LULEÅ TEKNISKA UNIVERSITET

PROJECT REPORT

A VHDL Implementation of a MIPS

January 7, 2000
Anders Wallander

Department of Computer
Science and Electrical Engineering

ABSTRACT

As a tutorial in computer aided digital design a Microprocessor without Interlocked Pipline Stages (MIPS) was implemented using VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) and a Field Programmable Gate Array (FPGA). To demonstrate that the processor worked in real life a Dual Tone Multi Frequency (DTMF) decoder was made in another project. This report describes the work and the conclusions made during the project.

CONTENTS

1 INTRODUCTION

Today microprocessors can be found in almost every digital system. The decision to include a microprocessor in a design is often very clear because it transforms the design effort from a logic design into a software design. With the ever-increasing size and reductions in cost of FPGA devices, it is now possible to implement a complete system on one device, a System-On-Chip (SOC). Today the largest FPGA from XILINX is the VirtexE. It contains 150 million transistors, resulting in 3 million equivalent gates. SOC is of special interest for Information Appliances such as PDAs and Mobile Phones.

The idea of this project was to create a microprocessor as a building block in VHDL that later easily can be included in a larger design. It will be useful in systems where a problem is easy to solve in software but hard to solve with control logic. A state machine dedicated to the function can of course replace the microprocessor and associated software. However, at a high level of complexity it is easier to implement the function in software.

2 THE PROCESSOR

Early in the project a study on the existing RISC implementations was conducted. It was found that all RISC implementations are based on the same “commandments”.

A RISC system satisfy the following properties [3]:

1. Single-cycle execution of all (over 80 percent) instructions.
2. Single-word standard length of all instructions.
3. Small number of instructions, not to exceed about 128.
4. Small number of instruction formats, not to exceed about 4.
5. Small number of addressing modes, not to exceed about 4.
6. Memory accesses by load and store instructions only.
7. All operations, except load and store, are register-to-register, within the CPU.
8. Hardwired control unit.
9. A relatively large general-purpose CPU register file (at least 32 registers).

It was decided to base the project on the MIPS R2000 processor. It was well documented and a lot of knowledge on the MIPS existed at the University.

2.1 THE HISTORY OF THE RISC

RISC usually refers to a Reduced Instruction Set Computer. RISC designs call for each instruction to execute in a single cycle, which is done with pipelines. This reduces chip complexity and increases speed. Operations are performed on registers only and memory is only accessed by load and store operations. Most RISC concepts can be traced back to 1964 and the CDC 6600 by Seymore Cray’s Control Data Corporation, considered to be the first supercomputer [1]. It pioneered the heavy use of optimising register operations, while restricting memory access to load and store operations. The first system to formalise these principles was the IBM 801 project in 1975 [2]. The design goal was to speed up frequently used instructions while discarding complex instructions that slowed the overall implementation. Like the CDC 6600, memory access was limited to load/store operations. Branches were delayed, and instructions used a three-operand format common to RISC processors. Execution was pipelined, allowing 1 instruction per cycle. The 801 had thirty-two 32 bit registers, but no floating point unit. It implemented Harvard architecture with separate data and instruction caches, and had flexible addressing modes.

Around 1981, projects at Berkeley (RISC I and II) and Stanford University (MIPS) further developed these concepts. The term RISC came from Berkeley's project, which was the basis for the SPARC processor. Because of this, features are similar, including windowed registers and register zero wired to the value 0. Branches are delayed, and like Advanced RISC Machines’ ARM, all instructions have a bit to specify if condition codes should be set and execute in a 3-stage pipeline [2]. In addition, next and current PC are visible to the user, and last Program Counter (PC) is visible in supervisor mode. The Stanford MIPS project had similarities with the Berkeley RISC project and stood as the base for the commercial MIPS R2000. MIPS stood for Microprocessor without Interlocked Pipeline Stages, using the compiler to eliminate register conflicts. Like the R2000, the MIPS had no condition code register, and a special HI/LO multiply and divide register pair. Unlike the R2000, the MIPS had only 16 registers, and two delay slots for load, store and branch instructions. The PC and last three PC values were tracked for exception handling. Being experimental, there was no support for floating point operations [2].

2.2 MIPS R2000

Some of the MIPS developer from Stanford funded the MIPS Computer Systems [3]. The MIPS R2000 was released in 1988, and was one of the first RISC chips designed. It offered a very clean instruction set and a "virtual machine" programming mode and automatic pipelining [2]. The idea behind the MIPS was to simplify processor design by eliminating hardware interlocks between the five pipeline stages. This means that only single execution cycle instructions can access the thirty-two 32-bit general registers, so that the compiler can schedule them to avoid conflicts. This also means that load, store and branch instructions have a 1-cycle delay to account for. However, because of the importance of multiply and divide instructions, a special HI/LO pair of multiply/divide registers exist which do have hardware interlocks, since these take several cycles to execute and produce scheduling difficulties. The R2000 has no condition code register (where conditions from the ALU are stored) considering it a potential bottleneck. The PC is user visible. The CPU includes a Memory Manager Unit (MMU) that also can control a cache. The CPU can operate as a big or little endian processor. An FPU, the R2010, was also specified for the processor. Newer versions include the R3000, with improved cache control, and the R4000, which was expanded to 64 bits, and has more pipeline stages for a higher clock rate and performance [2]. In 1992 Silicon Graphics bought MIPS Computer Systems and today their processors are embedded in many applications.

2.3 MYRISC

The processor designed in this project was based on the MIPS R2000. The objective of the design was to be able to run all instructions in one cycle. This was achieved, but the slow memory on the prototype board needed two waitstates, which inferred three cycles per instruction. Most of the MIPS R2000 instructions are supported, except multiply, division and floating point instructions. See Appendix A for a list of supported instructions. There are three instruction formats (table 2.1).

Field size	**6 bits**	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**	**Comment**
R-format	Op	Rs	Rt	Rd	ShAmt	Funct	Arithmetic instruction format
I-format	Op	Rs	Rt	Address / Immediate			Branch, imm. format
J-format	Op	Target address					Jump instruction format

Table 2.1 The instruction formats.

Where the Op field is the extended instruction opcode. The Rs, Rt and Rd fields are the register fields. The ShAmt field is the shift amount field, used by static shift instructions. The Funct field is the instruction opcode field. The Address / Immediate field is used by branch and immediate instructions. The Target address field is used by jump instructions.

3 HARDWARE DESIGN

3.1 PIPELINING

One of the most effective ways to speed up a digital design is to use pipelining. The processor can be divided into subparts, where each part may execute in one clock cycle. This implies that it is possible to increase the clock frequency compared to a non-pipelined design. It will also be easier to optimise each stage than trying to optimise the whole design. While the instruction throughput increases, instruction latency is added. MYRISC is using a pipeline with 5 stages (figure 3.1):

1. **Instruction Fetch**, instructions are fetched from the instruction memory.
2. **Instruction Decode**, instructions are decoded and control signals are generated.
3. **Execute**, arithmetic and logic instructions are executed.
4. **Memory access**, memory is accessed on load and store instructions.
5. **Write back**, the result is written back to the appropriate register.

3.1.1 Pipeline hazards

In some cases the next instruction cannot execute in the following clock cycle. These events are called *hazards* [1]. In this design there are three types of hazards.

3.1.1.1 Structural hazards

Though the MIPS instruction set was designed to be pipelined, it does not solve the structural limitation of the design. If only one memory is used it will be impossible to solve a store or load instruction without stalling the pipeline. This is because a new instruction is fetched from the memory every clock cycle, and it is not possible to access the memory twice during a clock cycle.

3.1.1.2 Control hazards

Control hazards arise from the need to make a decision based on the results of one instruction while others are executing. This applies to the branch instruction. If it is not possible to solve the branch in the second stage, we will need to stall the pipeline. One solution to this problem is branch prediction, where one actually guess, based on statistics, if a branch is to be taken or not. In the MIPS architecture *delayed decision* was used [1]. A delayed branch always executes the next sequential instruction following the branch instruction. This is normally solved by the assembler, which will rearrange the code and insert an instruction that is not affected by the branch. The assembler made for this project does not support code reordering, it has to be done manually.

3.1.1.3 Data Hazards

If an instruction depends on the result of a previous instruction still in the pipeline, we will have a data hazard. These dependencies are too common to expect the compilers to be able avoid this problem. A solution is to get the result from the pipeline before it reaches the write back stage. This solution is called *forwarding* or *bypassing* [1].

3.1.1.4 Dealing with the hazards

1. Using two memories solves the structural hazard. One for instructions and one for data. Normally only one memory is used in a system. In that case separate instruction and data caches can be used to solve the structural hazard. In this project only one memory was available and because no caches were implemented, the processor is stalled for each load and store instruction.

2. Using delayed decision solves the control hazards.

3. Forwarding solves the data hazards. Still it will not be possible to combine a load instruction and an instruction that reads its result. This is due to the pipeline design and a hazard detection unit will stall the pipeline one cycle.

