ÇUKUROVA UNIVERSITY
INSTUTE OF NATURAL AND APPLIED SCIENCES

MSc THESIS

Ali ŞENTÜRK

REDUCED INSTRUCTION SET PROCESSOR DESIGN

DEPARTMENT OF COMPUTER ENGINEERING

ADANA, 2009

INSTITUTE OF NATURAL AND APPLIED SCIENCE
UNIVERSITY OF ÇUKUROVA

REDUCED INSTRUCTION SET PROCESSOR DESIGN

Ali ŞENTÜRK

MSc THESIS

DEPARTMENT OF COMPUTER ENGINEERING

We certify this thesis is satisfactory the award of MSc degree at the date ……

Signature……..	Signature …….	Signature …….
Assist.Prof.Dr. Mustafa GÖK	Assist.Prof.Dr. Murat AKSOY	Assist.Prof.Dr. Mutlu AVCI
Supervisor	Member of Examining Committee	Member of Examining Committee

Certified that this thesis conforms to the formal standards of the Institute.

Code no:

Prof. Dr. Aziz ERTUNÇ
Director
Institute of Natural and Applied Science

ABSTRACT

MSc THESIS

REDUCED INSTRUCTION SET PROCESSOR DESIGN

Ali ŞENTÜRK

UNIVERSITY OF ÇUKUROVA

INSTUTE OF NATURAL AND APPLIED SCIENCES

DEPARTMENT OF COMPUTER ENGINEERING

Supervisor: **Assist. Prof. Dr. Mustafa GÖK**
Year: **January 2009, Pages: 59**
Jury: **Assist. Prof. Dr. Mustafa GÖK**
Assist. Prof. Dr. Murat AKSOY
Assist. Prof. Dr. Mutlu AVCI

Reduced instruction set computer (RISC) processors are designed according to the principle "simple is better." The RISC processors are widely using in; embedded systems to work stations. Even the complex instruction computers (CISC) use RISC type micro instructions internally. This thesis presents a 32-bit pipelined reduced instruction set processor design that has 32-bit basic arithmetic, logic, control and system instructions. Most of the pipeline hazards are eliminated by hardware support. The design is modeled with VHDL (Very High Speed Integrated Circuit Hardware Description Language) hardware description language and simulated with the Mentor Graphics Corporation's Modelsim simulator. The processor model is mapped on a low cost FPGA (Field Programmable Gate Array) chip.

Key Words: RISC Processor, Pipeline Architecture, FPGA

ÖZ

YÜKSEK LİSANS TEZİ

İNDİRGENMİŞ KOMUT SETLİ İŞLEMCİ TASARIMI

Ali ŞENTÜRK

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

Danışman: **Yrd. Doç. Dr. Mustafa GÖK**
Yıl: **Ocak 2009, Sayfa: 59**
Jüri: **Yrd. Doç. Dr. Mustafa GÖK**
Yrd. Doç. Dr. Murat AKSOY
Yrd. Doç. Dr. Mutlu AVCI

Basit daha iyidir felsefesi ile tasarlanan indirgenmiş komut setli bilgisayar (İKSB) işlemcileri her geçen gün tasarımcıların dikkatini daha çok çekmektedir. Gömülü sistemlerden iş istasyonlarına kadar çok alanda yaygın olarak kullanılmaya başlanan İKSB işlemciler, karmaşık komut setli bilgisayarlarda da dahili mikro komutlar olarak kullanmaktadır. Bu tezde 32-bit temel aritmetik, mantık, kontrol ve sistem komutlarının bulunduğu, boru hattı mimarisine sahip bir kısıtlı komut setli işlemci tasarımı sunulmuştur. Boru hattı organizasyonundan doğan riskler donanımsal metotlarla elimine edilmiştir. İKSB işlemcisi tasarım prensiplerinin gerçeklenmesi sağlanmıştır. Tasarım için VHDL (Very High Speed Integrated Circuit Hardware Description Language) donanım tanımlama dili kullanılmış ve tasarımın betimlenmesi Mentor Graphics firmasının Modelsim programı ile yapılmıştır. Tasarım düşük maliyetli bir APKD (Alan Programlanabilir Kapı Dizisi) yongası kullanılarak sentezlenmiştir.

