
```

General purpose registers file is stores registers and is responsible for setting up R0 to zero upon reset and dealing with hazardous situation for subsequent register usage within 2 cycles, which is solved by forwarding the data from the write port. Its clock input is controlled by core component.

5.5 Top Module

Generics:

```
IMEM_ABITS_g           :  positive := 9
DMEM_ABITS_g           :  positive := 10
USE_HW_MUL_g           :  boolean := true
USE_BARREL_g           :  boolean := true
COMPATIBILITY_MODE_g   :  boolean := false
```

Input ports:

```
clk_i          :  in  std_logic
rst_i          :  in  std_logic
halt_i         :  in  std_logic
int_i          :  in  std_logic
trace_i        :  in  std_logic
trace_kick_i   :  in  std_logic
imem_clk_i     :  in  std_logic
imem_en_i      :  in  std_logic
imem_we_i      :  in  std_logic_vector(3 downto 0)
imem_addr_i    :  in  std_logic_vector(8 downto 0)
imem_data_i    :  in  std_logic_vector(31 downto 0)
dmem_clk_i     :  in  std_logic
dmem_en_i      :  in  std_logic
dmem_we_i      :  in  std_logic_vector(3 downto 0)
dmem_addr_i    :  in  std_logic_vector(9 downto 0)
dmem_data_i    :  in  std_logic_vector(31 downto 0)
xmemb_i        :  in  DMEMB2CORE_Type
```

Output ports:

```
pc_o           :  out  std_logic_vector(31 downto 0)
```

```
halted_o      : out std_logic
halt_code_o   : out std_logic_vector(4 downto 0)
imem_data_o   : out std_logic_vector(31 downto 0)
dmem_data_o   : out std_logic_vector(31 downto 0)
xmemb_sel_o   : out std_logic
xmemb_o       : out CORE2DMEMB_Type
```

The top module. Wires all stages of the pipeline with the core component and adds general purpose register file and instruction and data memory. See core component in subsetion 5.3.5 for explanation for **halt_i**, **int_i**, **trace_i** and **trace_kick_i**. Additionally access for instruction and data memory map is provided along with a few state variables: **pc_o** for current program counter in execution stage, **halted_o** and **halt_code_o** to see if the core was halted with `HALT` instruction. External memory interface interconnect is provided to wire peripherals (this is done in the top module of the whole design). See appendix D on how to access the controlling signals and state variables from master CPU.

Chapter 6
Other FPGA Peripherals

This chapter describes implementation of other FPGA peripherals, namely **IRC co-processor** and **LX Master**.

6.1 IRC Co-processor

IRC co-processor handles IRC inputs for all axes and saves up to 8% chip space with 4 IRCs (which is about 400 LUTs) contrary to using 32-bit quad counters for every axis. It is divided into three parts:

- **Quad counter:** Low-level entity directly handling IRC inputs. Increments and decrements the IRC count based on decoding **IRC A** and **IRC B** signals and works with the lowest 8 bits of the count. Stores also lowest 8 bits of the count for index when **IDX** is asserted. There is one quad counter per axis.
- **Instruction fetch & decoder:** The co-processor uses primitive instructions, which are decoded into operation type and axis number. Instructions are generated by incrementing a counter and resetting it back to 0 in the last step.
- **Execution:** Reads current IRC count from memory, does the operation from section 3.3 and writes it back to memory. Each step has its own instruction, divided into two types of operations and two stages of each operation.

Block diagram is on figure 6.1.

6.1.1 Operation

The co-processor services one IRC input in four cycles, with one cycle used for the calculation of the IRC count, another one for calculation of the index, one more for setup for the next axis and one is used for idling. From section 3.3 we know there is the same operation form calculating both the current IRC count and the index. There is one difference in count and index calculation which is whether index event has actually occurred (**IDX** was asserted in the meantime). The counts and indexes are sequentially stored in a no-change BRAM addressed by instruction. This is an overview of the operation, the instruction is incremented by one during each step:

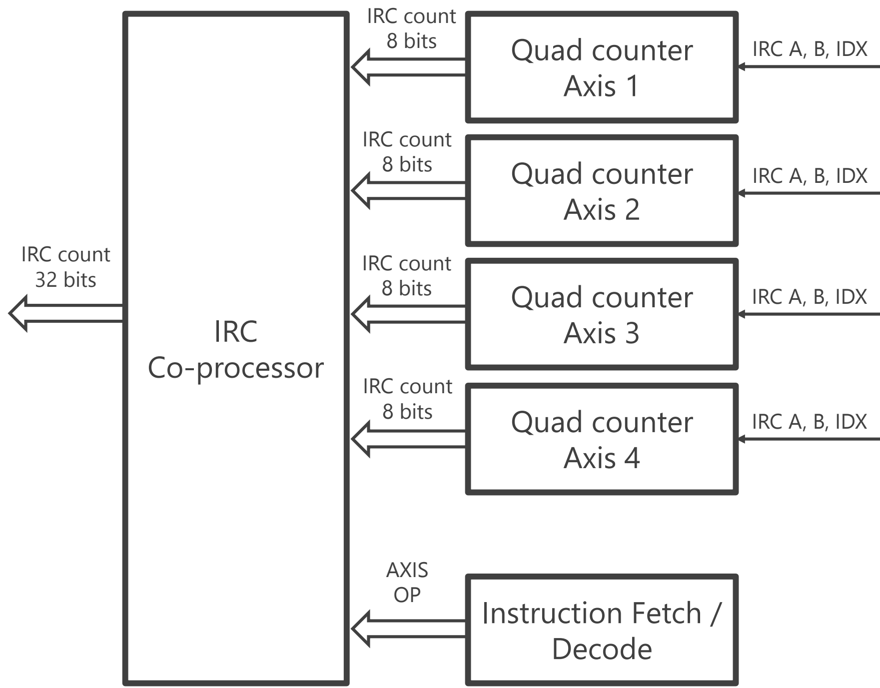

Figure 6.1. IRC handling block diagram

- **Reset:** Set BRAM address to **0b00 00** and enable reading, so that data is ready for first instruction.
- **Instruction 0b00 00:** Current Q is ready on the BRAM output port, from address **0b00 00**. Calculate new Q as described in section 3.3 and write it back on **0b00 00**.
- **Instruction 0b00 01:** Current Q is still on the BRAM output port. Check if there is index event, if so, calculate new Q for the index with C being the 8 bit count for the index event. Reset the index event and write it to address **0b00 01**.
- **Instruction 0b00 10:** Schedule reading Q for the next axis on address **0b00 02**.
- **Instruction 0b00 11:** No operation, reserved.

Next instruction **0b01 00** would do the same as instruction **0b00 00** but for the next axis and starting with address **0b01 00**. So the instruction encoding is that bits 1-0 determine the operation and the upper bits determine the axis (bits 3 - 2 with 4 axes).

6.2 LX Master

LX Master implements master side of the power stage module bus, described in section 3.4. It periodically sends one or more messages stored in its buffer. Double buffering is supported to allow work with the buffers without having to interrupt potentially ongoing transmission. The buffers are stored in a dual-port BRAM, as 16-bit words. There are 256

words per buffer, giving 512 words with two buffers and thus 9-bit address width. The first bit of the address is reserved for the active buffer. The messages are stored as plain data, preceded by message length, which is either at the beginning of the buffer for the first message or right after the last data of the previous message. Addresses **0x000** and **0x100** act as read-write registers and are explained table 6.1:

Address	Description
0x000	Bit 15: Determines which buffer is used, write 0 to use the first buffer address space 0x0xx or 1 to use address space 1x0xx Bits 14:8: Reserved Bits 7:0: Length of the first message in the first buffer in words, write 0 for no messages to transmit
0x100	Bits 15:8: Reserved Bits 7:0: Length of the first message in the second buffer in words, write 0 for no messages to transmit

Table 6.1. LX Master registers

Addresses **0x000** or **0x100** are followed by data of the first message. Another register stored after the data of the first message specifies the address of the next message and its length. In the register, bits 15:8 are the lower 8 bits of the address of the next message (highest bit is set by active buffer) and bits 7:0 are the length of that message. End of transmission is assumed when the length of the message is 0. An example transmission of two messages is shown on table 6.2.

Address	Data	Description
0x000	0x0004	First buffer, 4 words in the first message
0x001	0xC0CC	Data
0x002	0xC199	Data
0x003	0xC266	Data
0x004	0x0333	Data
0x005	0x1002	Next message is on address 0x010 and has 2 words
0x010	0x83F0	Data
0x011	0x3525	Data
0x012	0x0000	Last message, stop transmission

Table 6.2. LX Master sample messages

6.2.1 Operation

LX Master works as a state machine, described below. Each state switches to the one below it unless stated otherwise:

- **ST_BEGIN**: Initial state. Sets up reading register at **0x000** from BRAM to find out which buffer to use. Takes 1 cycle.
- **ST_DECIDE**: Based on register at **0x000** sets up reading either register at **0x100** from BRAM or does nothing. Takes 1 cycle.

- **ST_PREINIT**: Increments address by one to schedule reading of the first data from BRAM . Takes 1 cycle.
- **ST_INIT**: Sets up message length, as we just got it even if we were reading it from **0x100**. Takes 1 cycle.
- **ST_READY**: First data is available, and prepared to be sent to the bus in the coming cycle, alongside with assertion of **SYNC**. Takes 1 cycle.
- **ST_XFER**: Transfers the data. When needed, advances to next words of the message, until it reaches the last word. Takes $16n$ cycles, where n is message length in words.
- **ST_CRC**: Transfers 8-bit CRC, **SYNC** remains asserted. During transmission, reads the word behind the last data and parses it. If there is another message to be sent, loads the first word from BRAM and goes again to state **ST_READY**, releasing **SYNC** for one cycle. Otherwise goes to **ST_END** and releases **SYNC**.
- **ST_END**: Inactive state, waits till the end of the period per iteration.

Chapter 7
FPGA Simulation

With any FPGA design it is very important to simulate the behavior of the cores. It is many times the only way to actually see the state of the signals inside the cores and with larger cores much faster way for their basic validation. It doesn't include any routing and timing, therefore it is still possible to successfully simulate a non-synthesizable core. This can sometimes turn out as an advantage, as you can make a preliminary check if it's actually desirable to "fix" the core into synthesizable state (i.e. passing timing checks) or whether that would be a wasted effort. Choosing a good FPGA simulator is critical; aside from performance of the simulator, it's necessary to make sure it will correctly simulate primitives on the chosen platform (such as PLLs). For our project, we chose ModelSIM from Mentor Graphics. Interface is show on figure 7.1.

7.1 ModelSIM Setup

ModelSIM only needs to have the primitives simulation set up in order to simulate Xilinx FPGA. There are two ways of simulating Xilinx primitives:

- **Use inferred code:** You can create an inferred core for the primitive, such as dual-port block RAM (any type, read-first, write-first, no-change), as a template and use that in your cores. This way synthesizer will optimize the inferred code into a Xilinx primitive – the dual-port block RAM – and the simulator will optimize it into its own primitive. In both ways you have secured matching logic of both optimized primitives as they have to match the VHDL or Verilog descriptor. This is the solution with the best performance and doesn't require any specific platform setup.

- **Use compiled primitives:** In some cases, such as PLL core or other DCM primitives, you are not able to simulate them with inferred design of your own (it would be too complicated, such as simulating reset of PLLs). For this case, follow ModelSIM manual [21] on how to compile the primitives from Xilinx ISE.

Other than that, you of course need to have a test bench module, generally at least with a driven clock signal and stimulus process to reset the core. Then compile all cores in ModelSIM (packages go first in the compile order). It is generally better to use a ModelSIM script for the actual reset than a stimulus process, as that simulates what will happen with the real hardware.

7.2 Tumbl Simulations

All simulations of low-level parts, such as simulating conditional execution instructions, were done in ModelSIM by checking every related signal in the waveform. In the subsections we present some of the more important simulations in detail.

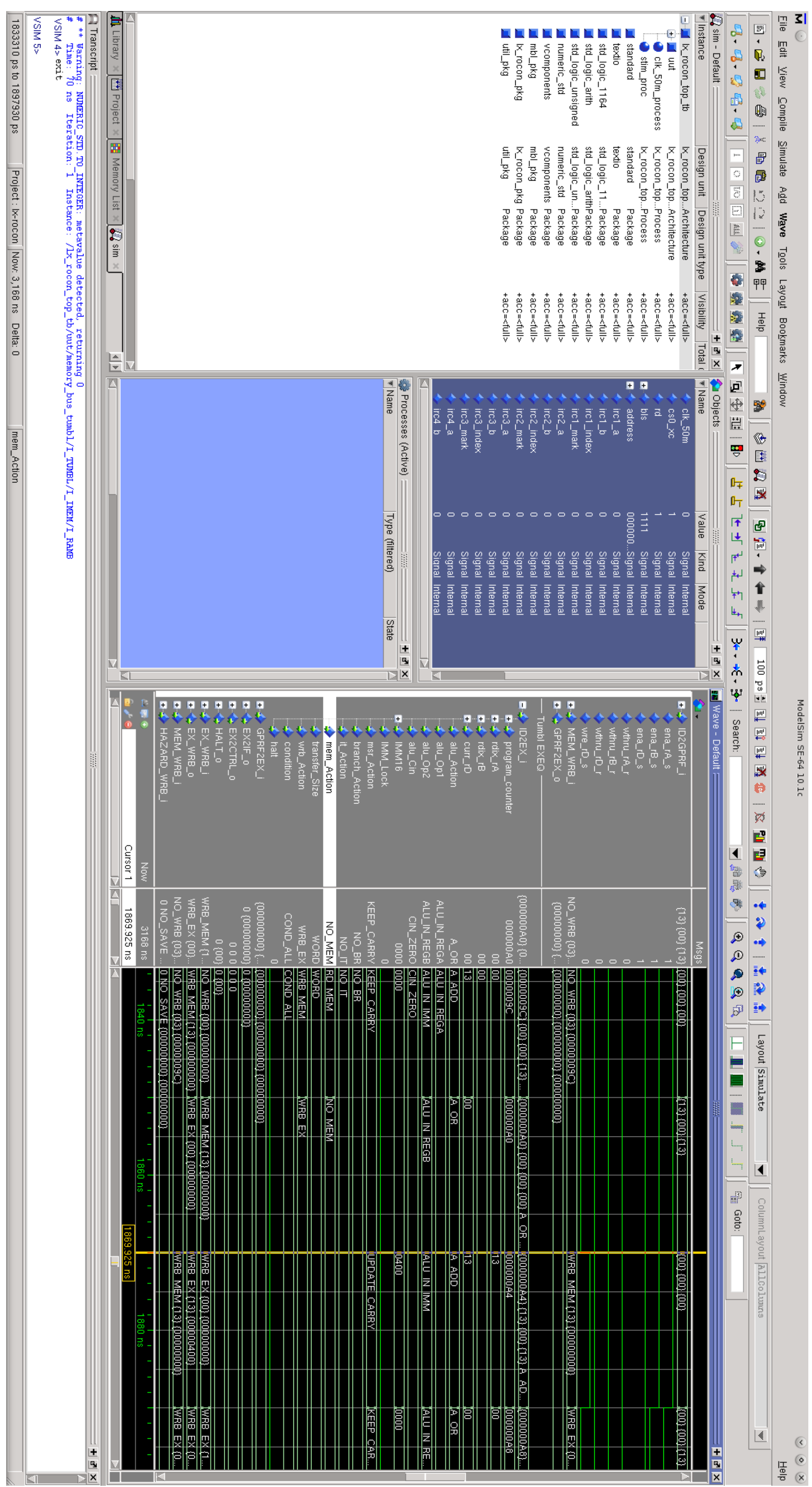

Figure 7.1. ModelSIM

7.2.1 Processor Core

First and the most testing was done for Tumbl, by checking every instruction working correctly and the hazard states are being properly handled, alongside with all custom instructions. To achieve that, we wrote a small utility to convert assembly to **mem** file format accepted by ModelSIM. The code also needs a linker script:

```
ENTRY(_main)

MEMORY
{
        imem (x)     : ORIGIN = 0x00000000, LENGTH = 2k
        dmem (ar!x)  : ORIGIN = 0x00000000, LENGTH = 4k
}

SECTIONS
{
        . = 0x00000000;
        . = ALIGN(4);
        .text : { *(.text) *(.text.*) } > imem

        . = 0x00000000;
        . = ALIGN(4);
        _sdata = . ;
        .data : { *(.data) *(.data.*) } > dmem

        . = ALIGN(4);
        .bss : { *(.bss) *(.bss.*) } > dmem
        _edata = . ;
}
```

Many instructions were tested using compiled common C code, such as addition, changing a value in memory etc.. New instructions not supported by **gcc** were tested using test benches shown in chapter 4 as examples.

7.2.2 Cycle Counting

From chapter 3 we know we need to service each PMSM motor in 625 cycles (for 4 motors) or 310 cycles (for 8 motors). We have created a skeleton of a PID controller (a voltage driven controller with 3 PIDs) and ran it through the simulator. In the code there is a variable that increments by one with every iteration, which was traced in ModelSIM wave form. Use **objdump** to create a disassembly of the code:

```
1e4:    e880002c        lwi     r4, r0, 44
1e8:    30840001        addik   r4, r4, 1
1ec:    f880002c        swi     r4, r0, 44
```

Then find the program counter in ModelSIM wave form. This happens on 4610 ns as shown on figure 7.2 and next time on 5990 ns as shown on figure 7.3. Subtracting them we get 1380 ns, which divided by 20 ns (period of frequency is 50 MHz) results in 69 cycles

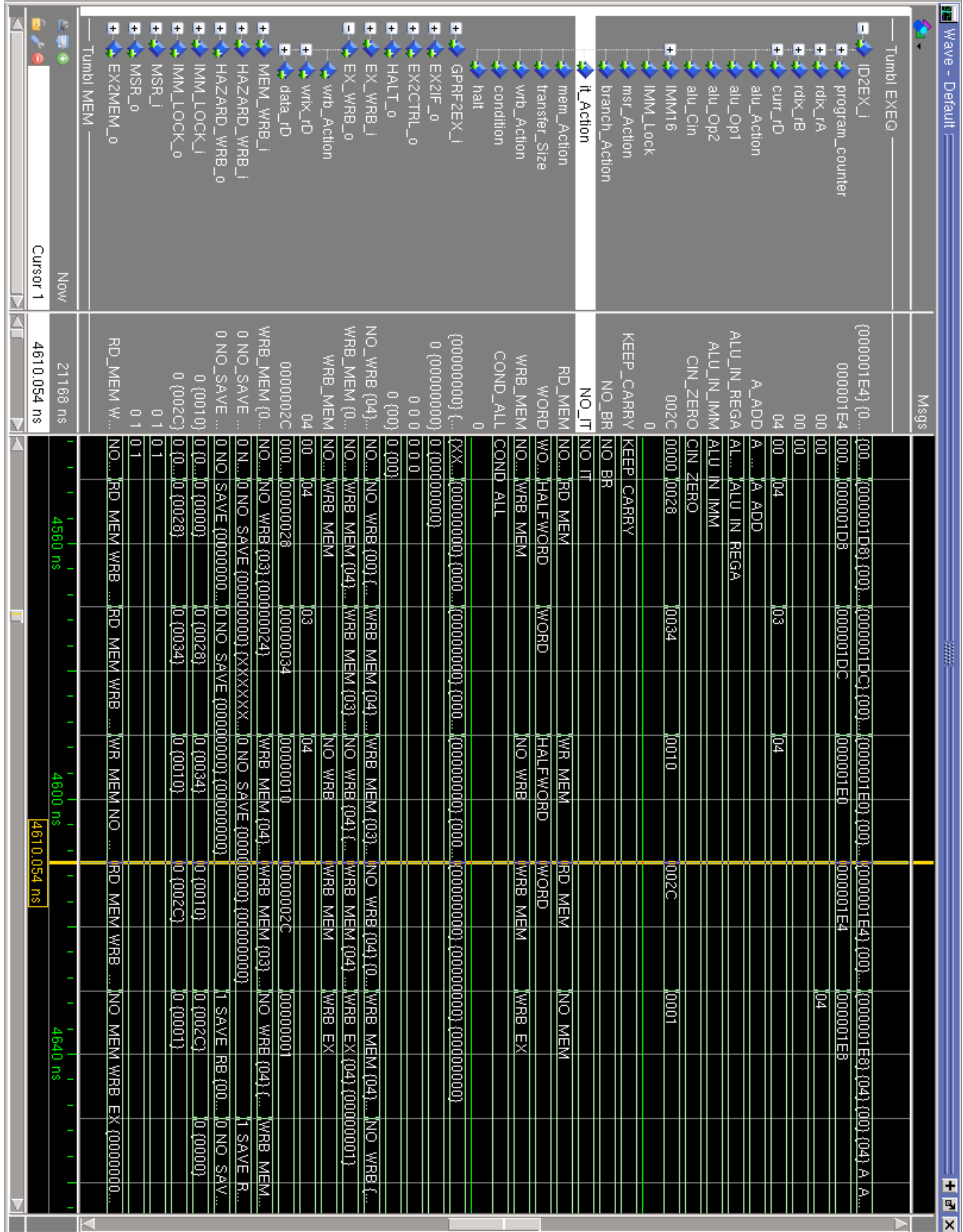

Figure 7.2. The first occurrence of the increment

(including the increment, which needs 4 cycles), so clean result is 65 cycles. This satisfies our requirements well enough and leaves securely enough instructions even for 8 PMSM motors (over 200 cycles), that we didn't conduct further timing tests as the controller developed from voltage driven controller to current driven controller.. Note that this code wasn't optimized as well.

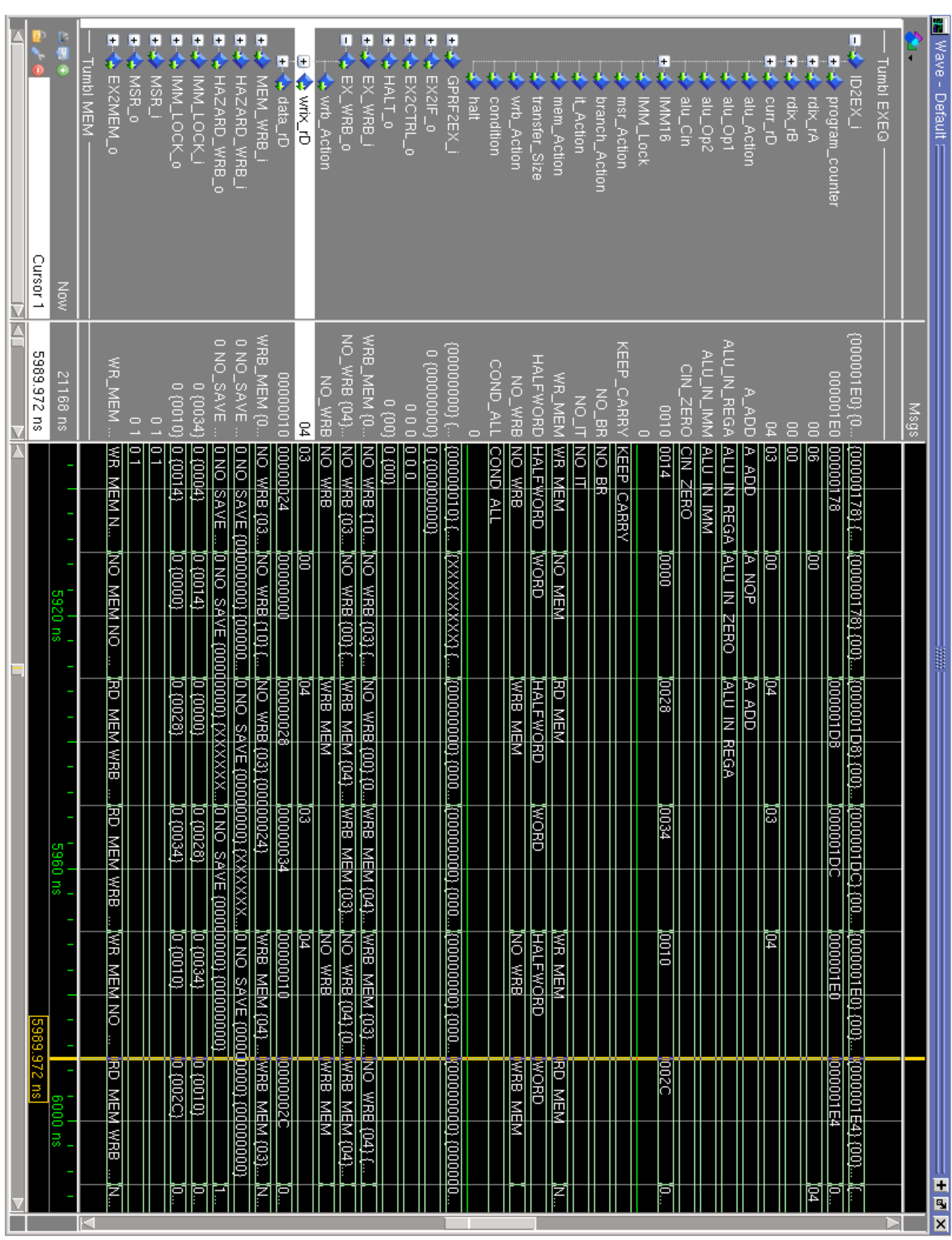

Figure 7.3. The second occurrence of the increment

7.2.3 Tumbl External Memory Interface Collisions

Another major test was checking collision between master CPU and Tumbl on external memory interface. We ran a simple code on Tumbl:

```
cyc:
        LWI   R10, R0, 0x2800
        ADDI  R10, R10, 1
        SWI   R10, R0, 0x2800
        BRI   cyc
```

This accesses **0x2800** address, mapped on the external memory. Concurrently to that, periodically run reading from master CPU using a ModelSIM script simulating a write operation:

```
force -freeze -cancel 112ns cs0_xc 0
force -freeze -cancel 112ns bls 0000
force -freeze -cancel 112ns address $1
force -freeze -cancel 112ns data $2
run 168ns
```

Parameter address is anything wired to Tumbl external memory space, such as **16#8000**, and with parameter data is random each iteration. Eventually these two will collide (if they happen to have the same periods then just add extra waiting on master CPU side). When they collide, check if **clken** is not asserted during master CPU write cycke and that during that address and data are set accordingly to what to be is written from master CPU side. The next cycle after that, make sure that **clken** is asserted back and that address and data are restored to what is read or written on Tumbl side.

7.3 IRC Co-processor

IRC co-processor basics can be checked by watching the loop and toggling their **A** and **B** signals to see if it updates the count correctly alongside with writing it to BRAM. In the simulator, we mainly check if error state is and is reset properly, by toggling **A** and **B** at once and then simulating the write transaction for the reset. When this is done, it's perhaps easiest to see that it works on a real hardware. Plug in a PMSM motor to an IRC input and rotate it manually, while monitoring IRC changes. Check if **IDX** works by rotating the motor back and forth around the place where it's asserted. It's a little harder to see the current IRC value is absolutely correct, because that needs at least hundreds of rotations, this is perhaps best checked back in the simulator by simulating a movement by n pulses in one direction and then seeing it the IRC counter incremented or decremented by n.

7.4 LX Master Transmission

LX Master transmission was also one of the more intense tests. We loaded the following streams into its BRAM. First check is a transmission of a single message in **mem** file format:

```
@0 0008 ; 1st buffer, length 8
@1 c0cc
```

```
@2 c199
@3 c266
@4 0333
@5 83f0
@6 0000
@7 0000
@8 0000
@9 0000 ; end
```

Then a transmission of two messages:

```
@0 0008 ; 1st buffer, length 8
@1 c0cc
@2 c199
@3 c266
@4 0333
@5 83f0
@6 0000
@7 0000
@8 0000
@9 0a02 ; 2nd stream on @a, length 2
@a 1111
@b 3525
@c 0000 ; end
```

Then a check if the second buffer works:

```
@0   8000 ; 2nd buffer
@100 0000 ; length 0
```

Lastly a transmission from the second buffer:

```
@0   8000 ; 2st buffer,
@100 0008 ; length 8
@101 c0cc
@102 c199
@103 c266
@104 0333
@105 83f0
@106 0000
@107 0000
@108 0000
@109 0a02 ; 2nd stream on @10a, length 2
@10a 1111
@10b 3525
@10c 0000 ; end
```

Keypoints of the testing are:

- **Initialization:** A check that initialization works, correct buffer is selected and the component is initialized to transmit the data.
- **CRC:** A check that CRC is properly reset, generated and transmitted after the last bit.

- **Next stream:** A check that the core finds and correctly selects the next stream or stops transmitting if it is at the end.
- **SYNC signal assertion:** A check that **SYNC** signal is not asserted at least for a cycle when starting to transmit a new stream.

7.5 Master CPU Online Debugging

Certain things, such as PMSM motor itself, are too hard to simulate. This means there is a need of online debugging and the main reason why everything Tumbl can control is still accessible to master CPU, completely omitting the need of JTAG interface for Tumbl. Instruction and data memories are accessible permanently without blocking Tumbl and can be monitored. Since external memory interface may block Tumbl, best is to find out when Tumbl is idling (which is going to happen for a long time with just a single motor), such as creating a register in Tumbl data memory on a specified address and checking when it was changed from "active" to "inactive" mode and then dumping LX Master buffers.

Chapter 8
Conclusion

In this thesis we have selected a combined CPU / FPGA budget platform as ARM Cortex-M3/4F CPU with Spartan-6 FPGA. We designed FPGA peripherals necessary for creating a current driven PMSM controller, supporting up to 4 PMSM motors, with possible later extension to 8 PMSM motor. Slave memory controller inside Spartan-6 was designed for memory transactions between master CPU, including a peripheral to measure the required delays. It was determined that the best way to control PMSM motors is to use synthesized processor as a co-processor due to resource usage. After a research of available synthesized processors we chose to start off with **MB-Lite+**. It was further modified, in order to correct errors in the architecture, change its role to a co-processor and to minimize the core by removing unneeded peripherals. Then we found out some ways how to improve performance, which resulted in conditional execution. And then we verified it has enough performance to satisfy the task to control PMSM motors using simulations. The core is named Tumbl. Given the changes in instruction set, we patched **binutils**, **gcc** and **newlib** to maintain C language and assembler support. Further research was made for division, which is generally costly on both cycles and chip space, where at this point we chose to use a lookup table to calculate an inverse number of the divider and multiply with it. We have also designed an IRC co-processor to save chip space when handling multiple IRCs (saves up to 400 LUTs for 4 IRCs). Finally a communication peripheral with power stage board was designed, albeit only the transmitting side, given that the protocol of the bus is still in development. The other cores are ready for productization.

For this project, 50 MHz was assumed as a minimum frequency on which Tumbl co-processor has to run. Further improvements to the core and for it to run on higher frequency would be to modify execution stage to be multi-cycle, to relax the critical path on barrel shifter and multiplier. Additionally a few improvements in the instruction sets could be made:

- **Removal of `IMM` instruction:** `IMM` instruction can be fully substituted by loading from memory generally at no performance cost (requires optimization in gcc, otherwise costs one extra cycle). Simplifies the core and doesn't remove any feature.
- **Use of 21-31 bits:** Instructions with `Rb` operand may use the remaining 10 bits for small arithmetic operations or shifting.
- **Hardware division:** If multi-cycle execution unit is implemented, see possibilities of division implementation.
- **Shared instruction and data memory:** This is rather experimental. Merge instruction and data memory into one 64-bit memory. Instruction fetch would read 2 words at once leaving one cycle for data access. A store buffer is needed to solve collisions between program counter leads and data.

Lastly, Inverse kinematics task for a 6R manipulator, presented in appendix A, can be used in the product for movement planning to allow using the final product for a wider range of

robotic manipulators.

Appendix A

Inverse Kinematics For A General 6R Manipulator

The following report written by **Martin Meloun** and **Tomáš Pajdla**, published as [23], describes an analytic solution of the inverse kinematics task for a general 6R manipulator and its implementation using BLAS and LAPACK libraries. Our approach was to secure and tune algorithm accuracy by minimizing the significance of poor conditioning in matrix operations while keeping the performance on an industrial level. The algorithm involves symbolic preprocessing in multiple steps, then the multivariate problem is reduced through matrix decompositions to a single univariate polynomial which is further reduced to a generalized eigenvalue problem.

CENTER FOR
MACHINE PERCEPTION

CZECH TECHNICAL
UNIVERSITY IN PRAGUE

RESEARCH REPORT

ISSN 1213-2365

Inverse Kinematics for a General 6R Manipulator

Martin Meloun
Tomáš Pajdla

meloumar@cmp.felk.cvut.cz
pajdla@cmp.felk.cvut.cz

CTU–CMP–2013–29

Available at
ftp://cmp.felk.cvut.cz/pub/cmp/articles/meloun/Meloun-TR-2013-29.pdf

This work was supported by PRoViDE-FP7-SPACE-3/2377 and SGS12/191/OHK3/37/13.

Research Reports of CMP, Czech Technical University in Prague, No. 29, 2013

Published by

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

A.1 Introduction

The problem of the inverse kinematics task for a 6R manipulator is that unlike for common 3-2-1 manipulators (a manipulator in which axes of 4th, 5th and 6th joints intersect in a single point) there is no plain and simple solution using just its geometric properties. It still remains a non-trivial task requiring matrix decompositions and numerical optimizations with symbolic construction of the matrices. This report is uses analytic solution first presented by [2] and then later improved by [4] for industrial applications. It was further optimized in [5] by adding numerical analysis. Later on there was further optimization suggested by [6] to reduce the degree of polynomials at the cost of having to do a matrix inversion which however is not acceptable as there is no way to solve that if the matrix to be inverted is poorly conditioned or even singular. This report builds mainly on [5] but uses a different approach in numerical optimization. We chose to optimize the intermediate results from the matrix decompositions rather than perfecting the input matrices in terms of their conditioning. The algorithm was modified to allow such approach.

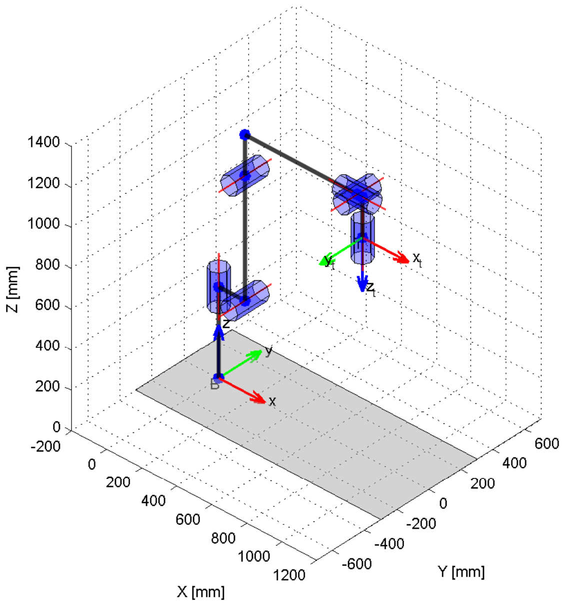

Figure A.1. 6R Manipulator Schematics

A.2 Inverse Kinematics Task

A.2.1 Formulation

Every joint of the robot is described using Denavit-Hartenberg notation[1] as a series of translations and rotations in specific order. The notation is minimizing the number of parameters needed to describe the manipulator kinematics. The transformations in Denavit-Hartenberg notation are in the following order: rotation around z axis by parameter θ, translation along z axis by parameter d, translation along x axis by parameter α and rotation around x axis by parameter a. In a general 6R manipulator, θ is considered to be the control variable. Consider $\mathtt{R_z}$ to be the rotation around z axis, $\mathtt{R_x}$ to be the rotation around x axis, $\mathtt{T_z}$ to be the translation along z axis and $\mathtt{T_x}$ to be the translation along x axis. The set of transformations can be represented by matrices as

$$
\begin{aligned}
\mathtt{M_i} &= \mathtt{R_z}(\theta_i)\mathtt{T_z}(d_i)\mathtt{R_x}(\alpha_i)\mathtt{T_x}(a_i) \\
&= \mathtt{M_{iA}}\mathtt{M_{iB}} \\
&= \begin{pmatrix} c_i & -s_i & 0 & 0 \\ s_i & c_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & a_i \\ 0 & \lambda_i & -\mu_i & 0 \\ 0 & \mu_i & \lambda_i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned} \tag{A.1}
$$

in which $c_i = \cos(\theta_i)$, $s_i = \sin(\theta_i)$, $\lambda_i = \cos(\alpha_i)$ and $\mu_i = \sin(\alpha_i)$. The end-effector pose is described as a transformation towards the base as

$$
\mathtt{M_H} = \begin{pmatrix} l_x & m_x & n_x & r_x \\ l_y & m_y & n_y & r_y \\ l_z & m_z & n_z & r_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.2}
$$

The problem of the general 6R inverse kinematics task is to calculate the control variables θ_1, θ_2, θ_3, θ_4, θ_5 and θ_6 from the equation

$$
\mathtt{M_1 M_2 M_3 M_4 M_5 M_6} = \mathtt{M_H} \tag{A.3}
$$

The left hand side entries of the matrices are functions of the sines and cosines of the control variables. Given the structure of the matrices, last row produces no equations which means there are only 12 equations. Furthermore, since the first 3 columns and first 3 rows represent rotation which is orthonormal, then there is only 6 linearly independent equations for solving 6 variables.

A.2.2 Raghaven and Roth Solution

Raghaven and Roth solution [2] reduces the above multivariate problem into solving a 16-degree polynomial in $\frac{\theta_3}{2}$. They use the property that inversion matrix of $\mathtt{M_i}$ remains linear and reduce the degree of the polynomials in the (A.3) by rearranging the matrices as

$$
\mathtt{M_3 M_4 M_5} = \mathtt{M_2^{-1} M_1^{-1} M_H M_6^{-1}} \tag{A.4}
$$

Furthermore, if we take a look on matrix $\mathtt{M}_6^{-1}$

$$\begin{aligned}\mathtt{M}_6^{-1} &= \mathtt{M}_{6B}^{-1}\mathtt{M}_{6A}^{-1} \\ &= \begin{pmatrix} 1 & 0 & 0 & -a_6 \\ 0 & \lambda_6 & \mu_6 & 0 \\ 0 & -\mu_6 & \lambda_6 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_6 & s_6 & 0 & 0 \\ -s_6 & c_6 & 0 & 0 \\ 0 & 0 & 1 & -d_6 \\ 0 & 0 & 0 & 1 \end{pmatrix}\end{aligned} \tag{A.5}$$

Notice that c_6, s_6 are only in the first two columns of $\mathtt{M}_{6A}^{-1}$. Take the equations only for the last two columns (marked red) to get rid of c_6, s_6 at a cost of 6 equations. The system now looks like

$$\begin{aligned}\mathtt{M}_3(c_3, s_3)\mathtt{M}_4(c_4, s_4) \begin{pmatrix} c_5 & -s_5 & 0 & 0 \\ s_5 & c_5 & 0 & 0 \\ 0 & 0 & 1 & d_5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & a_5 \\ -\mu_5 & 0 \\ \lambda_5 & 0 \\ 0 & 1 \end{pmatrix} = \\ \mathtt{M}_2^{-1}(c_2, s_2)\mathtt{M}_1^{-1}(c_1, s_1)\mathtt{M}_\mathtt{H} \begin{pmatrix} 1 & 0 & 0 & -a_6 \\ 0 & \lambda_6 & \mu_6 & 0 \\ 0 & -\mu_6 & \lambda_6 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & -d_6 \\ 0 & 1 \end{pmatrix}\end{aligned} \tag{A.6}$$

As a result there are 6 equations for 5 variables.

The above reduction has also a geometric interpretation (not mentioned in [2]). Consider the end-effector pose to be described with Cartesian coordinates and Euler angles using a convention in which the last rotation is around z axis (e.g. $z - x' - z''$ convention). The coordinates of the pose in such convention would be

$$H = (x_H, y_H, z_H, \alpha_H, \beta_H, \gamma_H)^\top \tag{A.7}$$

Let's look back on the initial equation (A.3) and expand $\mathtt{M}_6$ from (A.1)

$$\mathtt{M}_1(\theta_1)\mathtt{M}_2(\theta_2)\mathtt{M}_3(\theta_3)\mathtt{M}_4(\theta_4)\mathtt{M}_5(\theta_5)\mathtt{R}_\mathtt{z}(\theta_6)\mathtt{T}_\mathtt{z}(d_6)\mathtt{R}_\mathtt{x}(\alpha_6)\mathtt{T}_\mathtt{x}(a_6) = \mathtt{M}_\mathtt{H} \tag{A.8}$$

and consider a point W defined by transformation matrix $\mathtt{M}_\mathtt{W}$ as right hand side of

$$\mathtt{M}_1(\theta_1)\mathtt{M}_2(\theta_2)\mathtt{M}_3(\theta_3)\mathtt{M}_4(\theta_4)\mathtt{M}_5(\theta_5)\mathtt{R}_\mathtt{z}(\theta_6) = \mathtt{M}_\mathtt{H}\mathtt{T}_\mathtt{x}(-a_6)\mathtt{R}_\mathtt{x}(-\alpha_6)\mathtt{T}_\mathtt{z}(-d_6) \tag{A.9}$$

i.e.

$$\mathtt{M}_\mathtt{W} = \mathtt{M}_\mathtt{H}\mathtt{T}_\mathtt{x}(-a_6)\mathtt{R}_\mathtt{x}(-\alpha_6)\mathtt{T}_\mathtt{z}(-d_6) \tag{A.10}$$

Similarly to point H, point W can be also defined using Cartesian coordinates and Euler angles using the same convention

$$W = (x_W, y_W, z_W, \alpha_W, \beta_W, \gamma_W)^\top \tag{A.11}$$

Notice that in (A.9), $\mathtt{M}_W$ can be calculated from the right hand side and then converted to the coordinates of point W. The left hand side however ends with rotation around z axis,$\mathtt{R}_\mathtt{z}(\theta_6)$. Because we decided to use a convention ending in rotation around z axis, it only changes γ_W. Therefore the first 5 parameters x_W, y_W, z_W, α_W and β_W depend only on the first 5 control variables. This is the geometric interpretation of the step done in (A.6). Notice that for special manipulators (for instance a 3-2-1 manipulator), the reduction can go

even further (for 3-2-1 manipulator only the first three variables, the Cartesian coordinates, depend only on the first three control variables θ_1, θ_2 and θ_3).

Back to (A.6), Raghaven and Roth then simplify the equations by multiplying from the left with $\mathtt{U}_2\mathtt{M}_{2B}$ where

$$\mathtt{U}_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & d_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathtt{M}_{2B} = \begin{pmatrix} 1 & 0 & 0 & a_2 \\ 0 & \lambda_2 & -\mu_2 & 0 \\ 0 & \mu_2 & \lambda_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and label each equation as

$$\begin{aligned} \begin{pmatrix} l_1 & p_1 \\ l_2 & p_2 \\ l_3 & p_3 \\ 0 & 1 \end{pmatrix} &= \mathtt{U}_2\mathtt{M}_{2B}\mathtt{M}_3(c_3,s_3)\mathtt{M}_4(c_4,s_4) \begin{pmatrix} c_5 & -s_5 & 0 & 0 \\ s_5 & c_5 & 0 & 0 \\ 0 & 0 & 1 & d_5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & a_5 \\ -\mu_5 & 0 \\ \lambda_5 & 0 \\ 0 & 1 \end{pmatrix} \\ &= \mathtt{U}_2\mathtt{M}_{2A}^{-1}(c_2,s_2)\mathtt{M}_1^{-1}(c_1,s_1)\mathtt{M}_H \begin{pmatrix} 1 & 0 & 0 & -a_6 \\ 0 & \lambda_6 & \mu_6 & 0 \\ 0 & -\mu_6 & \lambda_6 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & -d_6 \\ 0 & 1 \end{pmatrix} \end{aligned} \tag{A.12}$$