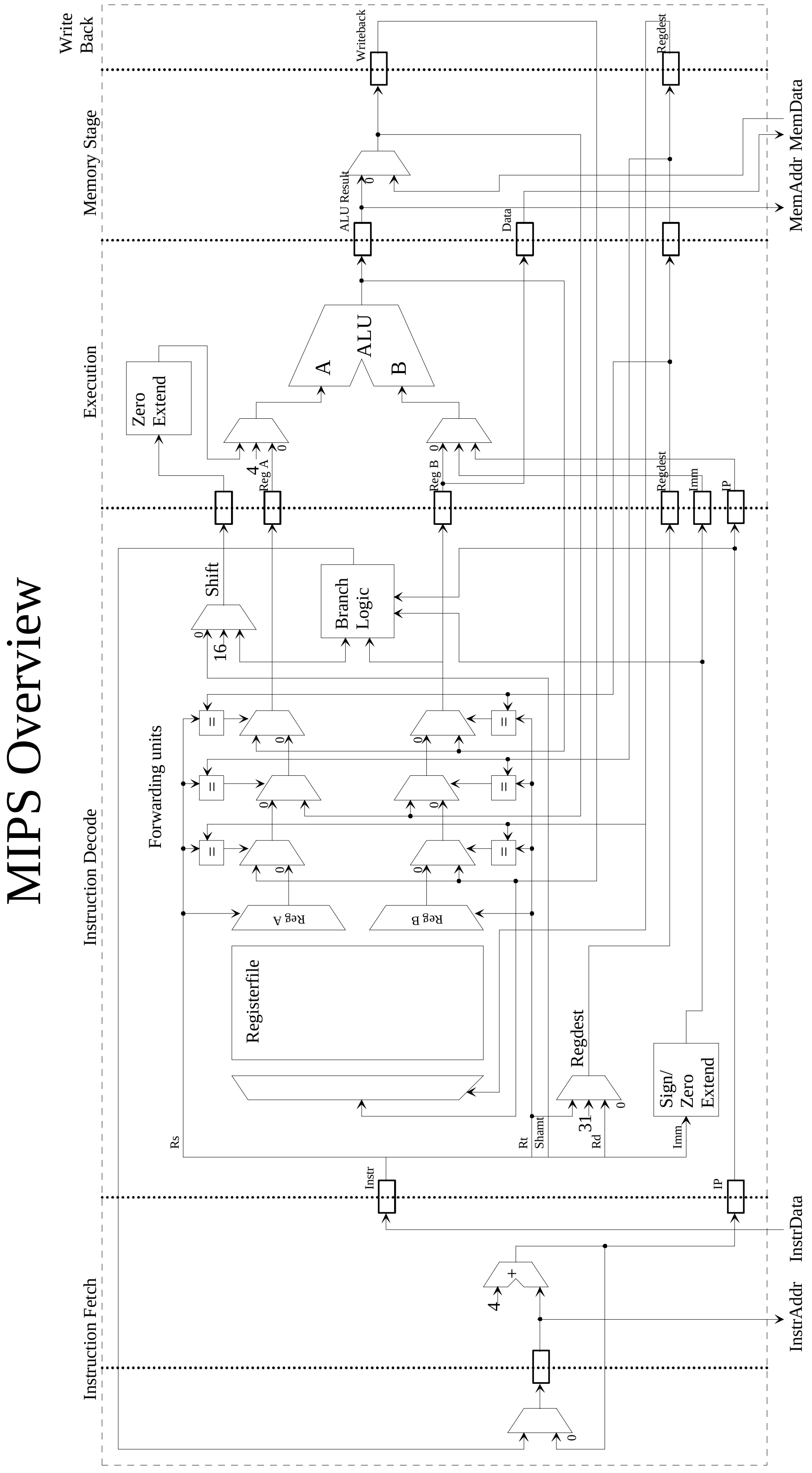

Figure 3.1 Overview over the MYRISC with the five pipline stages.

3.2 DESIGN PARTITIONING

The processor was partitioned into 6 subparts (figure 3.2), where each part was tested individually before merging them all together into a functional unit. The VHDL code can be found in Appendix B. The VHDL simulation was done in Modelsim from Model Technology. The simulation stage of the design process was crucial in order to achieve a functional unit. The whole system could be simulated with a program written in the MIPS assembler developed for this project.

3.2.1 Control Unit

The control unit creates control signals to all parts of the processor (table 3.1).

Signal name	**Effect when deasserted**	**Effect when asserted**
Ctrl_IF.Branch	Ip = Ip + 4	Branch to new address if (registers are equal) xor (bne)
Ctrl_IF.Jump	Ip = Ip + 4	Jump to new address
Ctrl_IF.bne	Branch only on equal registers	Branch only on non equal registers
Ctrl_ID.Branch	Branch target = jump address	Branch target = branch address
Ctrl_ID.Jr	Jumpaddress = Immediate 28 bits.	Jumpaddress = Rs
Ctrl_ID.Lui	Shiftvalue = Shift Amount	Shiftvalue = 16
Ctrl_ID.ShiftVar	Shiftvalue = Shift Amount	Shiftvalue = Rs(4 downto 0)
Ctrl_ID.ZeroExtend	Signextend imm value	Zeroextend imm value
Ctrl_Ex.ImmSel	ALU_B = Rt	ALU_B = imm value
Ctrl_Ex.JumpLink	Nothing	DestReg=31, WriteData=IP+1
Ctrl_Ex.ShiftSel	ALU_A=Rs	ALU_A = shiftvalue
Ctrl_Ex.OP	ALU opcode	
Ctrl_Mem.MemRead	Nothing	Read from datamemory
Ctrl_Mem.MemWrite	Nothing	Write to datamemory
Ctrl_Mem.MemBusAccess_n	Nothing	Memory stage is active
Ctrl_WB.RegWrite	Nothing	Write to registerfile

Table 3.1. Signals from the control unit.

3.2.2 Internal Bus State Machine

To be able to use a common memory for program and data, an internal bus state machine was inferred. When the Memory Access stage in the pipeline requests a memory access, the Internal Bus State Machine will stall the Instruction Fetch stage, and the Memory Access stage will be granted memory access (figure 3.2). The Hold_n signal allows waitstates to be inserted. Hold_n must be deasserted one cycle before the processor continues to operate.

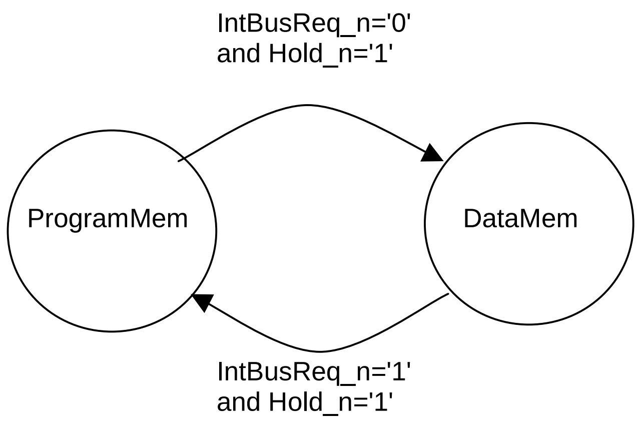

Figure 3.2. Internal Bus Statemachine

3.2.3 Instruction Fetch

The first stage in the pipeline is the Instruction Fetch. Instructions will be fetched from the memory and the Instruction Pointer (IP) will be updated.

3.2.4 Instruction Decode

The Instruction Decode stage is the second in the pipeline. Branch targets will be calculated here and the Register File, the dual-port memory containing the register values, resides in this stage. The forwarding units, solving the data hazards in the pipeline, reside here. Their function is to detect if the register to be fetched in this stage is written to in a later stage. In that case the data is forward to this stage and the data hazard is solved.

3.2.5 Execution

The third stage in the pipeline is where the arithmetic- and logic-instructions will be executed. All instructions are executed with 32-bit operands and the result is a 32-bit word. An overflow event handler was not included in this project.

3.2.6 Memory Access

The Memory Access stage is the fourth stage of the pipeline. This is where load and store instructions will access data memory.

Myrisc

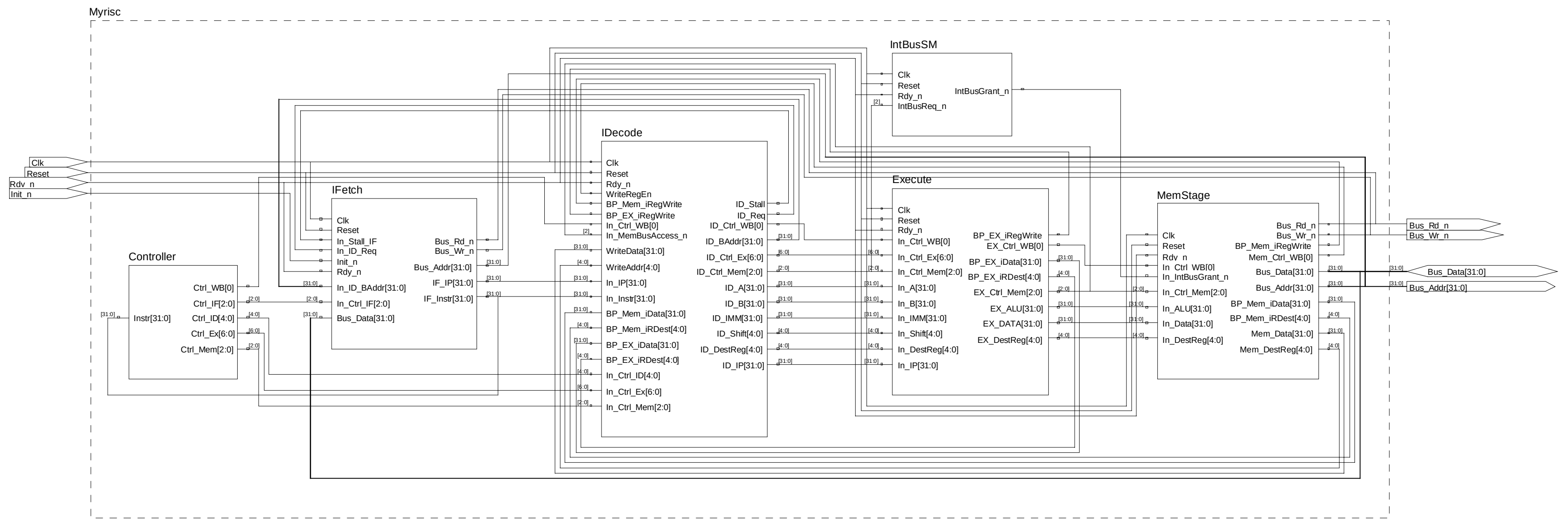

Figure 3.2 The processor design.

4 IMPLEMENTATION

4.1 FIELD PROGRAMMABLE GATE ARRAYS

Field Programmable Gate Array is a family of programmable logic devices that are common in prototyping and small series. Because they are re-programmable they can be used to prototype and verify designs before investing in technologies with high start-up costs, like full custom.

The project was targeted to a XILINX XC4036XLA with speed grade –9. It had 36000 equivalent gates and was placed on a Wildone PCI prototype board from Annapolis Micro Systems that resided in a PC.

4.1.1 XC4000 Architecture

The FPGA consists of several Configurable-Logic-Blocks (CLB), which are interconnected with each other via routing channels, and to the input/output blocks (figure 4.1). The interconnections are configured with a switch matrix based on SRAM technology. The FPGA is configured on power up or when the /program pin is cycled.