Anahtar Kelimeler: KKSB İşlemci, Boru Hattı Mimarisi, APKD

CONTENTS **PAGE**

LIST OF TABLES **PAGE**

LIST OF FIGURES **PAGE**

1. INTRODUCTION

A computer is a complex electronic device operates according to the instructions to perform data manipulation (Anonymous, 2008a). This definition introduces two main concepts. Complex electronic device is the physical side of the computer generally referred as hardware. Computer programs are constituted by instructions. Computer programs have many abstraction levels. High level programming languages are more close to human readability. Compilers convert high level language code to assembly language. Assembly language is one-to-one representation of the machine codes. Assembler converts assembly codes to binary codes. Instructions can be defined as the words that the computer understands and instruction set is the language of the computer (Patterson, 2005).

Computer design is a process of interconnecting electronic components to implement hardware of the computer that meets computing requirements. Computer designers must consider both performance and costs of the design. Resources must be used optimum. Computer uses memory units to hold data or instructions and other logic units for obtaining results.

From instruction set point of view, the computers can be divided into two groups. The first group constitutes complex instruction set computers (CISC) and the second group constitutes reduced instruction set computers (RISC). CISC instructions emulate high level programming languages to simplify compiler design and support many addressing modes (Dandamudi, 2003), (Abd-El-Barr, 2005). The instruction size is variable in CISC processor which result complex instruction decoding circuit designs. RISC instruction set contains fewer instructions. The instructions are fixed size small number of addressing modes are supported. These aspects simplify the control design and aid the design of pipeline organization (Dandamudi, 2004), (Patterson, 1980). The clock cycle of the RISC systems are shorter than CISC systems, though less work is done by one instruction (Bodur, 2005), (Colwell, 1985). Because of the performance advantage and ease of pipelining even modern general purpose CISC architectures imitate RISC like microinstructions in their control system designs (Alpert,1993), (Torres, 2006).

The popularity of RISC designs is expected to grow due to their recent trends in multicore chips where each chip contains two or more processor cores. RISC architecture is a good candidate for a multiprocessor core since it has a smaller area and less power consumptions (Yeager, 1996). Accounted advantages of the RISC systems motivated the work presented in this thesis.

The goal of this thesis is the design of a practically realizable RISC processor. To achieve this goal an instruction set similar to Berkeley RISC-I is designed (Mano, 1993). 32-bit instruction set contains 32-bit instructions that support basic arithmetic, logic, data, transfer and system functions. The processor has five pipeline stages and modern hardware techniques utilized to deal with data and structural hazards such as data forwarding unit. The presented processor is implemented by VHDL (Pedroni, 2004) and functionality of the processor is tested by Mentor Graphics Corporation's Modelsim SE 6.3f Simulator (Mentor Graphics, 2008). The processor is synthesized with the Xilinx ISE 9.2i with the target device Xilinx xc3s250e-4-vq100 (Xilinx, 2008).

2. INSTRUCTION SET ARCHITECTURE

Instructions can be defined as the words which the processor understands and instruction set is the language the processor speaks (Patterson, 2005). Each instruction has a name and syntax which increases human readability. The instructions in this thesis are 32-bit long. There are 4 types of instructions. These are explained in the following sections.

2.1. R-Type Instructions

R-type instructions are designed for arithmetic and logical operations. R-type instructions are 3 operand instructions. Instruction gets all operands from internal registers. The structure of the R-type instruction is shown in Figure 2.1.

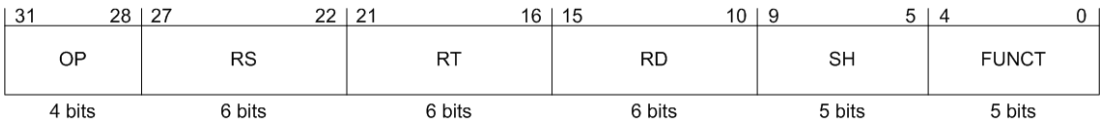

Figure 2.1 R-Type Instruction Structure

OP is used for Operation Code. OP field is 4-bits and defines the type of the instruction for R-type instructions. OP field is set to “0000” for R-type instructions. RS, RT and RD fields are used for the register address. Since RT, RS and RD fields are 6-bits long; they can address 2^6=64 registers. RS field is used for the first operand and RT is used for second operand of the arithmetic or logic operations. RD field is the address of the operation result. SH field keeps the shift amount data. This field is considered for shift instructions and for other R-type instructions SH field is ignored. R-type instructions are explained in the following sections.