and form two vectors $\mathtt{P} = \begin{pmatrix} p_1 & p_2 & p_3 \end{pmatrix}^\top$ and $\mathtt{L} = \begin{pmatrix} l_1 & l_2 & l_3 \end{pmatrix}^\top$ as a polynomial ideal (the p_1, p_2, p_3 and l_1, l_2, l_3 are both left and right hand sides of the equations). This polynomial ideal has a few interesting properties. The following operations form new equations which are not linear combination of the current equations and yet do not add new monomials:

- $\mathtt{P}^\top\mathtt{P}$ (1 equation)
- $\mathtt{P}^\top\mathtt{L}$ (1 equation)
- $\mathtt{P} \times \mathtt{L}$ (3 equations)
- $\left(\mathtt{P}^\top\mathtt{P}\right)\mathtt{L} - \left(2\mathtt{P}^\top\mathtt{L}\right)\mathtt{P}$ (3 equations)

This adds another 8 equations in total. In these new equations the additional monomials generated by the operations are truncated due to the geometric properties of the original 6 equations. Therefore in total there are 14 equations for 5 variables. The equations can now be written in matrix form as

$$\mathtt{P}_{14\times 9}(c_3,s_3) \begin{pmatrix} s_4s_5 \\ s_4c_5 \\ c_4s_5 \\ c_4c_5 \\ s_4 \\ c_4 \\ s_5 \\ c_5 \\ 1 \end{pmatrix} = \mathtt{N}_{14\times 8} \begin{pmatrix} s_1s_2 \\ s_1c_2 \\ c_1s_2 \\ c_1c_2 \\ s_1 \\ c_1 \\ s_2 \\ c_2 \end{pmatrix} \tag{A.13}$$

or in short as $\mathtt{P}\,p = \mathtt{N}\,n$, with p and n being the vectors of the monomials. All constants were moved to the left hand side, c_3, s_3 were placed inside the matrix $\mathtt{P}$ rather than the vector p and matrix $\mathtt{N}$ contains just plain numbers. Further steps were altered by Manocha and Canny.

A.2.3 Modified Manocha and Canny Optimization

Elimination of c_1, s_1, c_2, s_2

Raghaven and Roth continued further with splitting the matrices into two: one with 8 rows and the other one with 6 rows. The purpose of such step was to form a square sub-matrix of $\mathtt{N}$ and eliminate right hand side variables using it's inversion. Unfortunately in case the sub-matrix of $\mathtt{N}$ was poorly conditioned or even singular, such method would produce no results there.

Manocha and Canny suggest to use SVD decomposition instead to deal with the above problem and secure good conditioning. Matrix $\mathtt{N}$ is decomposed with SVD as

$$\mathtt{P}_{14\times 9}\, p_{9\times 1} = \mathtt{U}_{14\times 14}\Sigma_{14\times 8}\mathtt{V}^{\top}_{8\times 8}\, n_{8\times 1} \tag{A.14}$$

in which $\mathtt{U}$ is a unitary matrix and can be moved to the left hand side

$$\mathtt{U}^{\top}_{14\times 14}\mathtt{P}_{14\times 9}\, p_{9\times 1} = \Sigma_{14\times 8}\mathtt{V}^{\top}_{8\times 8}\, n_{8\times 1} \tag{A.15}$$

and Σ contains singular values on it's diagonal. Thus last $(14 - \mathrm{rank}(\Sigma))\geq 6$ rows contain just zeroes and right hand side of the corresponding equations is equal to zero. Let r be the rank of Σ. The equation above can be written as

$$\begin{pmatrix} \mathtt{U}^{\top}_{\mathtt{a}\,r\times 14}\mathtt{P}_{\mathtt{a}\,r\times 9} \\ \mathtt{Z}_{(14-r)\times 9} \end{pmatrix} p_{9\times 1} = \begin{pmatrix} \Sigma_{r\times 8} \\ \mathtt{0}_{(14-r)\times 8} \end{pmatrix} \mathtt{V}^{\top} n_{8\times 1} \tag{A.16}$$

and so, assuming $\mathtt{N}$ has full rank

$$\mathtt{Z}_{6\times 9}\, p_{9\times 1} = 0 \tag{A.17}$$

and therefore c_1, s_1 and c_2, s_2 have been eliminated. If matrix $\mathtt{N}$ is singular, use just the first 6 rows of matrix $\mathtt{Z}$ (which are the best conditioned).

There is however yet a different approach. Decompose matrix $\mathtt{N}$ with QR algorithm instead

$$\mathtt{P}_{14\times 9}\, p_{9\times 1} = \mathtt{Q}_{14\times 14}\mathtt{R}_{14\times 8}\, n_{8\times 1} \tag{A.18}$$

Matrix $\mathtt{R}$ is triangular and contains the desired property that its 6 bottom rows contain just zeroes. However unlike SVD, QR decomposition is not rank revealing but the rank can be estimated by checking for non-zero elements on the diagonal. For now assume that matrix $\mathtt{R}$ has full rank. Move matrix $\mathtt{Q}$ to the left hand side. Given that matrix $\mathtt{R}$ is triangular, we get

$$\mathtt{Q}^{\top}{}_{14\times 14}\mathtt{P}_{14\times 9}\, p_{9\times 1} = \begin{pmatrix} \mathtt{R}_{8\ 8\times 8} \\ \mathtt{0}_{6\times 8} \end{pmatrix} n_{8\times 1} \tag{A.19}$$

As last 6 rows of matrix $\mathtt{R}$ are just zeroes, the right hand side for the last 6 equations is equal to zero.

$$\begin{pmatrix} \mathtt{U}^{\top}_{\mathtt{a}\,8\times 14}\mathtt{P}_{\mathtt{a}\,8\times 9} \\ \mathtt{Z}_{6\times 9} \end{pmatrix} p_{9\times 1} = \begin{pmatrix} \mathtt{R}_{8\ 8\times 8} \\ \mathtt{0}_{6\times 8} \end{pmatrix} n_{8\times 1} \tag{A.20}$$

and we get (A.17) again.

Should matrix $\mathtt{R}$ be singular (or rather, in cases when it cannot be guaranteed that matrix $\mathtt{R}$ is regular) then it is necessary to decompose it further with SVD

$$\mathtt{R}_{8\ 8\times 8} = \mathtt{U}_{8\ 8\times 8}\Sigma_{8\ 8\times 8}\mathtt{V}_{8\times 8}^{\top} \tag{A.21}$$

Form the $\mathtt{U}$ and Σ matrices

$$\mathtt{U}_{14\times 14} = \begin{pmatrix} \mathtt{U}_{8\ 8\times 8} & \mathtt{0}_{8\times 6} \\ \mathtt{0}_{6\times 8} & \mathtt{I}_{6\times 6} \end{pmatrix} \tag{A.22}$$

$$\Sigma_{14\times 8} = \begin{pmatrix} \Sigma_{8\ 8\times 8} \\ \mathtt{0}_{6\times 8} \end{pmatrix} \tag{A.23}$$

The above is a property of QR decomposition, i.e: for any real matrix $\mathtt{X}$ decomposed by QR, the resulting triangular matrix has the same singular values as the original matrix. Substitute back to (A.19) to get

$$\mathtt{U}_{14\times 14}^{\top}\mathtt{Q}_{14\times 14}^{\top}\mathtt{P}_{14\times 9}\, p_{9\times 1} = \Sigma_{14\times 8}\mathtt{V}_{8\times 8}^{\top}\, n_{8\times 1} \tag{A.24}$$

in which the bottom $(14 - \mathrm{rank}(\Sigma)) > 6$ equations have zero on right hand side. Further steps to obtain matrix $\mathtt{Z}$ are identical to Manocha and Canny approach.

Sylvester Dialytic Elimination Method

To summarize the previous steps, the variables on the right hand side c_1, s_1 and c_2, s_2 were eliminated and there are 6 equations in

$$\mathtt{Z}_{6\times 9}(c_3, s_3) \begin{pmatrix} s_4 s_5 \\ s_4 c_5 \\ c_4 s_5 \\ c_4 c_5 \\ s_4 \\ c_4 \\ s_5 \\ c_5 \\ 1 \end{pmatrix} = 0 \tag{A.25}$$

Notice that the vector of the monomials is always non-zero, regardless of the values of the goniometric functions (because the last row is equal to 1). This means that matrix $\mathtt{Z}$ does not have full rank when there is a solution. To use such property we have to convert the system to a square system which is done in the next two steps. First step involves parametrizing c_i, s_i using tangents to convert two bounded variables into one variable x_i

$$x_i = \tan\left(\frac{\theta_i}{2}\right) \tag{A.26}$$

$$s_i = \frac{2x_i}{1 + {x_i}^2}, \quad c_i = \frac{1 - {x_i}^2}{1 + {x_i}^2} \tag{A.27}$$

Notice that x_i will converge to infinity when $\theta_i = \pi$. Substitute c_4, s_4 with x_4 and multiply all equations with common denominator $(1 + {x_4}^2)$ to get

$$\mathtt{Z}'_{6\times 9}(c_3, s_3) \begin{pmatrix} {x_4}^2 s_5 \\ {x_4}^2 c_5 \\ {x_4}^2 \\ x_4 s_5 \\ x_4 c_5 \\ x_4 \\ c_5 \\ s_5 \\ 1 \end{pmatrix} = 0 \tag{A.28}$$

Second step is to multiply all equations with x_4. This will add 6 new equations but only 3 new monomials

$$\mathtt{Z}''_{12\times 12}(c_3, s_3) \begin{pmatrix} {x_4}^3 s_5 \\ {x_4}^3 c_5 \\ {x_4}^3 \\ {x_4}^2 s_5 \\ {x_4}^2 c_5 \\ {x_4}^2 \\ x_4 s_5 \\ x_4 c_5 \\ x_4 \\ c_5 \\ s_5 \\ 1 \end{pmatrix} = 0 \tag{A.29}$$

Unlike in Raghaven and Roth approach (adopted by Manocha and Canny), do not substitute c_5, s_5 with x_5. From (A.25) we know that $\mathtt{Z}''$ has to be a singular matrix, therefore

$$|\mathtt{Z}''(c_3, s_3)| = 0 \tag{A.30}$$

which is a polynomial in c_3, s_3 with degree of 24. To make it univariate, substitute c_3, s_3 with x_3 and multiply it by common denominator $(1 + {x_3}^2)$ to get a polynomial in one variable x_3.

$$|\mathtt{Z}'''(x_3^2, x_3)| = 0 \tag{A.31}$$

This polynomial is divisible by $(1+x_3^2)^4$ (proof in [2]) and thus 8 solutions are known apriori ($\pm i$, $i = \sqrt{-1}$, both of algebraic multiplicity 4). The polynomial is however too expensive to expand symbolically and Manocha and Canny reduce the problem into solving generalized eigenvalue problem. Write $\mathtt{Z}'''$ as

$$\mathtt{Z}'''(x_3^2, x_3) = \mathtt{A}x_3^2 + \mathtt{B}x_3 + \mathtt{C} \tag{A.32}$$

in which $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ are 12×12 matrices with plain numbers. Consider p'' as the vector of the monomials in (A.29). The equation now looks like

$$(\mathtt{A}x_3^2 + \mathtt{B}x_3 + \mathtt{C})p'' = 0 \tag{A.33}$$

Rearrange it a bit

$$(x_3(\mathtt{A}x_3) + (\mathtt{B}x_3) + \mathtt{C})p'' = 0 \tag{A.34}$$

and then expand p'' as $\begin{pmatrix} p'' \\ x_3 p'' \end{pmatrix}$ and transform the above equation into

$$\left[x_3 \begin{pmatrix} \mathtt{I} & \mathtt{0} \\ \mathtt{0} & \mathtt{A} \end{pmatrix} + \begin{pmatrix} \mathtt{0} & -\mathtt{I} \\ \mathtt{C} & \mathtt{B} \end{pmatrix}\right] \begin{pmatrix} p'' \\ x_3 p'' \end{pmatrix} = 0 \tag{A.35}$$

Lastly, consider $\lambda = x_3$ and solve the generalized eigenvalue problem for

$$\begin{pmatrix} \mathtt{I} & \mathtt{0} \\ \mathtt{0} & \mathtt{A} \end{pmatrix} \lambda - \begin{pmatrix} \mathtt{0} & \mathtt{I} \\ -\mathtt{C} & -\mathtt{B} \end{pmatrix} \tag{A.36}$$

The matrices are sized 24×24 so there is up to 24 solutions. This method, generalizes the companion matrix method with matrices as its entries. It is fully described and proven in [5] for any polynomial matrix equation. Given that only real eigenvalues are considerable to be a solution and there is at least 8 complex eigenvalues (those we know apriori) there can be up to 16 solutions. The generalized eigenvalue problem may be reduced to a normal eigenvalue problem when the two matrices in (A.36) are not a singular pencil, by either making inversion of one of the matrices or substituting λ to form linear combination of the matrices when they both happen to be singular (but together contain the entire linear space). Furthermore, QR decomposition can be used to get rid of the eigenvalues known apriori, by using double-shift algorithm as the eigenvalues are complex. While this optimization saves some computation time in general cases, it significantly degrades numerical accuracy. Practically, λ has to be substituted in nearly all cases (the substitution is randomly providing linear combinations of the original matrices, limited by the number of iterations) to provide sufficient conditioning for the matrix inversion. For this reason, the optimization was abandoned in our implementation, as solving the generalized eigenvalue problem meets our speed requirement.

Solving $\mathbf{x_4}$ and $\mathbf{c_5}$, $\mathbf{s_5}$

There are two ways to solve x_4 and c_5, s_5. One way, as presented by Manocha and Canny, is to use the eigenvector from the previous eigenvalue decomposition. Consider v as an eigenvector corresponding to a chosen eigenvalue (x_3). Given (A.35), the vector p'' is a linear combination of all eigenvectors for the corresponding eigenvalue. When the geometric multiplicity of the eigenvalue is equal to 1 then it can be written as

$$v_{24\times1} = \begin{pmatrix} \mathbf{v}_{12\times1} \\ x_3 \mathbf{v}_{12\times1} \end{pmatrix} = k \begin{pmatrix} p'' \\ x_3 p'' \end{pmatrix}, \quad k \in \mathbb{R} \tag{A.37}$$

and we use the eigenvector to solve x_4 and c_5, s_5. It is expensive to determine the geometric multiplicity of the eigenvalue and the method looses accuracy for eigenvalues with higher algebraic multiplicity. Thus when the algebraic multiplicity is higher and 1, i.e. all cases in which geometric multiplicity *may* be higher than 1, use the other way to solve x_4 and c_5, s_5. The condition for the geometric multiplicity is missing in Manocha and Canny paper. There are two possible scenarios, depending on whether x_4 converges to infinity, this is

determined based on whether $\mathbf{v}_{12}$ member is non-zero:

$$\begin{pmatrix} \mathbf{v}_1/\mathbf{v}_{12} & {x_4}^3 s_5 & \\ \mathbf{v}_2/\mathbf{v}_{12} & {x_4}^3 c_5 & \\ \mathbf{v}_3/\mathbf{v}_{12} & {x_4}^3 & \\ \mathbf{v}_4/\mathbf{v}_{12} & {x_4}^2 s_5 & \\ \mathbf{v}_5/\mathbf{v}_{12} & {x_4}^2 c_5 & \\ \mathbf{v}_6/\mathbf{v}_{12} = & {x_4}^2 & x_4 \in \mathbb{R} \\ \mathbf{v}_7/\mathbf{v}_{12} & x_4 s_5 & \\ \mathbf{v}_8/\mathbf{v}_{12} & x_4 c_5 & \\ \mathbf{v}_9/\mathbf{v}_{12} & x_4 & \\ \mathbf{v}_{10}/\mathbf{v}_{12} & s_5 & \\ \mathbf{v}_{11}/\mathbf{v}_{12} & c_5 & \\ 1 & 1 & \end{pmatrix} \begin{pmatrix} \mathbf{v}_1/\mathbf{v}_3 & s_5 & \\ \mathbf{v}_2/\mathbf{v}_3 & c_5 & \\ 1 & 1 & \\ 0 & s_5/x_4 & \\ 0 & c_5/x_4 & \\ 0 = & 1/x_4 & x_4 \to \infty \\ 0 & x_4 s_5/{x_4}^2 & \\ 0 & x_4 c_5/{x_4}^2 & \\ 0 & 1/{x_4}^2 & \\ 0 & s_5/{x_4}^3 & \\ 0 & c_5/{x_4}^3 & \\ 0 & 1/{x_4}^3 & \end{pmatrix} \tag{A.38}$$

This part is simplified by removing the substitution of c_5, s_5 by x_5 because we don't have to deal with cases when x_5 would be converging to infinity.

The other way, as presented by Raghaven and Roth, is to solve x_4 just like x_3. In (A.28), matrix $\mathtt{Z}'$ is now plain numbers. Rearrange it so polynomials in x_4 are inside the matrix

$$\mathtt{F}_{6\times 3}(x_4^2, x_4) \begin{pmatrix} s_5 \\ c_5 \\ 1 \end{pmatrix} = 0 \tag{A.39}$$

Use the first three equations (for best conditioning) and solve x_4 as a generalized eigenvalue problem. After that solve c_5, s_5 as a set of linear equations.

A.2.4 Solving remaining variables

After solving θ_3, θ_4 and θ_5 head back to (A.19) in which the left hand side is now plain numbers. As matrix $\mathtt{R}$ is triangular, c_1, s_1 and c_2, s_2 are solved directly from the equation without the need of decomposing it any further

$$\begin{pmatrix} t_5 \\ t_6 \\ t_7 \\ t_8 \end{pmatrix} = \begin{pmatrix} \mathtt{R}_{55} & \mathtt{R}_{56} & \mathtt{R}_{57} & \mathtt{R}_{58} \\ 0 & \mathtt{R}_{66} & \mathtt{R}_{67} & \mathtt{R}_{68} \\ 0 & 0 & \mathtt{R}_{77} & \mathtt{R}_{78} \\ 0 & 0 & 0 & \mathtt{R}_{88} \end{pmatrix} \begin{pmatrix} s_1 \\ c_1 \\ s_2 \\ c_2 \end{pmatrix} \tag{A.40}$$

where $t_{14\times 1}$ is

$$t = \mathtt{Q}^\top \mathtt{P}\, p \tag{A.41}$$

The last variable to solve, θ_6, is solved as a set of linear equations from (A.4) using LU decomposition.

A.3 Implementation

A.3.1 Symbolic Preprocessing

The algorithm performs symbolic preprocessing, treating the entries of $\mathtt{M}_\mathtt{H}$ as input variables and the Denavit-Hartenberg notation of the manipulator as parameters. The symbolic

preprocessing is done in MAPLE, for derivation and simplification of the expressions. There are several stages of symbolic preprocessing:

- Generation of $\mathtt{P}$ and $\mathtt{N}$ matrices. Symbolically calculates $\mathtt{P}$ and $\mathtt{N}$ matrices using $\mathtt{M_H}$ and Denavit-Hartenberg notation of the manipulator. Since $\mathtt{P}$ contains polynomials in c_3, s_3, it is expressed as
$$\mathtt{P} = \mathtt{P_A} s_3 + \mathtt{P_B} c_3 + \mathtt{P_C} \tag{A.42}$$
where $\mathtt{P_A}$, $\mathtt{P_B}$ and $\mathtt{P_C}$ are matrices containing plain numbers.

- Generation of matrices for resolving the generalized eigenvalue problem. It starts off matrix $\mathtt{Z}$ which is calculated online from current $\mathtt{P_A}$, $\mathtt{P_B}$, $\mathtt{P_C}$ and $\mathtt{N}$ and creates the matrices $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ to be used for solving generalized eigenvalues.

- Raghaven and Roth way to solve x_4 and c_5, s_5. Again starts off matrix $\mathtt{Z}$ and transforms it into matrix $\mathtt{F}$ which is expressed as
$$\mathtt{F} = \mathtt{F_A} x_4^2 + \mathtt{F_B} x_4 + \mathtt{F_C} \tag{A.43}$$
and matrices $\mathtt{F_A}$, $\mathtt{F_B}$ and $\mathtt{F_C}$ are used just like matrices $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ in the generalized eigenvalue problem.

- Matrices for c_1, s_1 and c_2, s_2. No further preprocessing is made, triangular matrix $\mathtt{R}$ in its form is sufficient.

- Matrices for solving c_6, s_6. Two matrices $\mathtt{M_{6L}}_{\,6\times 2}$ (left hand side) and $\mathtt{M_{6R}}_{\,6\times 1}$ (right hand side) are symbolically expressed from (A.4).

The symbolic preprocessing is done offline as it is independent on the end-effector pose. In particular, parameters a_i, d_i and λ_i, μ_i are substituted with manipulator values into matrices $\mathtt{P_A}$, $\mathtt{P_B}$, $\mathtt{P_C}$ and $\mathtt{N}$.

A.3.2 Numerical Substitution

Given the pose of the end-effector $\mathtt{M_{HV}}$, substitute it into $\mathtt{P_A}$, $\mathtt{P_B}$, $\mathtt{P_C}$ and $\mathtt{N}$ matrices. Consider the corresponding numerical matrices to be $\mathtt{P_{AV}}$, $\mathtt{P_{BV}}$, $\mathtt{P_{CV}}$ and $\mathtt{N_V}$. Calculate QR decomposition of $\mathtt{N_V}$, let it be expressed as
$$\mathtt{N_V} = \mathtt{Q_V} \mathtt{R_V}$$
and check diagonal of $\mathtt{R_V}$. As noted above, QR decomposition is not rank revealing but it can be safely estimated that matrix $\mathtt{R_V}$ has full rank by checking its diagonal whether it has non-zero elements
$$|\mathtt{R_{ii}}| > \epsilon_Q, \qquad i \in \{1, 2, ..., 8\} \tag{A.44}$$
in which ϵ_Q is a user defined constant to test rank deficiency. Please note that this constant should be based on observing the actual values of $\mathtt{R_V}$. Choosing a bigger value here leads to more end-effector poses being further decomposed by SVD to determine the rank and therefore to a valid result at a performance loss, unlike choosing insufficient value would cause the potential results to be discarded on failed integrity checks. When we are unsure about $\mathtt{R_V}$ having full rank, decompose $\mathtt{R_V}$ with SVD, let it be expressed as
$$\mathtt{R_V} = \mathtt{U_V} \Sigma_\mathtt{V} \mathtt{V_V}^\top$$

and again check for rank deficiency in $\Sigma_{\mathtt{V}}$

$$\sigma_i = \begin{cases} \Sigma_{\mathtt{Vii}}, & x > \epsilon_S \\ 0, & \text{otherwise} \end{cases}, \qquad i \in \{1, 2, ..., 8\} \tag{A.45}$$

in which ϵ_S is another user defined constant for testing rank deficiency. Usually the same constant can be used for ϵ_S and ϵ_Q.

Further numerical problems arise when solving x_3, x_4 and c_5, s_5 and the generalized eigenvalue problem. Consider the following generalized eigenvalue problem $\mathtt{E}_1\lambda - \mathtt{E}_2$. The accuracy decreases in the following cases

- Eigenvalues with greater algebraic multiplicity are calculated with worse accuracy (the greater multiplicity, the worse accuracy). This usually results in two or more eigenvalues very close to each other on either the real or the imaginary axis.
- Matrices $\mathtt{E}_1$ and $\mathtt{E}_2$ may form a singular pencil and then some of the generalized eigenvalues cannot be defined (in Schur decomposition both a_{ii} and b_{ii} are equal to 0).
- "Infinite" generalized eigenvalue when $\mathtt{E}_1$ has at least one eigenvalue equal to 0. This will cause the eigenvector for the "infinite" eigenvalue to be less accurate.

Additionally calculating x_4 and c_5, s_5 from the eigenvector cannot be used if the geometric multiplicity is greater than 1. A few user defined constants are used to deal with the situation

- A constant to check whether the eigenvalue is real or complex. Consider $\lambda = \lambda_R + \lambda_i i$ where $i = \sqrt{-1}$. Then define the constant ϵ_I as

  $$\lambda' = \begin{cases} \lambda_R + \lambda_I i, & |\lambda_I| > \epsilon_I \\ \lambda_R, & \text{otherwise} \end{cases} \tag{A.46}$$

  and discard all λ's that are complex.
- A constant to check for greater algebraic multiplicity. Consider two eigenvalues λ'_1 and λ'_2. Define a constant ϵ_R such as two λ'_1 and λ'_2 are considered as a single eigenvalue with algebraic multiplicity of two when

  $$|\lambda'_1| - |\lambda'_2| < \epsilon_R \tag{A.47}$$

  More than two eigenvalues can be clustered this way. It is safer to use the actual eigenvalues and treat them as if they had greater algebraic multiplicity each. Do not add arithmetic mean of the clustered eigenvalues as it will be ill-conditioned if there were two very close eigenvalues that were not completely equal and still got clustered. There is an optimization step later fixing the case of one eigenvalue with greater multiplicity.
- A constant is to determine whether the eigenvalue is to be considered infinite. Define ϵ_F such as λ' is considered infinite when

  $$\frac{1}{|\lambda'|} < \epsilon_F \tag{A.48}$$

Another numerical problems occur when calculating x_4 and c_5, s_5 from the eigenvector. From (A.37) we know that eigenvector looks like

$$v = \begin{pmatrix} \mathbf{v} \\ x_3\mathbf{v} \end{pmatrix}$$

Depending on whether $|x_3| > 1$, pick the part with greater numbers for better numerical accuracy. Furthermore it's needed to check whether x_4 is infinite or not. From (A.38) check whether $|\mathbf{v}_{12}| < \epsilon_F$ which is true when x_4 converges to infinity. Obtain c_5, s_5 given the way in (A.38). Check integrity of c_5, s_5 over their square values. They are valid when

$$|(c_5^2 + s_5^2) - 1| < \epsilon_P \tag{A.49}$$

in which ϵ_P is a user defined constant for perturbation. It is not necessary to use square root over the sum of the square of sine and cosine but the constant should reflect that. Then check the integrity of the result by comparing numerical values of $\mathbf{v}$ vector with the newly calculated x_4 and c_5, s_5 for the other rows (e.g. check if $\mathbf{v}_6 = x_4^2$ etc.). If there is a mismatch in any check, use the Raghaven and Roth way to calculate x_4 and c_5, s_5.

When using Raghaven and Roth way to calculate c_5, s_5 and further when calculating c_1, s_1, c_2, s_2 and c_6, s_6 we have a set of equations with two bounded variables. Consider the general structure of such set of equations as

$$\begin{gathered} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} s_i \\ c_i \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \\ s_i^2 + c_i^2 = 1 \end{gathered} \tag{A.50}$$

This set of equations is solved using LU decomposition from the upper equation, using ϵ_S as a constant to test rank deficiency. Given the rank of the system there may be either infinite number of solutions (rank is equal to 0), two solutions (rank is equal to 1 but we are bounded by the second equation, in this case there is either c_i given and $s_i = \pm\sin(\arccos(c_i))$ or vice versa) or one solution for full rank. Verify the integrity using the second equation with perturbation constant ϵ_P. When solving c_1, s_1 and c_2, s_2, member a_{21} in (A.50) is equal to zero and LU decomposition is not needed (it's a sub-matrix over the diagonal of $\mathtt{R}_\mathtt{V}$, see (A.40)).

Methods used in solving x_4 and c_5, s_5 may produce numerical error large enough to make the rest of the algorithm fail in solving the remaining variables in poorly conditioned cases. In order to minimize the impact of these numerical errors we use Newton optimization over (A.25). Given that $\mathtt{Z}(c_i, s_i)$ can be converted to $\mathtt{Z}'(x_i^2, x_i)$, the local convergence is quadratic and a few steps are usually enough to minimize the error. The Newton optimization can be speeded up by approximating the sine and cosine functions of the increment

$$\begin{gathered} \sin(x + x_0) = \sin(x)\cos(x_0) + \cos(x)\sin(x_0) \\ \cos(x + x_0) = \cos(x)\cos(x_0) - \sin(x)\sin(x_0) \\ \sin(x_0) \approx x_0 \\ \cos(x_0) \approx 1 - \frac{{x_0}^2}{2} \end{gathered} \tag{A.51}$$

and define ϵ_R as a limit to recalculate the sine and cosine when $|x_0| > \epsilon_R$. Furthermore define ϵ_N as a tolerance limit for the optimization. The ϵ_R and ϵ_N should correspond to each other, in a manner that the error when approximating sine and cosine should be lower than the optimization tolerance. Use ϵ_S to test rank deficiency of the Jacobian in the Newton optimization.

Finally, numerical accuracy is severely ill-conditioned when using the general 6R algorithm on more constrained systems, such as 3-2-1 manipulator, especially in their singular configurations. The constrains on such manipulators can be used to simplify the algorithm however.

A.3.3 Verification

The implementation was verified over several thousand of end-effector poses, with portion being completely random poses and the other portion being poses in which the algorithm is poorly conditioned (i.e. choosing values to make x_i converge to infinity, singular configurations, using more constrained manipulator with some of the axes being parallel or with two following joins having intersecting axes, while keeping the manipulator general 6R). We studied the accuracy of the algorithm, classifying it either as poor accuracy or invalid result, given the deviation between the requested transformation matrix and transformation matrix obtained through direct kinematics of the returned joint coordinates. Furthermore we studied that poses generated through direct kinematics are returning the original coordinates and runtime of the algorithm, for both normal and singular configurations. We used MotoMidMan MA1400, Denavit-Hartenberg notation on table A.1, to test the algorithm on.

Joint	α [°]	a [mm]	θ [°]	d [mm]
1.	-90	150	0	450
2.	180	614	270	0
3.	-90	200	0	0
4.	90	0	0	-640
5.	90	30	270	0
6.	0	0	0	200

Table A.1. MotoMidMan MA1400 Denavit-Hartenberg Notation

An example run of the verification algorithm can be seen on table A.2. The difference between the requested and the obtained transformation matrix is divided into three groups. If the difference is lower than 0.01 mm in location and 0.001 in rotation matrix (square root of the sum of squares of the elements) then it's considered accurate. If the difference is higher than 1 mm or 0.1 in rotation matrix then it's considered incorrect. Everything in between is considered inaccurate.

Standard Positions	
Correct solutions	13222
Inaccurate solutions	164
Incorrect solutions	0
from which detected as unreachable	0
Solutions not returning original DKT coordinates	0
Total successfulness	98.77 %
including inaccurate results	100.00 %
Singluar Positions	
Correct solutions	11205
Inaccurate solutions	61
Incorrect solutions	0
from which detected as unreachable	0
Solutions not returning original DKT coordinates	2
Total successfulness	99.46 %
including inaccurate results	100.00 %
Unreachable Positions	
Unreachable positions correctly detected	1000
Non-existing solutions found by mistake	0
Positions turned out to be reachable	0
Inaccurate reachable positions	0
Total successfulness	100.00 %
including inaccurate results	100.00 %
Statistics	
Poses requiring Newton optimization	15978
Poses ineligible for eigenvector method	7936
Singular poses	3998
IKT runtime	1.30 ms
for singular poses	1.17 ms
Total	
Total tested poses	25652
Total correct solutions	25427
Total correct inaccurate solutions	225
Total correct incorrect solutions	0
Total successfulness	99.12 %
including inaccurate results	100.00 %

Table A.2. Inverse Kinematics Verification

A.3.4 Conclusion

In this report we presented a description and an implementation of the general 6R manipulator inverse kinematics task. The algorithm starts with several steps of symbolic preprocessing and uses matrix decompositions to reduce the problem to generalized eigenvalue problem. The numerical accuracy of the algorithm was well analyzed and a couple of user defined constants were defined to accomplish the required accuracy and keep the performance on industrial level. The algorithm was extensively tested on various end-effector poses including poorly conditioned ones. Average runtime of the algorithm was 1.2 ms. This algorithm can be directly extended on any general 6R manipulator, with the exception of more constrained manipulators, such as 3-2-1 manipulator, which require further simplification of the process due to persistent poor conditioning in the general 6R algorithm.

A.4 Documentation

A.4.1 Platform

The algorithm was implemented in C using LAPACK and BLAS libraries. The manipulator specific matrices are prefabricated inside MAPLE and exported to C environment, using milimeters for the distance and radians for the angles. The implementation intends to provide accurate result for every possible end-effector pose rather than keeping the runtime as low as possible, it was also left in easily readable state for matrix indexing and to easen potential further modifications or improvements. The current runtime is around 1.2 ms which is sufficient enough for practical usage. The implementation will work on Windows, Linux (both 32-bit and 64-bit) and other platforms, provided they are supported by LAPACK and BLAS. Please note that LAPACK is written in Fortran and uses column major order in matrix indexing.

A.4.2 Numerical Optimization

Numerical optimization and tolerance of the method is recommended to be relevant to the actual manipulator accuracy and repeatability. Given the usual manipulator repeatability to be around 0.02 mm, the default values for numerical optimization were set to give a result with maximal error around 0.001 mm in the distance and 0.0001 in the angles represented by rotation matrix (the error is calculated as a square root of the sum of squares of the elements). The numerical optimization setup is represented by the following structure in the algorithm:

```
typedef struct
{
  double perturbation;
  double imaginary_tol;
  double root_tol;
  double eigenvalue_safety;
  double eigenvalue_infinite_limit;
  double optimization_tol;
  int optimization_max_steps;
  double optimization_bigdiff_tol;
  int optimization_bigdiff_extra_steps;
  double optimization_recalc_limit;
  double dulpicate_solution_tol;
  int test_joint_limits;
} gen6rikt_limits_t;
```

The members of the **gen6rikt_limits_t** structure have the following meaning:

- **perturbation** is used as ϵ_Q, ϵ_S and partly as ϵ_P i.e. to test rank deficiency and numerical error in all intermediate results that have already passed numerical optimization.

- **imaginary_tol** is ϵ_I i.e. the constant to determine whether the eigenvalue is real or complex
- **root_tol** is ϵ_R i.e. the constant for clustering eigenvalues into one with greater multiplicity
- **eigenvalue_safety** is used as ϵ_P for solving x_4 and c_5, s_5 from the eigenvector in order to verify the integrity of the eigenvector
- **eigenvalue_infinite_limit** is ϵ_F i.e. the constant to check whether the eigenvalue is considered finite or not
- **optimization_tol** is ϵ_N i.e. the tolerance for the Newton optimization, below which it will not optimize further
- **optimization_max_steps** defines the maximum number of steps in the Newton optimization
- **optimization_bigdiff_tol** is used to determine whether the deviation in the Newton optimization is considered too high and allows more steps in the optimization
- **optimization_bigdiff_extra_steps** is the number of extra steps in the Newton optimization if there is a high deviation
- **optimization_recalc_limit** is ϵ_R i.e. the constant to determine whether in the Newton optimization we should recalculate sine a cosine for some of the arguments
- **dulpicate_solution_tol** is a tolerance in joint coordinates to determine duplicate solutions, this may usually occur when there is an eigenvalue with greater algebraic multiplicity
- **test_joint_limits** is just for debugging purposes, whether to use the manipulator joint limits to discard invalid solutions

A.4.3 Doxygen

Doxygen documentation describing the functions and other structures in detail is provided separately along with the implementation in C.

Appendix B
Tumbl Technical Reference Manual

This chapter contains an overview of Tumbl registers, interrupts and instruction set. Peripherals wired in memory space are described in appendix D.

B.1 General Purpose Registers

The following table overviews the convention of general purpose registers, alongside whether it belongs to hardware enforcement or is up to the convention.

Registers	Enforcement	Description
R0	HW	Always has a value of zero, writings are discarded
R1	SW	Stack pointer
R2	SW	Read-only small data area anchor
R3 - R4	SW	Returning values
R5 - R10	SW	Passing parameters
R11 - R12	SW	Temporaries
R13	SW	Read-only small data area anchor
R14	HW	Return address for interrupt
R15	SW	Return address for subroutines
R16 - R17	SW	Local stack, must be saved across function calls.
R18	SW	Reserved for assembler
R19 - R31	SW	Local stack, must be saved across function calls.

Table B.1. General purpose registers overview

B.2 Condition Evaluation

The following table overview condition evaluation.

Name	Semantics	Evaluation
EQ	$a = b$	Rd = 0
NE	$a \neq b$	Rd != 0
LT	$a < b$	Rd[31] = 1
LE	$a \leq b$	Rd[31] = 1 or Rd = 0
GT	$a > b$	Rd[31] = 0 and Rd != 0
GE	$a \geq b$	Rd[31] = 0

Table B.2. Condition Evaluation

B.3 Special Registers

The only special available register is the machine status register, MSR:

Bit	Code	Description
31 - 3	N/A	Reserved
2	C	Carry bit
1	IE	Interrupt enable bit
0	N/A	Reserved

Table B.3. Machine status register

B.4 Interrupts

Tumbl only supports single interrupt request type, branching to address 0x0000 0010 and setting interrupt enable bit in machine status register to 0. It has a designated instruction RTID to return from interrupt, setting the interrupt enable bit back to 1. R14 is a register used to assign the returning address and the core executes branching with link operation. There is no banking support, and thus all registers have to be saved, this also includes carry-bit in MSR, which has to be manually saved using MTS instruction and then restored using MFS instruction.

B.5 Instruction Set

All Tumbl instructions have fixed length of 32 bits and most of them keep the same binary encoding as with original Microblaze processor. Table B.5 describes the nomenclature used in the semantics of each instruction. The instruction set is on table B.5.

Symbol	Description
Ra	R0-R31, General purpose register, R0 is always 0, source operand a
Rb	R0-R31, General purpose register, R0 is always 0, source operand b
Rd	R0-R31, General purpose register, R0 cannot be written to, destination operand 1
Sd	Special purpose destination register
Sa	Special purpose source operand register
Cnd	Conditional operand, used for conditional branching and if-then instructions, see table B.2
Cnd(Ra)	Conditional operant evaluation
Cnd(Ra,Rb)	Assume Rd := Ra + $\overline{\text{Rb}}$ + 1 and then see Cnd(Rd)
Imm	16-bit signed immediate value operand b, can be extended with IMM instruction to 32 bits
Imm5	5-bit unsigned immediate value operand b, cannot be extended
MSR	Machine status register
C	Carry bit from machine status register
Addr	Memory contents of Addr (32-bit aligned)
$\overline{\text{x}}$	Bit inverted value of x
:=	Assignment operation
=	Equality comparison
!=	Inequality comparison
>	Greater than comparison

Symbol	Description
>=	Greater than or equals comparison
<	Lesser than comparison
<=	Lesser than or equals comparison
+	Arithmetic add
*	Arithmetic multiply
<< x	Bit shift right x bits
>> x	Bit shift left x bits
and	Logic and
or	Logic or
xor	Logic exclusive or
&	Concatenation
signed	Operation performed on signed integer datatype, this is the default mode unless specified otherwise
unsigned	Operation performed on unsigned integer datatype

Table B.4. Tumbl Instruction Nomenclature

Directive	0-5	6-10	11-15	16-20	21-31	Semantics
ADD Rd, Ra, Rb	000000	Rd	Ra	Rb	00000000000	Rd := Ra + Rb
RSUB Rd, Ra, Rb	000001	Rd	Ra	Rb	00000000000	Rd := Ra + $\overline{Rb}$ + 1
ADDC Rd, Ra, Rb	000010	Rd	Ra	Rb	00000000000	Rd := Ra + Rb + C
RSUBC Rd, Ra, Rb	000011	Rd	Ra	Rb	00000000000	Rd := Ra + $\overline{Rb}$ + C
ADDK Rd, Ra, Rb	000100	Rd	Ra	Rb	00000000000	Rd := Ra + Rb C := Rd[32]
RSUBK Rd, Ra, Rb	000101	Rd	Ra	Rb	00000000000	Rd := Ra + $\overline{Rb}$ + 1 C := Rd[32]
ADDKC Rd, Ra, Rb	000110	Rd	Ra	Rb	00000000000	Rd := Ra + Rb + C C := Rd[32]
RSUBKC Rd, Ra, Rb	000111	Rd	Ra	Rb	00000000000	Rd := Ra + $\overline{Rb}$ + C C := Rd[32]
MUL Rd, Ra, Rb	010000	Rd	Ra	Rb	00000000000	Rd := Ra * Rb
BSRL Rd, Ra, Rb	010001	Rd	Ra	Rb	00000000000	Rd := 0 & Ra >> Rb[4:0]
BSRA Rd, Ra, Rb	010001	Rd	Ra	Rb	01000000000	Rd := Ra >> Rb[4:0] arithmetic shift
BSLL Rd, Ra, Rb	010001	Rd	Ra	Rb	10000000000	Rd := Ra << Rb[4:0] & 0
CMP Rd, Ra, Rb	010010	Rd	Ra	Rb	00000000000	Rd := Ra + $\overline{Rb}$ + 1 Rd[0] := 0 if Rb >= Ra else Rd[0] := 1
CMPU Rd, Ra, Rb	010011	Rd	Ra	Rb	00000000000	Rd := Ra + $\overline{Rb}$ + 1 Rd[0] := 0 if Rb >= Ra else Rd[0] := 1, unsigned
IT Cnd, Ra, Rb	010100	00 & Cnd	Ra	Rb	00000000000	Cnd(Ra,Rb) if-then conditional execution
ITT Cnd, Ra, Rb	010100	01 & Cnd	Ra	Rb	00000000000	Cnd(Ra,Rb) if-then-then conditional execution
ITE Cnd, Ra, Rb	010100	10 & Cnd	Ra	Rb	00000000000	Cnd(Ra,Rb) if-then-else conditional execution
ITU Cnd, Ra, Rb	010100	00 & Cnd	Ra	Rb	00000000000	Cnd(Ra,Rb), unsigned if-then conditional execution
ITTU Cnd, Ra, Rb	010100	01 & Cnd	Ra	Rb	00000000000	Cnd(Ra,Rb), unsigned if-then-then conditional execution
ITEU Cnd, Ra, Rb	010100	10 & Cnd	Ra	Rb	00000000000	Cnd(Ra,Rb), unsigned if-then-else conditional execution

Directive	0-5	6-10	11-15	16-20	21-31	Semantics
OR Rd, Ra, Rb	100000	Rd	Ra	Rb	00000000000	Rd := Ra or Rb
AND Rd, Ra, Rb	100001	Rd	Ra	Rb	00000000000	Rd := Ra and Rb
XOR Rd, Ra, Rb	100010	Rd	Ra	Rb	00000000000	Rd := Ra xor Rb
ANDN Rd, Ra, Rb	100011	Rd	Ra	Rb	00000000000	Rd := Ra and $\overline{\text{Rb}}$
CLZ Rd, Ra	100100	Rd	Ra	00000	00000000000	Rd := 31 − ceil(log2(Ra)) Rd := 32 when Ra = 0
SRA Rd, Ra	100100	Rd	Ra	00000	00000000001	Rd := Ra >> 1 Rd[31] := Ra[31], C := Ra[0]
SRC Rd, Ra	100100	Rd	Ra	00000	00000100001	Rd := Ra >> 1 Rd[31] := C, C := Ra[0]
SRL Rd, Ra	100100	Rd	Ra	00000	00001000001	Rd := Ra >> 1 Rd[31] := 0, C := Ra[0]
SEXT8 Rd, Ra	100100	Rd	Ra	00000	00001100000	Rd := Ra[24:31] signed extension
SEXT16 Rd, Ra	100100	Rd	Ra	00000	00001100001	Rd := Ra[16:31] signed extension
MTS Sd, Ra	100101	00000	Ra	11 & Sd		Sd := Ra MSR is the only supported Sd
MFS Rd, Sa	100101	Rd	00000	10 & Sa		Rd := Sa MSR is the only supported Sa
BR Rb	100110	00000	00000	Rb	00000000000	PC := PC + Rb
BRL Rd, Rb	100110	Rd	00100	Rb	00000000000	PC := PC + Rb Rd := PC
BRA Rb	100110	00000	01000	Rb	00000000000	PC := Rb
BRAL Rd, Rb	100110	Rd	01100	Rb	00000000000	PC := Rb Rd := PC
BRD Rb	100110	00000	10000	Rb	00000000000	PC := PC + Rb Uses delay slot
BRLD Rd, Rb	100110	Rd	10100	Rb	00000000000	PC := PC + Rb Rd := PC + 4 Uses delay slot
BRAD Rb	100110	00000	11000	Rb	00000000000	PC := Rb Uses delay slot
BRALD Rd, Rb	100110	Rd	11100	Rb	00000000000	PC := Rb Rd := PC + 4 Uses delay slot
BRC Cnd, Ra, Rb	100111	00 & Cnd	Ra	Rb	00000000000	if Cnd(Ra) then PC := PC + Rb
BRCD Cnd, Ra, Rb	100111	10 & Cnd	Ra	Rb	00000000000	if Cnd(Ra) then PC := PC + Rb Uses delay slot
LBU Rd, Ra, Rb	110000	Rd	Ra	Rb	00000000000	Addr := Ra + Rb Rd[0:23] := 0 Rd[24:31] := *Addr[0:7]
LHU Rd, Ra, Rb	110001	Rd	Ra	Rb	00000000000	Addr := Ra + Rb Rd[0:15] := 0 Rd[16:31] := *Addr[0:15]
LW Rd, Ra, Rb	110010	Rd	Ra	Rb	00000000000	Addr := Ra + Rb Rd := *Addr
SB Rd, Ra, Rb	110100	Rd	Ra	Rb	00000000000	Addr := Ra + Rb *Addr[0:7] := Rd[0:7]
SH Rd, Ra, Rb	110100	Rd	Ra	Rb	00000000000	Addr := Ra + Rb *Addr[0:15] := Rd[0:15]
SW Rd, Ra, Rb	110100	Rd	Ra	Rb	00000000000	Addr := Ra + Rb *Addr := Rd

Directive	0-5	6-10	11-15	16-31	Semantics
ADDI Rd, Ra, Imm	001000	Rd	Ra	Imm[15:0]	Rd := Ra + Imm
RSUBI Rd, Ra, Imm	001001	Rd	Ra	Imm[15:0]	Rd := Ra + $\overline{Imm}$ + 1
ADDCI Rd, Ra, Imm	001010	Rd	Ra	Imm[15:0]	Rd := Ra + Imm + C
RSUBCI Rd, Ra, Imm	001011	Rd	Ra	Imm[15:0]	Rd := Ra + $\overline{Imm}$ + C
ADDIK Rd, Ra, Imm	001100	Rd	Ra	Imm[15:0]	Rd := Ra + Imm C := Rd[32]
RSUBIK Rd, Ra, Imm	001101	Rd	Ra	Imm[15:0]	Rd := Ra + $\overline{Imm}$ + 1 C:= Rd[32]
ADDIKC Rd, Ra, Imm	001110	Rd	Ra	Imm[15:0]	Rd := Ra + Imm + C C := Rd[32]
RSUBIKC Rd, Ra, Imm	001111	Rd	Ra	Imm[15:0]	Rd := Ra + $\overline{Imm}$ + C C:=Rd[32]
MULI Rd, Ra, Imm	011000	Rd	Ra	Imm[15:0]	Rd := Ra * Imm
BSRLI Rd, Ra, Imm5	011001	Rd	Ra	00000000000 & Imm5	Rd := 0 & Ra >> Imm5
BSRAI Rd, Ra, Imm5	011001	Rd	Ra	00000010000 & Imm5	Rd := Ra >> Imm5 arithmetic shift
BSLLI Rd, Ra, Imm5	011001	Rd	Ra	00000100000 & Imm5	Rd := Ra << Imm5 & 0
CMPI Rd, Ra, Rb	011010	Rd	Ra	Imm[15:0]	Rd := Ra + $\overline{Imm}$ + 1 Rd[0] := 0 if Imm >= Ra else Rd[0] := 1
CMPUI Rd, Ra, Rb	011011	Rd	Ra	Imm[15:0]	Rd := Ra + $\overline{Imm}$ + 1 Rd[0] := 0 if Imm >= Ra else Rd[0] := 1, unsigned
ITI Cnd, Ra, Imm	011100	00 & Cnd	Ra	Imm[15:0]	Cnd(Ra,Imm) if-then conditional execution
ITTI Cnd, Ra, Imm	011100	01 & Cnd	Ra	Imm[15:0]	Cnd(Ra,Imm) if-then-then conditional execution
ITEI Cnd, Ra, Imm	011100	10 & Cnd	Ra	Imm[15:0]	Cnd(Ra,Imm) if-then-else conditional execution
ITUI Cnd, Ra, Imm	011100	00 & Cnd	Ra	Imm[15:0]	Cnd(Ra,Imm), unsigned if-then conditional execution
ITTUI Cnd, Ra, Imm	011100	01 & Cnd	Ra	Imm[15:0]	Cnd(Ra,Imm), unsigned if-then-then conditional execution
ITEUI Cnd, Ra, Imm	011100	10 & Cnd	Ra	Imm[15:0]	Cnd(Ra,Imm), unsigned if-then-else conditional execution
ORI Rd, Ra, Imm	100000	Rd	Ra	Imm[15:0]	Rd := Ra or Imm
ANDI Rd, Ra, Imm	100001	Rd	Ra	Imm[15:0]	Rd := Ra and Imm
XORI Rd, Ra, Imm	100010	Rd	Ra	Imm[15:0]	Rd := Ra xor Imm
ANDNI Rd, Ra, Imm	100011	Rd	Ra	Imm[15:0]	Rd := Ra and $\overline{Imm}$
IMM Imm	101100	000000	000000	Imm[31:16]	Sets Imm[31:16] valid for next instruction only
RTS Ra, Imm	101101	00000	Ra	Imm[15:0]	PC := Ra + Imm
RTI Ra, Imm	101101	00001	Ra	Imm[15:0]	PC := Ra + Imm MSR[IE] := 1
RTSD Ra, Imm	101101	10000	Ra	Imm[15:0]	PC := Ra + Imm Uses delay slot
RTID Ra, Imm	101101	10001	Ra	Imm[15:0]	PC := Ra + Imm MSR[IE] := 1 Uses delay slot

Directive	0-5	6-10	11-15	16-31	Semantics
BRI Imm	100110	00000	00000	Imm[15:0]	PC := PC + Imm
BRLI Rd, Imm	100110	Rd	00100	Imm[15:0]	PC := PC + Imm Rd := PC
BRAI Rb	100110	00000	01000	Imm[15:0]	PC := Imm
BRALI Rd, Imm	100110	Rd	01100	Imm[15:0]	PC := Imm Rd := PC
BRID Rb	100110	00000	10000	Imm[15:0]	PC := PC + Imm Uses delay slot
BRLID Rd, Imm	100110	Rd	10100	Imm[15:0]	PC := PC + Imm Rd := PC + 4 Uses delay slot
BRAID Imm	100110	00000	11000	Imm[15:0]	PC := Imm Uses delay slot
BRALID Rd, Imm	100110	Rd	11100	Imm[15:0]	PC := Imm Rd := PC + 4 Uses delay slot
BRCI Cnd, Ra, Imm	100111	00 & Cnd	Ra	Imm[15:0]	if Cnd(Ra) then PC := PC + Imm
BRCID Cnd, Ra, Imm	100111	10 & Cnd	Ra	Imm[15:0]	if Cnd(Ra) then PC := PC + Imm Uses delay slot
LBUI Rd, Ra, Imm	110000	Rd	Ra	Imm[15:0]	Addr := Ra + Imm Rd[0:23] := 0 Rd[24:31] := *Addr[0:7]
LHUI Rd, Ra, Imm	110001	Rd	Ra	Imm[15:0]	Addr := Ra + Imm Rd[0:15] := 0 Rd[16:31] := *Addr[0:15]
LWI Rd, Ra, Imm	110010	Rd	Ra	Imm[15:0]	Addr := Ra + Imm Rd := *Addr
SBI Rd, Ra, Imm	110100	Rd	Ra	Imm[15:0]	Addr := Ra + Imm *Addr[0:7] := Rd[0:7]
SHI Rd, Ra, Imm	110100	Rd	Ra	Imm[15:0]	Addr := Ra + Imm *Addr[0:15] := Rd[0:15]
SWI Rd, Ra, Imm	110100	Rd	Ra	Imm[15:0]	Addr := Ra + Imm *Addr := Rd
HALT Imm5	111111	00000	00000	00000000000 & Imm5	HaltCode := Imm5 Halts processor with code Imm5

Table B.5. Tumbl instruction set

Appendix C
Tumbl C lanugage support

This chapter overviews building the pieces of Tumbl toolchain. The target was named **mbtumbl-elf** and throughout configuration choose the same path for prefix and system root for all tools, with placeholders as **/path/to/prefix** and **/path/to/sysroot**.

C.1 binutils

binutils were patched for customized instruction and providing disassembled code. You can build them as:

```
> configure --with-gnu-as --with-gnu-ld --disable-nls
--target=mbtumbl-elf --with-sysroot=/path/to/sysroot
--prefix=/path/to/prefix
> make
> make install
```

Note that **binutils** by default do not clean up redundant `IMM` instructions, unless you pass **-relax** as a parameter.

C.2 gcc

gcc was patched with Tumbl machine descriptor for its C compiler and **libgcc**. Building it is a little bit more complicated:

```
> configure --target=mbtumbl-elf --with-ppl=no --with-cloog=no
CFLAGS_FOR_TARGET=-O2 CFLAGS_FOR_BUILD=-O2 --disable-shared
--disable-threads --disable-multilib --with-newlib
--enable-languages=c --disable-libquadmath --disable-libgomp
--disable-libssp --disable-sjlj-exceptions --disable-nls
--with-sysroot=/path/to/sysroot --prefix=/path/to/prefix
> make all-gcc
> make install-gcc
# copy newlib folder
> make all-target-libgcc
> make install-target-libgcc
```

As you can see, many things were disabled due to missing support or generally not being needed. There are few more thing to note: no optimization has been made and **gcc** doesn't support conditional execution instructions, `IT`, `ITT`, `ITE` and derived instructions. `CLZ` instruction is also not supported. As of now, any optimization has to be made by hand, however given that **MicroBlaze** is a synthesized core and not mainstream, no real optimization was ever made for it, unlike for **ARM** and **x86**. The reason why **newlib** has

to be built in the middle is that **libgcc** relies on it's header files. You may also need to manually create **/path/to/sysroot/usr/include**.

C.3 newlib

newlib was patched slightly, to deal with assembler changes. You can build it as:

```
> configure --target=mbtumbl-elf --prefix=/path/to/sysroot/usr
--disable-multilib --disable-newlib-supplied-syscalls
--disable-shared CFLAGS_FOR_TARGET=''-g -O2 -ffunction-sections
-fdata-sections''
> make
> make install
```

You may need to move the headers from **mbtumbl-elf** subdirectory up one level after installing.

Appendix D

Memory Map

This chapter is a summary of registers available and memory spaces for both master CPU and Tumbl.

D.1 Master CPU Memory Map

Master CPU memory space covering FPGA peripherals is **0x8000 0000 - 0x8003 FFFF**.

Register	Address	Type	Description
Tumbl Control Register	0x8000 3000	RW	Bits 31:5: reserved Bit 4 (R): Tumbl is halted using HALT instruction Bit 3 (RW): Toggles trace mode Bit 2 (RW): Write 1 to halt Tumbl (externally) Bit 1 (RW): Toggles interrupt, must be unset manually Bit 0 (RW): Toggles reset, must be unset manually Reset value: 0x0000 0001
Tumbl Trace Kick	0x8000 3004	W	Write 1 to pass one clock cycle to Tumbl, resumes execution when Tumbl is halted with HALT instruction
Tumbl Program Counter	0x8000 3008	R	Tumbl program counter
Tumbl Halt Code	0x8000 300C	R	Halt code when Tumbl is halted with HALT instruction
Measurement RD 1	0x8000 7FF0	R	Returns constant value 0xAAAAAAAA, used to test memory reading
Measurement WR 1	0x8000 7FF4	RW	Used to test memory writing
Measurement RD 2	0x8000 7FF8	R	Returns constant value 0x55555555, used to test memory reading
Measurement WR 2	0x8000 7FFC	RW	Used to test memory writing

Table D.1. Master CPU registers

Start Address	End Address	Description
0x8000 0000	0x8000 07FF	Tumbl instruction memory
0x8000 1000	0x8000 1FFF	Tumbl data memory

Table D.2. Master CPU memory spaces

D.2 Tumbl Memory Map

All Tumbl registers are available as well to master CPU. Tumbl external memory space is covered in address range **0x8002 0000 - 0x8003 FFFF** with **0x8002 0000** as the base address.

Register	Tumbl Address Master CPU Address	Type	Description
IRC 1 Count	0x0000 2000 0x8002 2000	RW	IRC 1 count
IRC 1 Index	0x0000 2004 0x8002 2004	RW	IRC 1 index count
IRC 2 Count	0x0000 2008 0x8002 2008	RW	IRC 2 count
IRC 2 Index	0x0000 200C 0x8002 200C	RW	IRC 2 index count
IRC 3 Count	0x0000 2010 0x8002 2010	RW	IRC 3 count
IRC 3 Index	0x0000 2014 0x8002 2014	RW	IRC 3 index count
IRC 4 Count	0x0000 2018 0x8002 2018	RW	IRC 4 count
IRC 4 Index	0x0000 201C 0x8002 201C	RW	IRC 4 index count
IRC 1 Status	0x0000 2020 0x8002 2020	RW	Bits 31:2: reserved Bit 1 (RW): Read error decoding **A** and **B** signals in quad counter, write 1 to reset it Bit 0 (R): IRC 1 Mark
IRC Reset	0x0000 2024 0x8002 2004	RW	Bits 31:1: reserved Bit 0: Write 1 to reset IRC peripherals, defaults to 1, must be cleared manually
IRC 2 Status	0x0000 2028 0x8002 2028	RW	Bits 31:2: reserved Bit 1 (RW): Read error decoding **A** and **B** signals in quad counter, write 1 to reset it Bit 0 (R): IRC 2 Mark
IRC 3 Status	0x0000 2030 0x8002 2030	RW	Bits 31:2: reserved Bit 1 (RW): Read error decoding **A** and **B** signals in quad counter, write 1 to reset it Bit 0 (R): IRC 3 Mark
IRC 4 Status	0x0000 2038 0x8002 2038	RW	Bits 31:2: reserved Bit 1 (RW): Read error decoding **A** and **B** signals in quad counter, write 1 to reset it Bit 0 (R): IRC 4 Mark

Table D.3. Tumbl registers

Tumbl Start Address Master CPU Start Address	Tumbl End Address Master CPU End Address	Description
0x0000 2400 0x8002 2400	0x0000 25FF 0x8002 25FF	LX Master memory, 16-bit word size, upper 16 bits set to 0 when reading and discarded when writing

Table D.4. Tumbl memory spaces

Tumbl instruction memory is wired on **0x0000 0000** - **0x0000 07FF** address range (on instruction fetch bus, cannot be used for data) and data memory is wired on **0x0000 0000** - **0x0000 0FFF** address range.

Bibliography

[1] J. Denavit and R. S. Hartenberg, *A kinematic notation for lower-pair mechanisms based upon matrices*, J. App. Mechanics, 77, pp. 215-221, 1955.

[2] M. Raghavan and B. Roth, *Kinematic analysis of the 6R manipulator of general geometry*, Int. Symp. Robotics Res., pp. 314-320, Tokyo, 1989.

[3] M. Raghavan and B. Roth,, *Inverse kinematics of the general 6R manipulator and related linkages*, Trans. ASME J. Mech. Des, pp 502-508, 1993.

[4] J. Manocha and J. F. Canny, *Real time Inverse Kinematics for General 6R Manipulators*, IEEE International Conference Robotics and Automation pp. 383-389, 1992.

[5] J. Manocha and J. F. Canny, *Efficient Inverse Kinematics for General 6R Manipulators*, IEEE Transactions on Robotics and Automation, Vol. 10, No. 5, pp. 648-657, October 1994.

[6] Songguo Liu and Shiqiang Zhu, *An Optimized Real Time Algorithm for the Inverse Kinematics of General 6R Robots*, IEEE International Conference Robotics and Automation pp. 2080-2084, 2007.

[7] V. Burian, *Control of brush-less DC motors with use of FPGA device*, Bachelor Thesis CTU FEE, 2011.

[8] K. Skup, *Motion Control for Mobile Robot*, Bachelor Thesis CTU FEE, 2007.

[9] PiKRON Ltd., *PXMC library*, http://pxmc.org/, 2001-2014.

[10] Xilinx, Inc., *MicroBlaze Processor Reference Guide*, http://xilinx.com/, UG081.

[11] Freescale Semiconductor, Inc., *PMSM Vector Control with Quadrature Encoder on Kinetis*, http://freescale.com/, 2012.

[12] T. Kranenburg, *MB-Lite*, http://opencores.org/project,mblite,overview, 2012.

[13] H. J. Lincklaen Arriëns, *MB-Lite+*, http://ens.ewi.tudelft.nl/ huib/vhdl/mb-lite_plus.php, April 2012.

[14] O. Girard, *openMSP430*, http://opencores.org/project,openmsp430, 2014.

[15] S. Rhoads, *Plasma - most MIPS I(TM) opcodes*, http://opencores.org/project,plasma, 2013.

[16] L. Barthe and L. V.Cargnini and P. Benoit and L. Torres, *The SecretBlaze: A Configurable and Cost-Effective Open-Source Soft-Core Processor*, IEEE International Symposium on Digital Object Identifier, pp. 310-313, 2011.

[17] Digi-key Corporation, http://www.digikey.com/, 2014.

[18] *System-less framework*, http://rtime.felk.cvut.cz/hw/index.php/System-Less_Framework, 2014.

[19] NXP Semiconductors, Inc., *LPC17xx User Manual*, August 2010.

[20] Xilinx, Inc., *Spartan-6 FPGA Configuration*, 2013.

[21] Mentor Graphics, *ModelSim User's Manual*, 2012.

[22] PiKRON Ltd., *LX_CPU*, http://pikron.com/pages/products/cpu_boards/lx_cpu.html, 2014.

[23] M. Meloun and T. Pajdla, *Inverse Kinematics For A General 6R Manipulator*, Research Reports of CMP, Czech Technical University in Prague, CTU–CMP–2013–29, 2013,