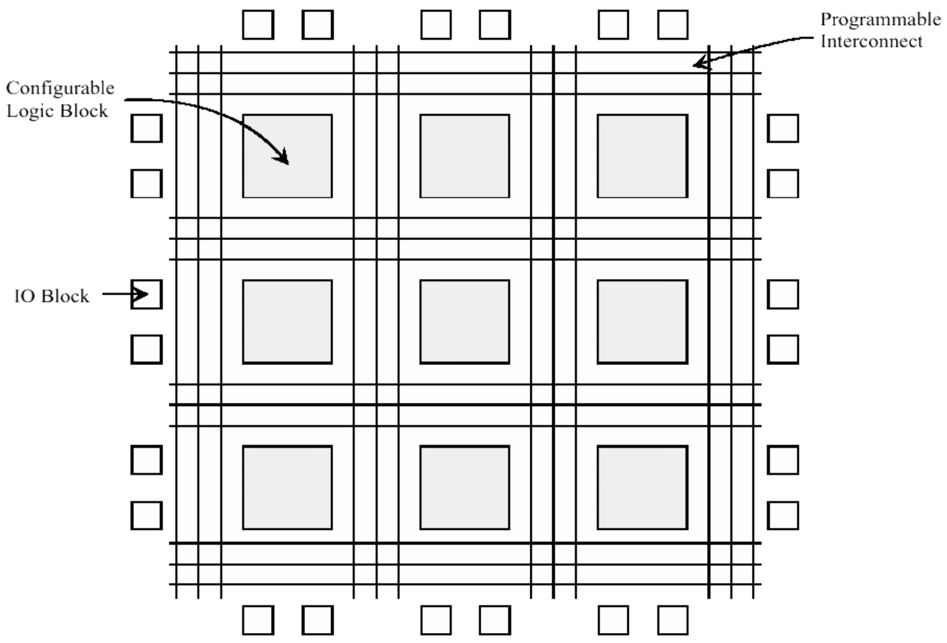

Figure 4.1. Generic FPGA structure [6].

4.1.1.1 Configurable Logic Blocks

Configurable Logic Blocks (CLB) realise most of the logic in an FPGA (figure 4.2). One of the basic building blocks in an FPGA is the lookup table (LUT). It is basically a 2^n-bit memory that can realise any logic function with n inputs by programming the lookup table with the appropriate bits [4]. A CLB consists of two 4-input LUTs, one 3-input LUT and two storage elements (configured as flip-flops or latches). A CLB can be configured to implement any of the following functions [6]:

1. Any function of up to four variables, plus any second function of up to four unrelated variables, plus any third function of up to three unrelated variables.

2. Any single function of five variables.

3. Any function of four variables together with some functions of six variables.

4. Some functions of up to nine variables.

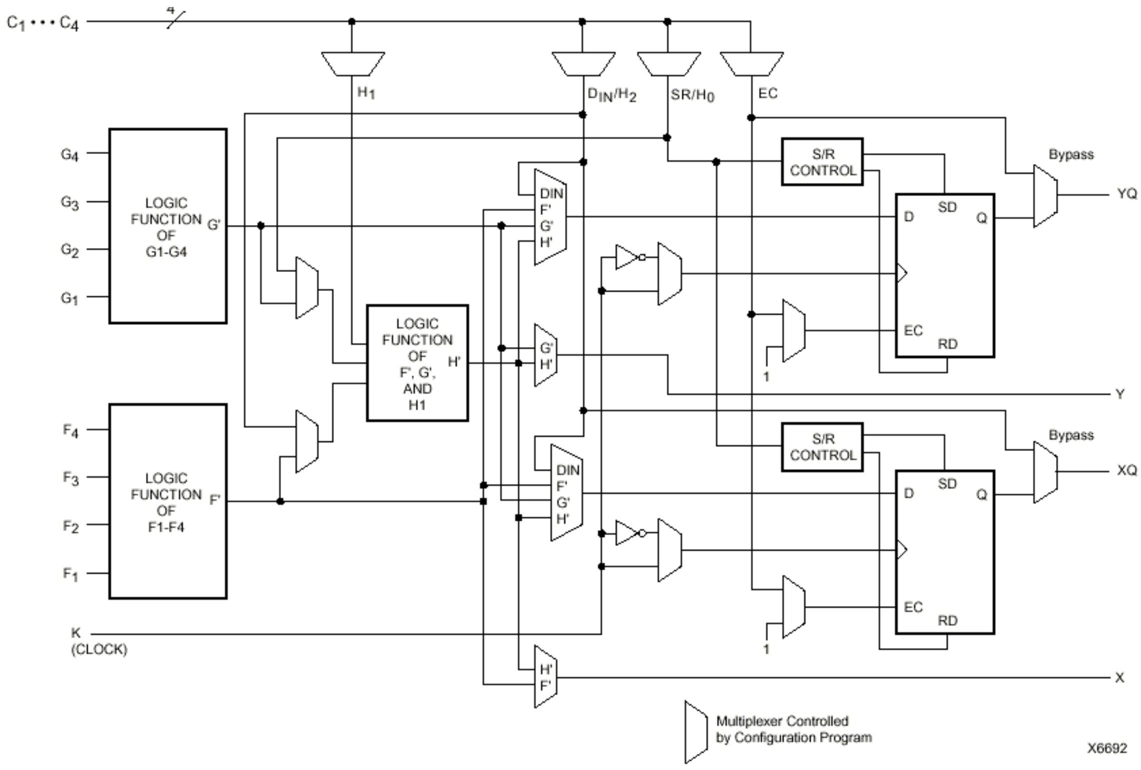

Figure 4.2. Principal view of a CLB. (RAM and Carry Logic functions not shown)

4.1.1.2 RAM

A CLB can be configured to implement a high speed RAM. The dual-port memory in the register file of the processor was realised as a 2x32x32 bit edge-trigged single-port RAM (figure 4.3).

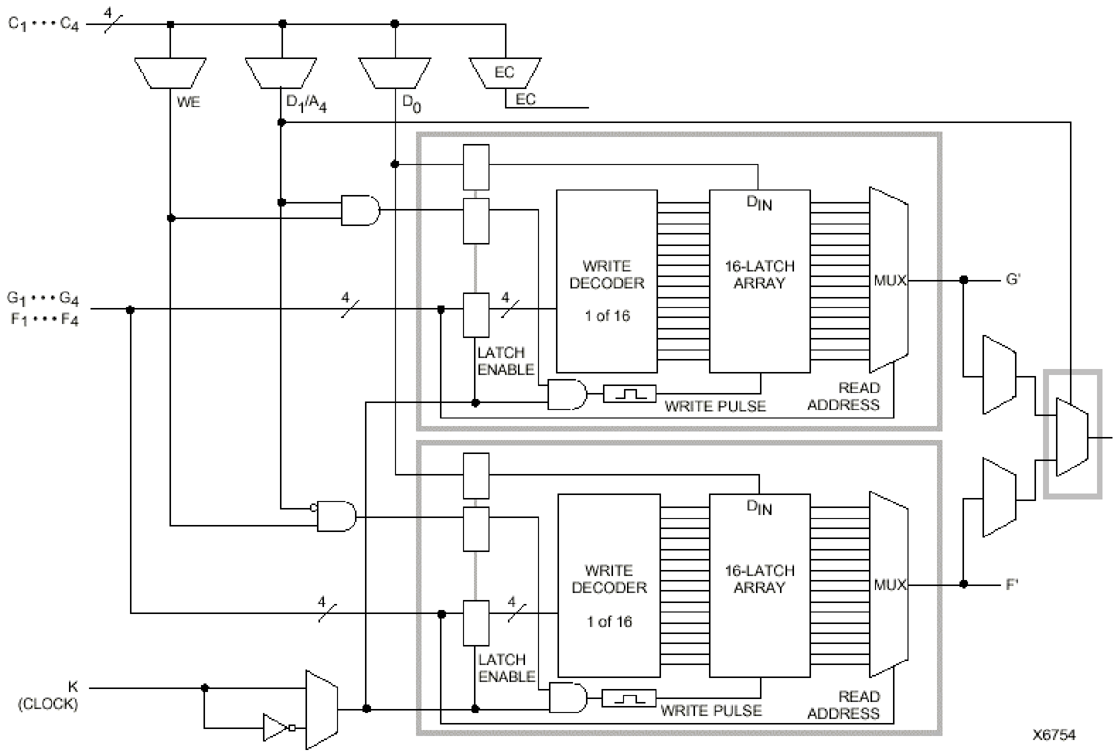

Figure 4.3. 32x1 Edge-Triggered Single-Port RAM (F and G addresses are identical)

4.1.1.3 Input/Output Blocks (IOBs)

The IOBs provide the interface between external package pins and internal logic (figure 4.4). Each IOB controls one package pin and can be configured for input, output or bidirectional signals [5].

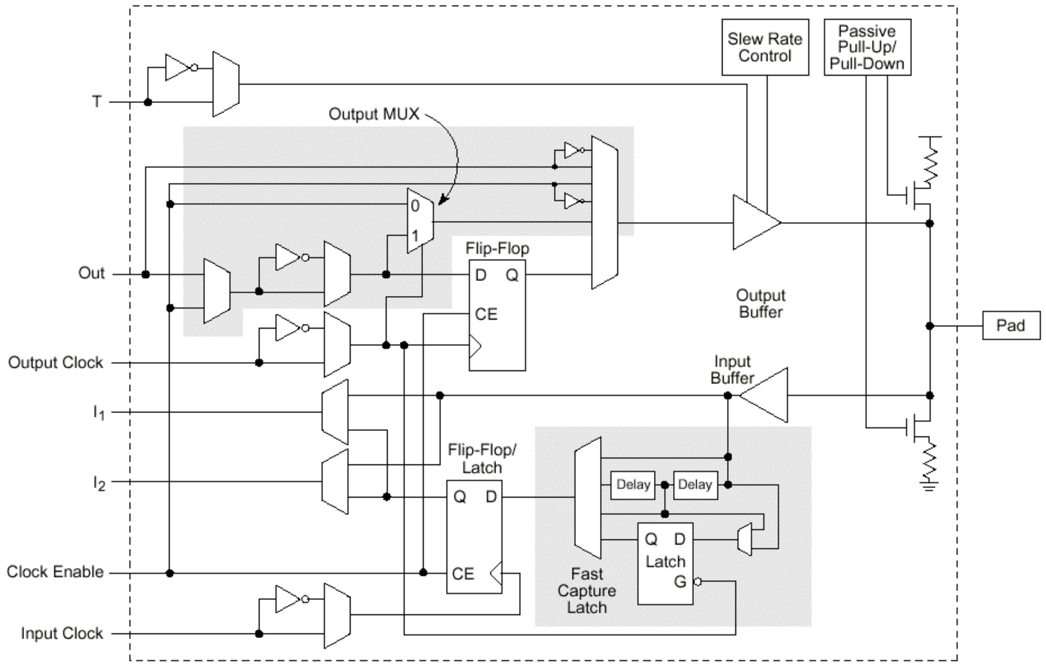

Figure 4.4. Simplified block diagram of XC4000X IOB (shaded areas indicate differences from XC4000E)