2.1.1. And Instruction

This instruction performs logical AND operations on two registers. The structure of the And instruction is shown in Figure 2.2. OP field is 0000 and FUNCT field is set to 00100. RS and RT is the source addresses and RD is destination address of the operation. SH field is ignored for this operation.

And RD, RS, RT is the assembly language notation of the And operation. “#” symbol is used for comments to increase human readability.

Example:

And R1, R5, R3 # R1 = R5 and R3

This instruction makes and operation on the 5th and 3rd registers and writes the result to the 1st register. The binary code of this operation is
0000 000101 000011 000001 00000 0100.

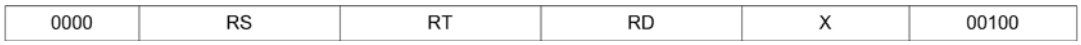

Figure 2.2 And Instruction Structure

2.1.2. Or Instruction

Or instruction performs logical bitwise OR operation of the registers in the address RS and RT fields. The result is written to the register that the address of the register is in the RD field. The structure of the register is same as the structure in the Figure-2 but only the FUNCT field of the Or instruction is set to 00101.

Example:

Or R4, R2, R5 # R4 = R2 or R5

2.1.3. Not Instruction

Not instruction generates bitwise complement of an operand. This instruction makes 1’s 0 and vice versa in the register addressed with the RS field. For this register FUNCT field is set to 01001, RT and SH fields are “don’t care”.

Example:

Not R3, R5 # R3 = not R5

2.1.4. Xor Instruction

Xor instruction performs bitwise XOR operation. In other words the result for each position is 0 if the bits the corresponding bits are equal and 1 if they are different. Again RS and RT are the source addresses of the operands, RD is the destination address. FUNCT field is set to 00110.

Example:

```
Xor R2, R3, R2   # R2=R3 xor R2
```

2.1.5. Sll Instruction

Sll stands for shift left logical. This instruction used for logical shift operations that shifts all bits in a register to the left by the amount of the SH field. Empty bits are filled by 0s. RS field is the address of the executed to be shifted and the result of the shift is written in the register of address RD. RT field is don't care field for this instruction. FUNCT field of the Sll instruction is 01010.

Example:

```
Sll R2, R4, 3   # R2 = R4 sll 2
```

2.1.6. Srl Instruction

Srl instruction is used for logical right shift operation. This instruction operates as Sll instruction but shifting is directed to right. Again empty fields are filled with 0. RS source address, RD destination address and RT don't care fields. SH field is used for shift amount. FUNCT field is 01011.

Example:

```
Srl R3, R3, 8   #R3 = R3 srl 8
```

R3 (initial): 01111110010010101001000111000011
R3 (after Srl operation): 00000000011111100100101010010001.

2.1.7. Sra Instruction

Sra instruction is arithmetic right shift instruction. This instruction works similar to sla however empty spaces filled with the copies of the most significant bit. FUNCT field for this instruction is 01101. Other fields are the same function as in the other shift instructions.

Example:

Sra R3, R5, 5 #R3 = R5 sra 5

R5: 11001000010011101000001010100011
R3: 11111110010000100111010000010101

Left most bit of R5 register is 1 so the empty 5 bit after shift operation is filled with 1.

R5: 01001000010011101000001010100011
R3: 00000010010000100111010000010101

R3 is the result and written in the register of R3 address.

2.1.8. Add Instruction

This instruction is used to obtain the sum of two registers. Source registers are in the addresses of RS and RT. Sum is written to the register of address RD. FUNCT field is set to 00001 for this instruction.

Example:

Add R5, R2, R4 # R5 = R2 + R4

R2: 10101100010010011011011111011110
R4: 00001000111001111101011011001111
Result (R5): 10110101001100011000111010101101

2.1.9. Sub Instruction

Sub instruction performs subtraction operation. RS field holds the address of the register that is used as minuend and RT field holds the address of the subtrahend. The difference is written to the register in the address of RD. FUNCT field is 00010. SH field is ignored.