4.1.2 Three-state buffers

Each CLB has two three-state buffers connected to the longlines next to the CLB (figure 4.5). The three-state buffers have three modes [5]:

1. Standard three-state buffer.
2. Wired AND with input on the I pin.
3. Wired OR-AND.

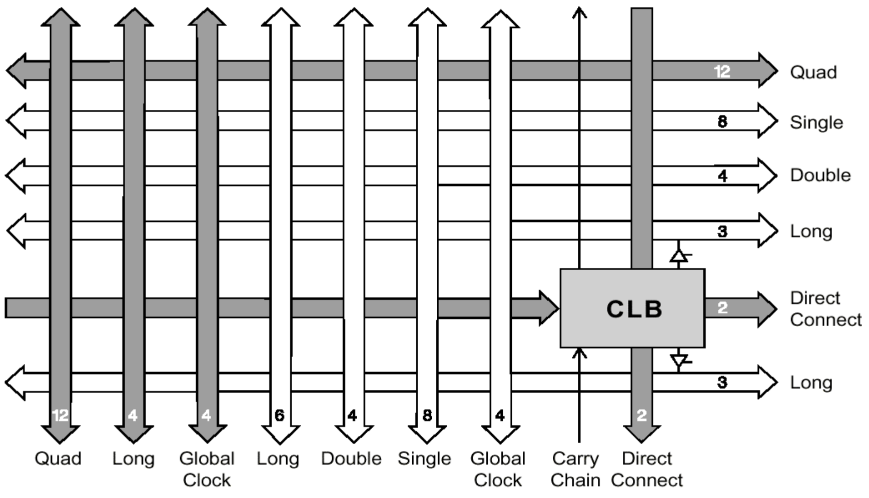

Figure 4.5. High level routing diagram of XC4000 series FPGA.

4.1.2.1 Wide edge decoders

When the address or data field is wider than the function generator inputs (i.e. more than nine), FPGAs need multiple level decoding and are thus slower than PALs. The XC4000 series FPGA has four wide edge decoders, one on each side of the device, in order to boost performance of the wide decoding functions [5].

4.1.2.2 Programmable interconnect

All internal connections are composed of metal segments. Programmable switching points and switching matrices are used to implement the routing. The relative length of the segments distinguishes five types: single-, double-, quad- and octal-length lines and longlines (figure 4.5).

4.2 SYNTHESISE, PLACE AND ROUTE

After the design passed the simulation, it was synthesised with Synplify from Synplicity. The synthesis process converts the VHDL code into a netlist of target specific components. The netlist was then used as the input for the Place and Route (PAR) utilities from XILINX. The output from the PAR process was a binary file that was used to configure the FPGA.

The design used 1035 CLBs out of 1296 (79%). Total equivalent gate count for the design was 32362 gates. The static timing analysis set the maximum frequency to 10 MHz. It seems like the branch logic is the critical part of the processor. A further optimising there could speed up the design.

5 MIPS ASSEMBLER

To facilitate program development a MIPS assembler was developed. It was written in PERL, a scripting language that is excellent for text parsing applications. The program was divided into these stages:

1. Read source file.
2. Replace pseudo instructions with real instructions.
3. Look up labels in the source code.
4. Convert instructions to machine code.
5. Write resulting machine codes.

Note: The assembler does not support instruction reordering, so one has to take the branch slot in consideration when programming.

5.1 COMMAND LINE

The MIPS assembler is to be run from the command line. Usage:

```
asm.pl assemblerfile.asm program.mem
```

Where assemblerfile.asm is the assembler program and program.mem is the outfile that will contain a memory image of the compiled program.

5.2 TABS AND SPACES

Tabs and spaces may be used freely to format the program.

5.3 CASE SENSITIVITY

The assembler is case sensitive on labels, but not on instructions.

5.4 CONSTANTS

Constants can be written both in decimal and hexadecimal. A hexadecimal value must be proceeded by 0x, e.g. 0xFF is interpreted as 255 decimal.

5.5 REGISTERS

Only numbered registers are supported. Registers are written on the form $RegisterNumber, e.g. $1 indicates register 1.

Note: Register 0 is hardwired to a value of zero.

5.6 LABELS

Labels can be any alphanumeric combination. The underscore (“_”) character can be used as a separator. They are case sensitive and are written on the form: WhatEverLabel:.

5.7 INSTRUCTIONS

Instructions can be written on two forms:

1. move $1, $2, $3

2. move $1 $2 $3

Supported instructions can be found in appendix A.

5.8 COMMENTS

Any characters specified after a semicolon (“;”) or a number sign (“#”) until the end of the line are considered as comments.

5.9 DIRECTIVES

Assembler directives control the assembling process. Supported assembler directives are:

5.9.1 .org [address]

The .org directive redirects the current address to its parameter.

e.g. .org 0x1000 will place the following code or data on the address 0x1000 in memory.

5.9.2 .text

The .text directive will place the following code in the code segment.

5.9.3 .data

The .data directive will place the following data in the data segment.

5.9.4 .const

The .const directive will place the follwing constants in the const segment.

5.9.5 .space [amount]

The .space directive will insert an amount of spaces. This is used to allocate static buffers.

5.9.6 .word [data{,data}]

The .word directive will insert one or more 32-bit constants into the memory.

5.10 PROGRAM EXAMPLE

A program example that will multiply two 16-bit values and return a 32-bit result is presented here:

```
#
# Example program that multiplies two 16-bit
# integers to a 32-bit result.
#
.text
      ori   $1, $0, 578 ;Multiplicand
      ori   $2, $0, 345 ;Multiplier
      jal   mult
      nop

end:  j     end
      nop

mult:
      ori   $9, $0, 15
      move  $10,$0
m_Loop:
      andi  $3, $2, 1
      beq   $3, $0, m_NoAdd1
      srl   $2, $2, 1
      addu  $10,$10,$1
m_NoAdd1:
      sll   $1, $1, 1
      bne   $9, $0,  m_Loop
      addiu $9, $9, -1

      jr    $31
      nop
```

6 CONCLUSION

By using VHDL in digital design it is possible to use a high level of abstraction in the design. This lets you put more effort on the functionality of the circuit. Another aspect of VHDL is that the design will be self-documented. I have for a long time been interested in low-level programming of microprocessors. During the design of the MIPS I learned a lot on processor design that will be useful in the future when optimising time-critical programs. I have also learned a lot on digital design in general.

7 TERMINOLOGY

ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machines
Big Endian	MSB is stored first in memory
CLB	Configurable Logic Block
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IP	Instruction Pointer, same as PC
Little Endian	LSB is stored first in memory
LUT	Look-Up Table
LSB	Least significant bit
MIPS	Microprocessor without Interlocked Pipline Stages
MMU	Memory Manager Unit
MSB	Most significant bit
PC	Program Counter, same as IP
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SRAM	Static RAM
SPARC	A RISC developed by Sun Microsystems, Inc.
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuit

8 REFERENCES

[1] Computer Organization & Design. David A. Patterson and John L. Hennessy, ISBN 1-55860-428-6, p 476-501, 525-256.

[2] University of California at Davis Computer Science Museum, http://wwwcsif.cs.ucdavis.edu/~csclub/museum/cpu.html, October 1999.

[3] Advanced Microprocessors, Daniel Tabak, ISBN 0-07-062843-2, p 79-99.

[4] The Practical XILINX Designer Lab Book, Dave Van den Bout, ISBN 0-13-095502-7, p 30-31.

[5] XILINX datasheet library, http://www.xilinx.com/partinfo/4000.pdf, November 1999.

[6] Evaluation of a reconfigurable computing engine for digital communication applications, Jonas Thor, ISSN 1402-1617, p 12-17.

APPENDIX A

I. INSTRUCTION SET

The instruction set in the MYRISC is a subset of the MIPS R2000 instruction set. Most of the arithmetic instructions are supported. Multiplication, division and floating point operations are not supported.

Instruction	**Function**
add rd, rs, rt	rd := rs + rt, overflow will trap
addi rd, rs, rt	rt := rs + SX(expr), overflow will trap
addu rd, rs, rt	rd := rs + rt, no trap on overflow
addiu rd, rs, rt	rt := rs + SX(expr), no trap on overflow
and rd, rs, rt	rd := rs & rt
andi rd, rs, rt	rt := rs & ZX(expr)
beq rs, rt, target	branch if contents rs equals rt's
bne rs, rt, target	branch if contents rs not equal to rt's
j target	unconditionally jump to target
jal target	unconditionally jump to target, link to reg 31
jr rs	unconditionally jump to address in rs
lui rt, expr	rt := (expr shifted left 16 bits)
lw rt, addr	load 32-bit word, word-aligned
nop	no operation
nor rd, rs, rt	rd := rs nor rt
or rs, rs, rt	rd := rs | rt
ori rt, rs, expr	rt := rs | ZX(expr)
sll rd, rt, expr	rd := rt logical-shifted-left expr bits
sllv rd, rt, expr	rd := rt logical-shifted-left by value in rs
slt rd, rs, rt	rd := rs < rt, signed
slti rd, rs, rt	rt := rs < SX(expr), signed
sltu rd, rs, rt	rd := rs < rt, unsigned
sltiu rd, rs, rt	rt := rs < ZX(expr), unsigned
sra rd, rt, expr	rd := rt arithmetic-shifted-right expr bits
srav rd, rt, expr	rd := rt arithmetic-rightshifted by val in rs
srl rd, rt, expr	rd := rt logical-shifted-right expr bits
srlv rd, rt, expr	rd := rt logical-rightshifted by val in rs
sub rd, rs, rt	rd := rs – rt, overflow will trap
subu rd, rs, rt	rd := rs – rt, no trap on overflow
sw rt, addr	store 32-bit word, word-aligned
xor rd, rs, rt	rd := rs ^ rt
xori rd, rs, expr	rt := rs ^ ZX(expr)

Note: Overflow is not implemented in this version.