Example:

```
Sub R2, R3, R2   #R2 = R3 - R2

R3:             10101100010010011011011111011110
R2:             00001000111001111101011011001111
Result (R2):    10100011011000011110000100001111
```

2.1.10. Mul Instruction

This instruction performs two's complement multiplication. For this instruction FUNCT field is set to 00011 RS and RT fields hold the addresses of the multiplicand and multiplier respectively. RD is the address of the register that result is written.

Example:

```
Mul R3, R3, R2   #R3 = R3 x R2

R3:             11111111111111111111110011011110 (-802)10
R2:             00000000000000010001011011001111 (71375)10
Result (R3):    11111100100101101000101110000010 (-57242750)10
```

2.1.11. Mulu Instruction

Mulu instruction is the unsigned multiplication instruction. In this instruction binary numbers are considered as positive numbers and all the bits in the registers are forms the magnitude. FUNCT field is 00111.

Example:

```
Mulu R2, R4, R3   # R2=R4 x R3
```

R3: $00000000000000111011100011011110\,(243934)_{10}$
R2: $00000000000000000011011011001111\,(14031)_{10}$
Result (R3): $11001100000000010100111110000010\,(3422637954)_{10}$

2.1.12. Mov Instruction

Mov instruction is used for carry one register's content to other register. Mov instruction is implemented by using Add instruction. FUNCT field is 00001 as Add instruction. RS is the address of the register that will be carried. RD is the destination address. RT address is set to 00000 which addresses the 0^{th} register that is fixed to 0. This instruction is takes one of the registers and adds 0 to that register and writes the result to the destination register. Move process is completed in this manner. Figure 2.3 shows the structure of the Mov instruction.

Example:

```
Mov R36, R25   # R36=R25
```

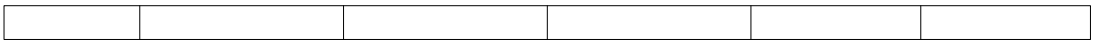

Figure 2.3 Structure of the Mov Instruction

2.2. I-type Instructions

I-type or immediate type instructions are for again arithmetic and logical operations but these instructions have a data field. Data in the instruction is processed with the content of one register. The structure of the instruction is shown in Figure 2.4.

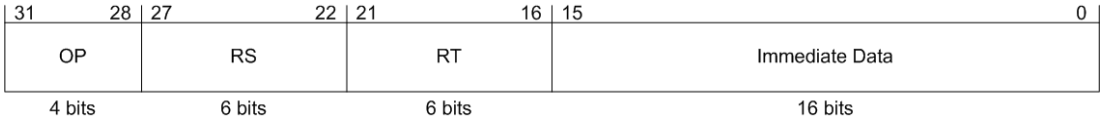

Figure 2.4 I-Type Instruction Structure

I-type instructions have 4 fields. OP field is 4 bits and specifies the operation. 6-bits RS field holds the address of the source register that is processed. 6-bits RT field holds the destination address for I-type instruction. Immediate field is 16-bits and hold the immediate data.

To perform I-type arithmetic and logical instructions', data field is extended to 32-bits. This operation is a signed operation. If the 15^{th} bit is 1, bits 16 to 31 are 1s otherwise bits 16 to 31 are 0s. This operation converts 16 bit to 32 bit and protects its magnitude and sign.

2.2.1. Andi Instruction

Andi instruction is used for bitwise AND operation as And instruction. But this instruction operates on the immediate constant and the data in one of the registers. OP field is 0001 for Andi instruction.

Example:

Andi R10, R12, 23067 # R10=R12 and 23067

R12: 00000010000100111011100011011110
$(23067)_{10}$: 00000000000000000101101000011011
R10: 00000000000000000001100000011010

2.2.2. Ori Instruction

Ori instruction is immediate bitwise OR operation instruction. It performs OR operation on a register and the immediate constant. RS source address, RT destination address of the operation. OP field is 0101 for Ori instruction.

Example:

Ori R10, R12, 23067 # R10=R12 or 23067

R12: 00000010000100111011100011011110
$(23067)_{10}$: 00000000000000000101101000011011
R10: 00000010000100111111101011011111

2.2.3. Xori Instruction

This instruction performs immediate bitwise XOR operation. OP field is 0110.

Example:

Xori R10, R12, 23067 # R10=R12 xor 23067

R12: 00000010000100111011100011011110
$(23067)_{10}$: 00000000000000000101101000011011
R10: 00000010000100111110001011000101

2.2.4. Addi Instruction

This instruction adds the immediate constant and one of the register's content. RS field holds the address of the one operand and RT field holds the address of the register which the result is be written. OP field is 00001 for Addi instruction.

Example:

Addi R2, R2, 18320 # R2 = R2 + 18320

R2: 10101100010010011011011111011110 $(-1404454946)_{10}$
$(18320)_{10}$: 00000000000000000100011110010000
R2: 10101100010010011111111101101110 $(-1404436626)_{10}$

2.2.5. Muli Instruction

Muli is short form of multiply immediate word. Muli instruction is for immediate signed multiply operation. OP field is 0011 for Muli instruction. RT is the address of the multiplicand and immediate data on the instruction is multiplier.

Example:

Muli R25, R14, -18320 # R25 = R14 x (-18320)

R14: 11111111111111100101100010011010 $(-108390)_{10}$
$(-2893)_{10}$: 11111111111111111111010010110011
R25: 00010010101100001011101110101110 $(313572270)_{10}$

2.2.6. Movi Instruction

Movi instruction is immediate move instruction. This instruction storesimmediate constant to the destination register. Similar to Mov instruction this instruction implemented by using Addi instruction. OP field is set to 0001 and RS field is set to 00000 which addresses the 0^{th} register that's content is filled with 0s. RT field is the address of the immediate data is going to be written. Movi instruction takes the 0^{th} instruction, sum up with the immediate data on itself and write it back to destination address.

Example:

Movi R45, 45323 # R45=45323

2.2.7. Beq Instruction

Beq instruction is used for conditional branching. Beq instruction tests if the register addressed in the RS field is equal to register addressed in the RT field, if the test is true PC is set to branch address. This address is computing by adding the value of PC with the offset value stored in least significant half of the instruction. OP field is set to 1001. RS and RT fields are holds the addresses of the registers that are

compare if they are equal or not. If it is required to know that whether one of the register is equal to 0 or not one of the RS or RT fields can be set to 0 so that addresses the 0^{th} register which holds 0 and it is compared with the other register.

Example:

Beq R3, R48, 28 # if(R3 = =R48) then branch to $(PC+1+28)^{th}$ instruction

2.2.8. Bne Instruction

Bne is also used for conditional branching but branching occurs when the registers addressed in the instruction are not equal. RS and RT fields are holds the registers that are compared. OP field is 1010 for Bne instruction. If the compared registers are not equal processor jumps to the instruction in the address that sum of the branch instruction address plus one and offset.

Example:

Bne R10, R53, 35 # if(R10 ! =R53) then branch to (next address +35)th instruction

2.2.9. Lw

Lw and Sw instructions are the only instructions that reach to the memory. Lw is the short form of load word. If any data in the memory is needed, it can be brought with Lw instruction to the processor and write to the destination address that RT field holds. OP field is set to 0111 for Lw instruction and RS field is don't care. Immediate field is the absolute address of the memory location. Since immediate field is 16 bit, 65535 memory blocks can be addressed.

Example:

Lw R6, 250 #R6=Mem[250]

2.2.10. Sw Instruction

Sw instruction is used for write a register content to memory. The OP field is 1000 for this instruction and RT field is don't care because there is no need destination register. Register is only used as source and its address is hold in the RS field. Immediate field holds the absolute address of memory again.

Example:

Lw 170, R2 #Mem[170]=R2

2.2.11. Out Instruction

Out instruction is used to write any registers content to the output. There is a output register added to the project for this instruction. The OP field is 1101 for this instruction. RS field specifies the source address of the register. RT field is fixed to 0000. The other fields are don't care for this instruction.

Example:

Out R4 # Put Register 4 content to the output.