II. PSEUDO INSTRUCTION SET

Pseudo instruction are those that are converted into real instruction by the assembler. They exist in order to facilitate program development.

Pseudo Instruction	**Function**
beqz rs, target	branch if contents rs equals zero
bneqz rs, target	branch if contents rs not equals zero
la rd, target	rd := address of target
li rd, expr	rd := expr
move rd, rs	rd := rs
neg rd, rs	rt := neg(rs)
not rd, rs	rd := not(rs)

APPENDIX B

The VHDL code of the MIPS is presented here.

I. MYRISC VHDL CODE

```
-------------------------------------------------------------------------------
--
--  MYRISC project in SME052 1999-12-19
--
--  Anders Wallander
--  Department of Computer Science and Electrical Engineering
--  Luleå University of Technology
--
--  A VHDL implementation of a MIPS, based on the MIPS R2000 and the
--  processor described in Computer Organization and Design by
--  Patterson/Hennessy
--
--
--  MYRISC toplevel
--
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.RiscPackage.all;

entity myrisc is
  port (Clk         : in std_logic;
        Reset       : in std_logic;
        Bus_Data    : inout TypeWord;
        Bus_Addr    : out TypeWord;
        Bus_Rd_n    : out std_logic;
        Bus_Wr_n    : out std_logic;
        Init_n      : in std_logic;
        Rdy_n       : in  std_logic);
end myrisc;

architecture behavior of myrisc is

  component Controller is
  port (Instr     : in TypeWord;
        Ctrl_IF   : out TypeIFCtrl;
        Ctrl_ID   : out TypeIDCtrl;
        Ctrl_Ex   : out TypeExCtrl;
        Ctrl_Mem  : out TypeMemCtrl;
        Ctrl_WB   : out TypeWBCtrl
     );
  end component;

  component IntBusSM is
   port (
      Clk           : in  std_logic;
      Reset         : in  std_logic;
      Rdy_n         : in std_logic;
      IntBusReq_n   : in std_logic;   -- Memory request signal
      IntBusGrant_n : out  std_logic  -- Memory grant signal
      );
  end component IntBusSM;

  component ifetch is
    port (
      Clk           : in  std_logic;
      Reset         : in  std_logic;
      In_Stall_IF   : in  std_logic;    -- Asserted if pipe line is stalled or memstage memaccess
      In_ID_Req     : in  std_logic;    -- Asserted if RegA equals RegB from the registerfile
(Including bypass...)
      In_ID_BAddr   : in  TypeWord;     -- Jump/Branch target address
      In_Ctrl_IF    : in  TypeIFCtrl;
      Init_n        : in std_logic;
      Rdy_n         : in std_logic;
      Bus_Data      : inout TypeWord;
      Bus_Addr      : out TypeWord;
      Bus_Rd_n      : out std_logic;
      Bus_Wr_n      : out std_logic;
      IF_IP         : out TypeWord;
      IF_Instr      : out TypeWord);
  end component ifetch;

  component IDecode is
    port (Clk         : in std_logic;
          Reset       : in std_logic;
          Rdy_n       : in  std_logic;
          WriteRegEn  : in std_logic;
          WriteData   : in TypeWord;
          WriteAddr   : in TypeRegister;
          In_IP       : in TypeWord;
          In_Instr    : in TypeWord;

          BP_Mem_iData    : in  TypeWord;     -- Bypass from memstage
        BP_Mem_iRDest   : in  TypeRegister; -- Bypass from memstage
        BP_Mem_iRegWrite: in std_logic;     -- Bypass from memstage
        BP_EX_iData     : in  TypeWord;     -- Bypass from execution
        BP_EX_iRDest    : in  TypeRegister; -- Bypass from execution
        BP_EX_iRegWrite : in std_logic;     -- Bypass from execution

        In_Ctrl_ID  : in  TypeIDCtrl;
        In_Ctrl_Ex  : in  TypeExCtrl;
        In_Ctrl_Mem : in  TypeMemCtrl;
        In_Ctrl_WB  : in  TypeWBCtrl;
        In_MemBusAccess_n:  in  std_logic;

        ID_Stall  : out std_logic;
        ID_Req    : out std_logic;
        ID_BAddr  : out TypeWord;

        ID_Ctrl_Ex  : out TypeExCtrl;
        ID_Ctrl_Mem : out TypeMemCtrl;
        ID_Ctrl_WB  : out TypeWBCtrl;
        ID_A        : out TypeWord;
        ID_B        : out TypeWord;
        ID_IMM      : out TypeWord;
        ID_Shift    : out TypeRegister;
        ID_DestReg  : out TypeRegister;
        ID_IP       : out TypeWord
     );
end component;

component Execute is
  port (Clk         : in std_logic;
        Reset       : in std_logic;
        Rdy_n       : in  std_logic;
        In_Ctrl_Ex  : in TypeExCtrl;
        In_Ctrl_Mem : in TypeMEMCtrl;
        In_Ctrl_WB  : in TypeWBCtrl;
        In_A        : in TypeWord;
        In_B        : in TypeWord;
        In_IMM      : in TypeWord;
        In_Shift    : in TypeRegister;
        In_DestReg  : in TypeRegister;
        In_IP       : in TypeWord;

        BP_EX_iData     : out TypeWord;      -- Bypass to idecode
        BP_EX_iRDest    : out TypeRegister;   -- Bypass to idecode
        BP_EX_iRegWrite : out std_logic;  -- Bypass to idecode

        EX_Ctrl_WB  : out TypeWBCtrl;
        EX_Ctrl_Mem : out TypeMemCtrl;
        EX_ALU      : out TypeWord;
        EX_DATA     : out TypeWord;
        EX_DestReg  : out TypeRegister
        );
end component;

component MemStage is
  port (Clk            : in std_logic;
        Reset          : in std_logic;
        Rdy_n          : in  std_logic;
        In_Ctrl_WB     : in TypeWBCtrl;
        In_Ctrl_Mem    : in TypeMEMCtrl;
        In_ALU         : in TypeWord;
        In_Data        : in TypeWord;
        In_DestReg     : in TypeRegister;
        In_IntBusGrant_n: in  std_logic;

        Bus_Data    : inout TypeWord;
        Bus_Addr    : out TypeWord;
        Bus_Rd_n    : out std_logic;
        Bus_Wr_n    : out std_logic;

        BP_Mem_iData  : out TypeWord;     -- Bypass to idecode
        BP_Mem_iRDest : out TypeRegister; -- Bypass to idecode
        BP_Mem_iRegWrite: out std_logic;  -- Bypass to idecode

        Mem_Ctrl_WB : out TypeWBCtrl;
        Mem_Data    : out TypeWord;
        Mem_DestReg : out TypeRegister
        );
end component;

-- Controller
signal  ctrl_IF : TypeIFCtrl;
signal  ctrl_ID : TypeIDCtrl;
signal  ctrl_Ex : TypeExCtrl;
signal  ctrl_Mem: TypeMemCtrl;
signal  ctrl_WB : TypeWBCtrl;

-- IntBusSM
```

```
  signal  ib_IntBusGrant_n  :  std_logic;  -- Bus grant signal

  -- ifetch
  signal IF_IP    : TypeWord;
  signal IF_Instr : TypeWord := (others => '0'); -- Avoid metastability during simulation

  -- iDecode
  signal ID_Stall   : std_logic;
  signal ID_Req     : std_logic;
  signal ID_BAddr   : TypeWord;
  signal ID_Ctrl_Ex : TypeExCtrl;
  signal ID_Ctrl_Mem: TypeMemCtrl;
  signal ID_Ctrl_WB : TypeWBCtrl;
  signal ID_A       : TypeWord;
  signal ID_B       : TypeWord;
  signal ID_IMM     : TypeWord;
  signal ID_Shift   : TypeRegister;
  signal ID_DestReg : TypeRegister;
  signal ID_IP      : TypeWord;

  -- Execute
  signal BP_EX_iData: TypeWord;     -- Bypass to idecode
  signal BP_EX_iRDest:TypeRegister; -- Bypass to idecode
  signal BP_EX_iRegWrite: std_logic;  -- Bypass to idecode
  signal BP_Mem_iData:  TypeWord;     -- Bypass to idecode
  signal BP_Mem_iRDest:TypeRegister;  -- Bypass to idecode
  signal BP_Mem_iRegWrite:  std_logic;  -- Bypass to idecode

  signal EX_Ctrl_Mem: TypeMemCtrl;
  signal EX_Ctrl_WB : TypeWBCtrl;
  signal EX_ALU     : TypeWord;
  signal EX_DATA    : TypeWord;
  signal EX_DestReg : TypeRegister := (others => '0');

  -- Memstage
  signal Mem_Ctrl_WB:   TypeWBCtrl;
  signal Mem_Data   :   TypeWord := (others => '0');
  signal Mem_DestReg:   TypeRegister := (others => '0'); -- Avoid metastability

begin

  controller1: controller
    port map (Instr =>  IF_Instr,
              Ctrl_IF =>  ctrl_IF,
              Ctrl_ID =>  ctrl_ID,
              Ctrl_Ex =>  ctrl_Ex,
              Ctrl_Mem => ctrl_Mem,
              Ctrl_WB =>  ctrl_WB);

  intBusSM1: IntBusSM
    port map (
      Clk => Clk,
      Reset => Reset,
      Rdy_n =>  Rdy_n,
      IntBusReq_n => ID_Ctrl_Mem.MemBusAccess_n,    -- Internal bus request signal
      IntBusGrant_n => ib_IntBusGrant_n);

  ifetch1: ifetch
    port map (Clk => Clk,
              Reset => reset,
              In_Stall_IF =>  ID_Stall,
              In_ID_Req   =>  ID_Req,
              In_ID_BAddr =>  ID_Baddr,
              In_Ctrl_IF  =>  ctrl_IF,
              Init_n      =>  Init_n,
              Rdy_n     =>  Rdy_n,
              Bus_Data  =>  Bus_Data,
              Bus_Addr  =>  Bus_Addr,
              Bus_Rd_n  =>  Bus_Rd_n,
              Bus_Wr_n  =>  Bus_Wr_n,
              IF_IP => if_ip,
              IF_Instr => if_instr);

  IDecode1: IDecode
    port map (Clk => Clk,
              Reset => reset,
              Rdy_n =>  Rdy_n,
              WriteRegEn => Mem_Ctrl_WB.RegWrite,
              WriteData => Mem_Data,
              WriteAddr => Mem_DestReg,
              In_IP     => IF_IP,
              In_Instr  => IF_Instr,

              BP_Mem_iData => BP_Mem_iData,         -- Bypass from memstage
              BP_Mem_iRDest => BP_Mem_iRDest,       -- Bypass from memstage
              BP_Mem_iRegWrite => BP_Mem_iRegWrite, -- Bypass from memstage
              BP_EX_iData => BP_EX_iData,           -- Bypass from execution
              BP_EX_iRDest => BP_EX_iRDest,         -- Bypass from execution
```

```
            BP_EX_iRegWrite => BP_EX_iRegWrite,   -- Bypass from execution

            In_Ctrl_ID  => ctrl_ID,
            In_Ctrl_Ex  => ctrl_Ex,
            In_Ctrl_Mem => ctrl_Mem,
            In_Ctrl_WB  => ctrl_WB,

            In_MemBusAccess_n => Ex_Ctrl_mem.MemBusAccess_n,

            ID_Stall  =>  ID_Stall,
            ID_Req    =>  ID_Req,
            ID_BAddr  =>  ID_BAddr,

            ID_Ctrl_Ex  =>  ID_Ctrl_Ex,
            ID_Ctrl_Mem =>  ID_Ctrl_Mem,
            ID_Ctrl_WB  =>  ID_Ctrl_WB,
            ID_A        =>  ID_A,
            ID_B        =>  ID_B,
            ID_IMM      =>  ID_IMM,
            ID_Shift    =>  ID_Shift,
            ID_DestReg  =>  ID_DestReg,
            ID_IP       =>  ID_IP);

execute1: execute
 port map ( Clk => Clk,
            Reset => reset,
            Rdy_n =>  Rdy_n,
            In_Ctrl_WB  => ID_Ctrl_WB,
            In_Ctrl_Mem => ID_Ctrl_Mem,
            In_Ctrl_Ex => ID_Ctrl_Ex,
            In_A => ID_A,
            In_B => ID_B,
            In_IMM => ID_IMM,
            In_Shift => ID_Shift,
            In_DestReg => ID_DestReg,
            In_IP =>  ID_IP,
            BP_EX_iData => BP_EX_iData,
            BP_EX_iRDest => BP_EX_iRDest,
            BP_EX_iRegWrite => BP_EX_iRegWrite,
            EX_Ctrl_Mem => EX_Ctrl_Mem,
            EX_Ctrl_WB => EX_Ctrl_WB,
            EX_ALU => EX_ALU,
            EX_DATA => EX_DATA,
            EX_DestReg => EX_DestReg
        );
memstage1:  memstage
  port map (Clk => Clk,
            Reset => reset,
            Rdy_n =>  Rdy_n,
            In_Ctrl_WB => EX_Ctrl_WB,
            In_Ctrl_Mem => EX_Ctrl_Mem,
            In_ALU => EX_ALU,
            In_Data => EX_Data,
            In_DestReg => EX_DestReg,

            In_IntBusGrant_n => ib_IntBusGrant_n,

            Bus_Data  =>  Bus_Data,
            Bus_Addr  =>  Bus_Addr,
            Bus_Rd_n  =>  Bus_Rd_n,
            Bus_Wr_n => Bus_Wr_n,
            BP_Mem_iData => BP_Mem_iData,
            BP_Mem_iRDest => BP_Mem_iRDest,
            BP_Mem_iRegWrite => BP_Mem_iRegWrite,
            Mem_Ctrl_WB => Mem_Ctrl_WB,
            Mem_Data => Mem_Data,
            Mem_DestReg => Mem_DestReg
          );
```

end behavior;

II. INTERNAL BUS STATEMACHINE VHDL CODE

```
------------------------------------------------------------------------------
--
--  MYRISC project in SME052 1999-12-19
--
--  Anders Wallander
--  Department of Computer Science and Electrical Engineering
--  Luleå University of Technology
--
--  A VHDL implementation of a MIPS, based on the MIPS R2000 and the
--  processor described in Computer Organization and Design by
--  Patterson/Hennessy
--
--
--  Internal Bus State Machine
--
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;


entity IntBusSM is
   port (
      Clk              : in  std_logic;
      Reset            : in  std_logic;
      Rdy_n          : in std_logic;
      IntBusReq_n      : in std_logic;    -- Memory request signal
      IntBusGrant_n    : out  std_logic  -- Memory grant signal
      );
end IntBusSM;

architecture RTL of IntBusSM is

type state_type is (ProgramMemAccess, DataMemAccess);

signal PresentState, NextState : state_type;

begin

  process( Clk, Reset)
  begin
    if Reset = '1' then
      PresentState <= ProgramMemAccess;
    elsif rising_edge(Clk) then
      PresentState <= NextState;
    end if;
  end process;

  process(PresentState, Rdy_n,  IntBusReq_n)

  begin
    case PresentState is
      when ProgramMemAccess =>

            IntBusGrant_n <=  '1';

            if (IntBusReq_n = '0' and Rdy_n = '1')  then
              NextState <= DataMemAccess;
            else
              NextState <= ProgramMemAccess;
            end if;

      when DataMemAccess =>
            IntBusGrant_n <= '0'
            if (IntBusReq_n = '1' and Rdy_n = '1')  then
              NextState <= ProgramMemAccess;
            else
              NextState <= DataMemAccess;
            end if;
      when others =>  NextState <= ProgramMemAccess;
    end case;
  end process;
end;
```

III. INSTRUCTION FETCH VHDL CODE

```
-------------------------------------------------------------------------------
--
--  MYRISC project in SME052 1999-12-19
--
--  Anders Wallander
--  Department of Computer Science and Electrical Engineering
--  Luleå University of Technology
--
--  A VHDL implementation of a MIPS, based on the MIPS R2000 and the
--  processor described in Computer Organization and Design by
--  Patterson/Hennessy
--
--
--  Instruction fetch stage
--    - Evaluate the program counter
--    - Fetch instructions
--
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library Work;
use Work.RiscPackage.all;

entity ifetch is

  port (
    Clk           : in  std_logic;
    Reset         : in  std_logic;
    In_Stall_IF   : in  std_logic;    -- Asserted if pipe line is stalled
    In_ID_Req     : in  std_logic;    -- Asserted if RegA equals RegB
                                      -- from the registerfile (Including bypass...)
    In_ID_BAddr   : in  TypeWord;     -- Jump/Branch target address
    In_Ctrl_IF    : in  TypeIFCtrl;
    Init_n        : in std_logic;
    Rdy_n       : in std_logic;
    Bus_Data      : inout TypeWord;
    Bus_Addr      : out TypeWord;
    Bus_Rd_n      : out std_logic;
    Bus_Wr_n      : out std_logic;
    IF_IP         : out TypeWord;
    IF_Instr      : out TypeWord);

end ifetch;

Architecture Struct of ifetch is

  signal nextPC : TypeWord;           -- Next PC
  signal intPC : TypeWord;            -- Internal PC
  signal intIncPC : TypeWord;         -- Internal incremented PC
  signal instrData_i: TypeWord;
  signal Stall : std_logic;
  signal TriStateBus : std_logic;

begin

  process (Clk, Reset)
  begin  -- process
    if Reset = '1' then
      intPC <= x"0000_0100"; --(others => '0');
    elsif rising_edge(Clk) then
      if Stall = '0' then
        intPC <= nextPC;
      end if;

      if Init_n = '0' then
        intPC <= x"0000_0100"; --(others => '0');
      end if;

    end if;
  end process;


  -- Multiplex next PC
  nextPC <= In_ID_BAddr when ( ( (In_ID_Req = '1' xor In_Ctrl_IF.bne = '1') and In_Ctrl_IF.Branch
= '1') or In_Ctrl_IF.Jump = '1') else
            intIncPC;

  -- Increment PC with 4
  intIncPC <= intPC + X"0000_0004";
```

```
--------------------------------------------------------------------------------------
-- Bus stuff
--------------------------------------------------------------------------------------

  -- Evaluate stall
  Stall <= In_Stall_IF or (not Rdy_n);
  TriStateBus <= In_Stall_IF or (not Init_n);

  -- Tristate address bus if stall
  Bus_Addr <= intPC when TriStateBus = '0' else
                (others => 'Z');

  -- Insert NOP instruction if stall...
  InstrData_i <= Bus_Data;

  -- Only read from bus when data memory is not accessed
  Bus_Rd_n <= '0' when TriStateBus = '0' else
        'Z';

  Bus_Wr_n <= '1' when TriStateBus = '0' else
        'Z';

  -- Always output tristate on data (never write...)
  Bus_Data <= (others => 'Z');


--------------------------------------------------------------------------------------
-- Shift pipeline
--------------------------------------------------------------------------------------
  pipeline : process(Clk, Reset)
  begin
    if reset = '1' then
      IF_IP <= x"0000_0100"; --(others => '0');
      IF_Instr <= (others => '0');
    elsif rising_edge(Clk) then
      if Stall = '0' then
        IF_IP <= intIncPC;
        IF_Instr <= InstrData_i;
      end if;

      if Init_n = '0' then
        IF_IP <= x"0000_0100"; --(others => '0');
        IF_Instr <= (others => '0');
      end if;

    end if;
  end process;


end architecture Struct;
```