2.3. J-type Instructions

J-type instructions are unconditional branching instructions. J capital is used for jump word. There are 2 types of unconditional branching instructions. OP field specify type of jump instruction and 16-bit Jump Amount field is for relative jump address. The bits between 27-16 are don't care bits. Figure 2.5 shows the structure of the J-type instructions.

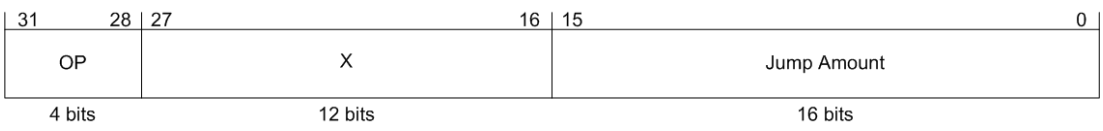

Figure 2.5 Structure of J-Type Instruction

2.3.1. Ba Instruction

Ba is the short form of branch always words. Ba instruction is used for unconditionally branching only. OP field is 1011 and 15-0 bits are for jump amount.

Example:

```
Ba 125   #jump to (next address + 125)
```

2.3.2. Bl Instruction

Bl is acronym for branch and link words. This instruction can be used for jumping to an address and the address of the next instruction is written in a register. With using this instruction after executing desired instructions, returning to the before executing sequence can be possible. OP field is 1100.

Example:

```
Bl 100   #jump to (next address + 100) and link next address
```

2.3.3. S-type Instructions

Remaining instructions are system instructions. System instructions' OP field is 1110 and FUNCT field specifies the exact function of S-type instruction. 27 to 5 bits are don't care bits. Figure 2.6 shows the structure of S-type instruction. Nop instruction can be considered as S-type instruction although its OP field is 1111.

Figure 2.6 S-Type Instruction Structure

2.3.4. Hlt Instruction

FUNCT field is 00001 for Hlt instruction. Hlt is short form of halt word. Hlt instruction stops executing processor. After using Hlt instruction, the system can be restarted by using a switch.

Example:

```
Hlt   #stop executing
```

2.3.5. Syscall Instruction

Syscall instruction is short form of the system call. This instruction is used for system interrupts. The processor branches to a predetermined location and handling of the interrupt is done by the software on that location. FUNCT field is 00010 for Syscall instruction.

Example:

```
Syscall   #jump to specified address to handle interrupt
```

2.3.6. Lret Instruction

After using Bl instruction if it is required to return to the branching address, Lret instruction can be used. Lret address provides returning to the address stored by Bl instruction in a register named as ReturnReg.

Example:

```
Lret   #return to the branch address
```

2.3.7. Eret Instruction

This instruction is used for returning to the address of an instruction which causes exception. This instruction has no parameters.

2.3.8. Nop

If this instruction is used, processor does not do any operation. OP field is 1111 for this instruction

Table 2.1 List of the Instructions with Explanation Type and Field

Instruction	**Explanation**	**Type**	**Fields**
Add	Addition	R	0 - RS - RT - RD - X - 1
Sub	Subtraction	R	0 - RS - RT - RD - X - 2
Mul	Multiplication	R	0 - RS - RT - RD - X - 3
Mulu	Unsigned Multiplication	R	0 - RS - RT - RD - X - 7
And	AND	R	0 - RS - RT - RD - X - 4
Or	OR	R	0 - RS - RT - RD - X - 5
Not	NOT	R	0 - RS - X - RD - X - 9
Xor	XOR	R	0 - RS - RT - RD - X - 6
Sll	Shift Left Logical	R	0 - RS - RT - RD - SAmt - 10
Srl	Shift Right Logical	R	0 - RS - RT - RD - SAmt - 11
Sra	Shift Right Arithmetic	R	0 - RS - RT - RD - SAmt - 13
Beq	Branch if Equal	I	9 - RS – RT - RelJump
Bne	Branch if not Equal	I	A - RS - RT - RelJump
Ba	Branch Always	J	B - X - X - RelJump
BL	Branch and Link	J	C - X - X - RelJump
Mov	Move	R	0 - RS - Zero - RD - X - 1
Movi	Move Immediate	I	1 - Zero - RT - Immediate
Addi	Add Immediate	I	1 - RS - RT - Immediate
Out	Word Out	I	2 - RS - Zero - X
Muli	Multiply Immediate	I	3 - RS - RT - Immediate
Andi	AND Immediate	I	4 - RS - RT - Immediate
Ori	OR Immediate	I	5 - RS - RT - Immediate
Xori	XOR Immediate	I	6 - RS - RT - Immediate
Lw	Load Word	I	7 - X - RT - Address
Sw	Store Word	I	8 - RS - X - Address
Nop	No Operation	S	F -X
Hlt	Halt	S	E - 1
Syscall	Software Interrupt	S	E - 2
Lret	Lint Return	S	E - 3
Eret	Exception Return	S	E - 4