IV. INSTRUCTION DECODE VHDL CODE

```vhdl
-------------------------------------------------------------------------------
--
--  MYRISC project in SME052 1999-12-19
--
--  Anders Wallander
--  Department of Computer Science and Electrical Engineering
--  Luleå University of Technology
--
--  A VHDL implementation of a MIPS, based on the MIPS R2000 and the
--  processor described in Computer Organization and Design by
--  Patterson/Hennessy
--
--
--  Instruction decode stage
--    Compute branch and jump destinations
--    Evaluate data for execution units, branch conditions and jump register
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library Work;
use Work.RiscPackage.all;

entity IDecode is
  port (Clk   : in std_logic;
        Reset   : in std_logic;
        Rdy_n : in  std_logic;

        WriteRegEn  : in std_logic;
        WriteData   : in TypeWord;
        WriteAddr   : in TypeRegister;
        In_IP       : in TypeWord;
        In_Instr    : in TypeWord;

        BP_Mem_iData: in  TypeWord;     -- Bypass from memstage
        BP_Mem_iRDest:  in  TypeRegister; -- Bypass from memstage
        BP_Mem_iRegWrite: in std_logic; -- Bypass from memstage
        BP_EX_iData:  in  TypeWord;       -- Bypass from execution
        BP_EX_iRDest: in  TypeRegister; -- Bypass from execution
        BP_EX_iRegWrite: in std_logic;  -- Bypass from execution

        In_Ctrl_ID  : in  TypeIDCtrl;
        In_Ctrl_Ex  : in  TypeExCtrl;
        In_Ctrl_Mem:  in  TypeMemCtrl;
        In_Ctrl_WB  : in  TypeWBCtrl;

        In_MemBusAccess_n : in  std_logic;

        ID_Stall  : out std_logic;
        ID_Req    : out std_logic;
        ID_BAddr  : out TypeWord;

        ID_Ctrl_Ex  : out TypeExCtrl;
        ID_Ctrl_Mem : out TypeMemCtrl;
        ID_Ctrl_WB  : out TypeWBCtrl;
        ID_A    : out TypeWord;
        ID_B    : out TypeWord;
        ID_IMM  : out TypeWord;
        ID_Shift: out TypeRegister;
        ID_DestReg  : out TypeRegister;
        ID_IP       : out TypeWord
     );
end;

architecture RTL of IDecode is

  -- Instruction aliases
  alias Op : unsigned(5 downto 0) is In_Instr(31 downto 26);
  alias Rs : TypeRegister is In_Instr(25 downto 21);
  alias Rt : TypeRegister is In_Instr(20 downto 16);
  alias Rd : TypeRegister is In_Instr(15 downto 11);
  alias Shift : TypeRegister is In_Instr(10 downto 6);
  alias Funct : unsigned(5 downto 0) is In_Instr(5 downto 0);

  -- Register file
  signal rf_Regs  : TypeArrayWord(31  downto 0) := (others => (others => '0'));

  signal rf_RegA :  TypeWord;
  signal rf_RegB :  TypeWord;
  signal rf_WE    : std_logic;

  -- Signed extended immediate
  signal immSigned : TypeWord;

  -- Destination reg multiplexer
  signal dm_TempReg : TypeRegister;
  signal dm_DestReg : TypeRegister;

  -- Shift multiplexer
  signal sm_Shift : TypeRegister;

  -- Branch logic
  signal br_JAddr : TypeWord;
  signal br_BAddr : TypeWord;

  -- Hazard detection signals
  signal hd_Nop : std_logic;
  signal hd_Stall : std_logic;

  -- Bypass signals
  signal bp_ID_Ctrl_Mem: TypeMemCtrl;
  signal bp_ID_Rt : TypeRegister;

    -- Forwarding unit
  signal bp_Rs_A: std_logic;
  signal bp_Rt_A: std_logic;
  signal bp_Rs_B: std_logic;
  signal bp_Rt_B: std_logic;
  signal bp_Rs_C: std_logic;
  signal bp_Rt_C: std_logic;

  signal bp_Rs_A_val: TypeWord;
  signal bp_Rt_A_val: TypeWord;
  signal bp_Rs_B_val: TypeWord;
  signal bp_Rt_B_val: TypeWord;
  signal bp_Rs_C_val: TypeWord;
  signal bp_Rt_C_val: TypeWord;

  signal  Stall: std_logic;

begin

-------------------------------------------------------------------------------
-- Register file
-------------------------------------------------------------------------------

  -- Write register file -- Write on positive flank
  rf : process(Clk)
  begin
    if rising_edge(Clk) then
      if rf_WE  = '1' then
          rf_Regs(to_integer(WriteAddr)) <= WriteData;
      end if;
    end if;
  end process;

  rf_WE <= WriteRegEn when WriteAddr /= "00000" else
               '0';

  -- Read register file
  rf_RegA <= rf_Regs(to_integer(Rs));
  rf_RegB <= rf_Regs(to_integer(Rt));

-------------------------------------------------------------------------------
-- Sign or zero extend immediate data
--
-- If In_Ctrl_ID.ZeroExtend is asserted then we will zero extend
--
-------------------------------------------------------------------------------
  immSigned(15 downto 0) <= In_Instr(15 downto 0) ;
  immSigned(31 downto 16) <= (others => (In_Instr(15) and not(In_Ctrl_ID.ZeroExtend)));

-------------------------------------------------------------------------------
-- Regdest multiplexer
-------------------------------------------------------------------------------

  dm_TempReg <= Rd when In_Ctrl_Ex.ImmSel = '0' else  -- always asserted when ImmSel is ssserted
                Rt;

  dm_DestReg <= dm_TempReg when In_Ctrl_Ex.JumpLink = '0' else  -- If jal instruction, load
destreg with $31
                "11111";

-------------------------------------------------------------------------------
-- Shift amount multiplexer
-------------------------------------------------------------------------------

  sm_Shift <= "10000" when In_Ctrl_ID.Lui = '1' else              -- If lui instruction force ALU
to shift left 16bits
```

```
              bp_Rs_C_val(4 downto 0) when In_Ctrl_ID.ShiftVar = '1' else  -- If shift variable is
used
              Shift;                                                -- Else shift amount is used

-------------------------------------------------------------------------------
-- Branch and Jump logic
-------------------------------------------------------------------------------

  -- Calculate branch target
  br_BAddr <= (immSigned sll 2) + In_IP;

  -- Assert Req if bp_Rs_C_val equals bp_Rt_C_val
  ID_Req <=   '1' when bp_Rs_C_val =  bp_Rt_C_val else
              '0';

  -- Multiplex jump target
  br_JAddr <= bp_Rs_C_val when In_Ctrl_ID.Jr = '1' else
              In_IP(31 downto 28) & shift_left(In_Instr(27 downto 0), 2);


  -- Multiplex Jump and Branch targets
  ID_BAddr <= br_BAddr when In_Ctrl_ID.Branch = '1' else
              br_JAddr;

-------------------------------------------------------------------------------
-- Forwarding units
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- ALU bypass multiplexer Rs_A  From Writeback stage
-------------------------------------------------------------------------------

  bp_Rs_A_val <=  WriteData when bp_Rs_A = '1' else
                  rf_RegA;

-------------------------------------------------------------------------------
-- Bypass multiplexer Rs_B  From Memstage
-------------------------------------------------------------------------------

  bp_Rs_B_val <=  BP_Mem_iData when bp_Rs_B = '1' else
                  bp_Rs_A_val;

-------------------------------------------------------------------------------
-- ALU bypass multiplexer Rs_C  From Execution stage
-------------------------------------------------------------------------------

  bp_Rs_C_val <=  BP_EX_iData when bp_Rs_C = '1' else
                  bp_Rs_B_val;

-------------------------------------------------------------------------------
-- ALU bypass multiplexer Rt_A  From Writeback stage
-------------------------------------------------------------------------------

  bp_Rt_A_val <=  WriteData when bp_Rt_A = '1' else
                  rf_RegB;

-------------------------------------------------------------------------------
-- Bypass multiplexer Rt_B  From Memstage
-------------------------------------------------------------------------------

  bp_Rt_B_val <=  BP_Mem_iData when bp_Rt_B = '1' else
                  bp_Rt_A_val;

-------------------------------------------------------------------------------
-- ALU bypass multiplexer Rt_C From Execution stage
-------------------------------------------------------------------------------

  bp_Rt_C_val <=  BP_EX_iData when bp_Rt_C = '1' else
                  bp_Rt_B_val;


-------------------------------------------------------------------------------
-- Forwarding evaluation
-------------------------------------------------------------------------------

  -- See if want to bypass Rs from Writeback stage
  bp_Rs_A <= '1' when (WriteRegEn = '1' and
                       WriteAddr /= "00000" and
                       WriteAddr = Rs) else
              '0';

  -- See if want to bypass Rs from MemStage
  bp_Rs_B <=  '1' when (BP_Mem_iRegWrite = '1' and
                        BP_Mem_iRDest /= "00000" and
                        BP_Mem_iRDest = Rs)
                        --TODO???
                        --(BP_EX_iRDest /= In_Rs or BP_EX_MEM(0) = '0') and       -- Prevent dest
reg after load instr.
```

```
                         else
                '0';

  -- See if want to bypass Rs from Execution stage
  bp_Rs_C <= '1' when (BP_EX_iRegWrite = '1' and
                        BP_EX_iRDest /= "00000" and
                        BP_EX_iRDest = Rs) else
                '0';

  -- See if want to bypass Rt from Writeback stage
  bp_Rt_A <= '1' when (WriteRegEn = '1' and
                        WriteAddr /= "00000" and
                        WriteAddr = Rt) else
                '0';


  -- See if want to bypass Rt from MemStage
  bp_Rt_B <=  '1' when (BP_Mem_iRegWrite = '1' and
                         BP_Mem_iRDest /= "00000" and
                         BP_Mem_iRDest = Rt)
                         --TODO???
                         --(BP_EX_iRDest /= In_Rs or BP_EX_MEM(0) = '0') and       -- Prevent dest
reg after load instr.
                         else
                '0';

  -- See if want to bypass Rt from Execution stage
  bp_Rt_C <= '1' when (BP_EX_iRegWrite = '1' and
                        BP_EX_iRDest /= "00000" and
                        BP_EX_iRDest = Rt) else
                '0';

-------------------------------------------------------------------------------
-- Hazard detection
--
-- Stall the pipeline if use of the same register as the load instruction in
-- the previous operation.
--
-------------------------------------------------------------------------------

  hd_Nop <= '1' when  (bp_ID_Ctrl_Mem.MemRead = '1' and
                       ((bp_ID_Rt = Rs) or  bp_ID_Rt = Rt)) else
              '0';

  hd_Stall <= hd_Nop or not(In_MemBusAccess_n);

  ID_Stall <= hd_Stall;     -- Avoid using inout type...

  Stall <= hd_Stall or (not Rdy_n);

-------------------------------------------------------------------------------
-- Shift pipeline
-------------------------------------------------------------------------------

  pipeline : process(Clk,Reset)
  begin
    if reset = '1' then

      ID_Ctrl_EX <= ('0','0','0',"0000");
      bp_ID_Ctrl_Mem <= ('0','0','1');
      ID_Ctrl_WB <= (others => '0');
      ID_A <= (others => '0');
      ID_B <= (others => '0');
      ID_IMM <= (others => '0');
      ID_Shift <= (others => '0');
      ID_DestReg <= (others => '0');
      ID_IP <= (others => '0');
      bp_ID_Rt <= (others => '0');

    elsif rising_edge(Clk) then
      if (Stall = '0') then
        ID_Ctrl_EX <= In_Ctrl_Ex;
        bp_ID_Ctrl_Mem <= In_Ctrl_Mem;
        ID_Ctrl_WB <= In_Ctrl_WB;

        ID_A <= bp_Rs_C_val;
        ID_B <= bp_Rt_C_val;
        ID_IMM <= immSigned;
        ID_Shift <= sm_Shift;
        ID_DestReg <= dm_DestReg;
        ID_IP <= In_IP;
        bp_ID_Rt <= Rt;
      end if;

      if (hd_Stall = '1') then
        --If the pipeline is stalled by hazard detection: insert NOP...
        ID_Ctrl_EX <= ('0','0','0',"0000");
        bp_ID_Ctrl_Mem <= ('0','0','1');
        ID_Ctrl_WB <= (others => '0');
      end if;

    end if;
  end process;

  ID_Ctrl_Mem <= bp_ID_Ctrl_Mem;

end architecture RTL;
```

V. EXECUTION VHDL CODE

```
----------------------------------------------------------------------------
--
--  MYRISC project in SME052 1999-12-19
--
--  Anders Wallander
--  Department of Computer Science and Electrical Engineering
--  Luleå University of Technology
--
--  A VHDL implementation of a MIPS, based on the MIPS R2000 and the
--  processor described in Computer Organization and Design by
--  Patterson/Hennessy
--
--
--  Execute stage
--
----------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library Work;
use Work.RiscPackage.all;

entity Execute is
  port (Clk   : in std_logic;
        Reset : in std_logic;
        Rdy_n : in  std_logic;

        In_Ctrl_Ex  : in TypeExCtrl;
        In_Ctrl_Mem : in TypeMEMCtrl;
        In_Ctrl_WB  : in TypeWBCtrl;
        In_A        : in TypeWord;
        In_B        : in TypeWord;
        In_IMM      : in TypeWord;
        In_Shift    : in TypeRegister;
        In_DestReg  : in TypeRegister;
        In_IP       : in TypeWord;

        BP_EX_iData : out TypeWord;       -- Bypass to idecode
        BP_EX_iRDest: out TypeRegister;   -- Bypass to idecode
        BP_EX_iRegWrite: out  std_logic;  -- Bypass to idecode

        EX_Ctrl_WB  : out TypeWBCtrl;
        EX_Ctrl_Mem : out TypeMemCtrl;
        EX_ALU  : out TypeWord;
        EX_DATA: out TypeWord;
        EX_DestReg: out TypeRegister
        );
end;

architecture RTL of Execute is

  -- ALU
  signal alu_RegA : TypeWord;
  signal alu_RegB : TypeWord;
  signal alu_Shift  : TypeWord;
  signal alu_Result: TypeWord;

begin
----------------------------------------------------------------------------
-- Zero extend shift value
----------------------------------------------------------------------------
  -- TODO ... Isn't this just too ugly...?
  alu_Shift <= x"000000" & "000" & In_Shift;

----------------------------------------------------------------------------
-- Shift / JumpLink multiplexer
----------------------------------------------------------------------------
  alu_RegA <= alu_Shift when In_Ctrl_Ex.ShiftSel = '1' else
              x"0000_0004" when In_Ctrl_Ex.JumpLink = '1' else
              In_A;

----------------------------------------------------------------------------
-- Immediate / JumpLink multiplexer
----------------------------------------------------------------------------
  alu_RegB <= In_IMM when In_Ctrl_Ex.ImmSel = '1' else
              In_IP when In_Ctrl_Ex.JumpLink = '1' else
              In_B;

----------------------------------------------------------------------------
-- ALU
----------------------------------------------------------------------------

  ALU : process(alu_RegA, alu_RegB)
  begin
```

```
    case In_Ctrl_Ex.OP is
      when  ALU_ADD |
            ALU_ADDU  => alu_Result <= alu_RegA + alu_RegB;
      when  ALU_SUB |
            ALU_SUBU  => alu_Result <=  alu_RegA - alu_RegB;
      when  ALU_AND => alu_Result <= alu_RegA and alu_RegB;
      when  ALU_OR => alu_Result <= alu_RegA or alu_RegB;
      when  ALU_XOR => alu_Result <= alu_RegA xor alu_RegB;
      when  ALU_NOR => alu_Result <= alu_RegA nor alu_RegB;

      when ALU_SLT => alu_Result(31 downto 1) <= (others => '0'); alu_Result(0) <=
(signed(alu_RegA) - signed(alu_RegB))(31);
      when ALU_SLTU=> alu_Result(31 downto 1) <= (others => '0'); alu_Result(0) <=  ( alu_RegA -
alu_RegB )(31);
--      when  ALU_SLT => alu_Result(31 downto 1) <= (others => '0'); alu_Result(0) <=
unsigned((signed(alu_RegA) - signed(alu_RegB)))(31);
--      when  ALU_SLTU=> alu_Result(31 downto 1) <= (others => '0'); alu_Result(0) <=  unsigned((
alu_RegA - alu_RegB ))(31);

      when  ALU_SLL => alu_Result <=  shift_left( alu_RegB, to_integer(alu_RegA(4 downto 0)));
      when  ALU_SRL => alu_Result <=  shift_right( alu_RegB, to_integer(alu_RegA(4 downto 0)));
      when  ALU_SRA => alu_Result <=  unsigned(shift_right( signed(alu_RegB),
to_integer(alu_RegA(4 downto 0))));
      when  others => alu_Result <= (others=>'-');
    end case;
  end process;


-------------------------------------------------------------------------------
-- Shift the pipeline
-------------------------------------------------------------------------------

  pipeline : process(Clk,Reset)
  begin
    if reset = '1' then
      EX_Ctrl_Mem <= ('0','0','1');
      EX_Ctrl_WB <= (others => '0');
      EX_ALU <= (others => '0');
      EX_DATA <= (others => '0');
      EX_DestReg <= (others => '0');
    elsif rising_edge(Clk) then
      if (Rdy_n = '1') then
        EX_Ctrl_Mem <= In_Ctrl_Mem;
        EX_Ctrl_WB <= In_Ctrl_WB;
        EX_ALU <= alu_Result;
        EX_DATA <= In_B;
        EX_DestReg <= In_DestReg;
      end if;
    end if;
  end process;


-------------------------------------------------------------------------------
-- Bypass signals to idecode
-------------------------------------------------------------------------------

  BP_EX_iData <= alu_Result;
  BP_EX_iRDest <= In_DestReg;
  BP_EX_iRegWrite <= In_Ctrl_WB.RegWrite;

end architecture RTL;
```

VI. MEMORY ACCESS VHDL CODE

```
-------------------------------------------------------------------------------
--
--  MYRISC project in SME052 1999-12-19
--
--  Anders Wallander
--  Department of Computer Science and Electrical Engineering
--  Luleå University of Technology
--
--  A VHDL implementation of a MIPS, based on the MIPS R2000 and the
--  processor described in Computer Organization and Design by
--  Patterson/Hennessy
--
--
--  Memory stage
--
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library Work;
use Work.RiscPackage.all;

entity MemStage is
  port (Clk   : in std_logic;
        Reset   : in std_logic;
        Rdy_n : in  std_logic;

        In_Ctrl_WB    : in TypeWBCtrl;
        In_Ctrl_Mem : in TypeMEMCtrl;
        In_ALU  : in TypeWord;
        In_Data : in TypeWord;
        In_DestReg: in TypeRegister;

        In_IntBusGrant_n :  in  std_logic;

        Bus_Data  : inout TypeWord;
        Bus_Addr  : out TypeWord;
        Bus_Rd_n    : out std_logic;
        Bus_Wr_n    : out std_logic;

        BP_Mem_iData: out TypeWord;     -- Bypass to idecode
        BP_Mem_iRDest:  out TypeRegister; -- Bypass to idecode
        BP_Mem_iRegWrite: out std_logic;  -- Bypass to idecode

        Mem_Ctrl_WB : out TypeWBCtrl;
        Mem_Data: out TypeWord;
        Mem_DestReg: out TypeRegister
        );
end;

architecture RTL of MemStage  is

  -- Mem to Reg mux
  signal mr_mux:  TypeWord;

begin


-------------------------------------------------------------------------------
-- Mem to reg multiplexer
-------------------------------------------------------------------------------
  mr_mux <= Bus_Data when In_Ctrl_Mem.MemRead = '1' else
            In_ALU;

--------------------------------------------------------------------------------------
-- Bus stuff
--------------------------------------------------------------------------------------

  -- Tristate address bus if no access...
  Bus_Addr <= In_ALU when In_IntBusGrant_n = '0' else
              (others => 'Z');

  Bus_Rd_n <= not In_Ctrl_Mem.MemRead when In_IntBusGrant_n = '0' else
              'Z';

  -- Write when MemWrite asserted
  Bus_Wr_n <= not In_Ctrl_Mem.MemWrite when In_IntBusGrant_n = '0' else
              'Z';

  -- Output data when MemWrite asserted
  Bus_Data <= In_Data when In_Ctrl_Mem.MemWrite = '1' and In_IntBusGrant_n = '0' else
          (others => 'Z');

-------------------------------------------------------------------------------
```

```
-- Shift the pipeline
------------------------------------------------------------------------------

  pipeline : process(Clk,Reset)
  begin
    if reset = '1' then
      -- TODO - do we really need to flush everything?
      Mem_Ctrl_WB <= (others => '0');
      Mem_DATA <= (others => '0');
      Mem_DestReg <= (others => '0');
    elsif rising_edge(Clk) then
      if (Rdy_n = '1') then
        Mem_Ctrl_WB <= In_Ctrl_WB;
        Mem_DATA <= mr_mux;
        Mem_DestReg <= In_DestReg;
      end if;
    end if;
  end process;

------------------------------------------------------------------------------
-- Pass signals to idecode
------------------------------------------------------------------------------
  BP_Mem_iData <= mr_mux;
  BP_Mem_iRDest <= In_DestReg;
  BP_Mem_iRegWrite <= In_Ctrl_WB.RegWrite;

end architecture RTL;
```