3. PIPELINED DATAPATH

3.1. Pipeline Structure

Pipeline is one of the key method to increase performance of processors (Hennessy, 2003), (Parhami, 2005). Units of the processor are grouped according to operating time to constitute pipeline structure. These groups can be considered as the pipeline stages which perform subtasks of the processor to execute instruction. Pipeline registers are used between these pipeline stages to separate them. This structure allows processors to process more than one instruction inside. The aim of this structure is using each stage by different instructions at a time and increasing execution rate to1 instruction per clock.

Early RISC processors have almost the same pipeline structure. This pipeline structure has 5 stages and known as classic RISC pipeline (Anonymous, 2008b). The processor in this thesis has 5 pipeline stages as classic RISC pipeline. These pipeline stages are:

1. Instruction Fetch
2. Decode
3. Execute
4. Memory
5. Write back

We can illustrate the comparison of pipeline and multicycle implementations as in Figure 3.1. It is assumed that both processors have same subunits. As shown in the Figure 3.1 multicycle processor executes 2 instructions in 10 cycle period. In this configuration, instruction used only one group of unit in a specific period. Remaining 4 groups of units are not used. Obviously this is inefficient way and waste resources. The stages emptied by previous instructions are used by the following instruction in the pipeline architecture. After first 4 instructions, all stages of the pipelined processor is used by instructions simultaneously and there are 5 instructions in the pipeline at the same time. As shown in the Figure 3.1 multicycle processor executes 2 instructions and pipelined processor executes 6 instructions at the same time.

Actually if the first 4 are ignored, one instruction execution per one clock is provided by this architecture.

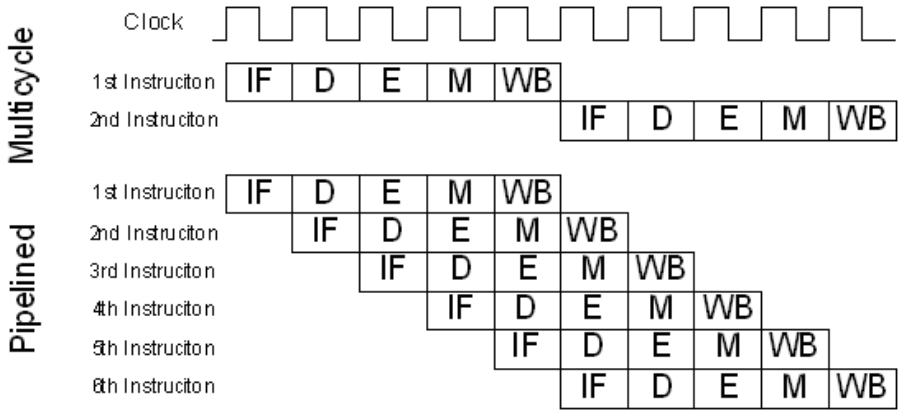

Figure 3.1 Pipelined Datapath vs Single Cycle Datapath

3.2. Pipeline Stages

In this section these stages are explain in more detail including all logic elements in the stage.

3.2.1. Instruction Fetch

This stage consists of the units that bring instructions from memory unit. There are 4 main components of the instruction fetch stage.

3.2.1.1. Program Counter

This register holds the address of the instruction. It is a 16-bit register and generally called as program counter (PC). The address stored in PC is changed with the rising edge of the clock signal. There are two additional inputs of the PC. One is reset signal (rst); the other is write signal (wrt) shown in Figure 3.2.

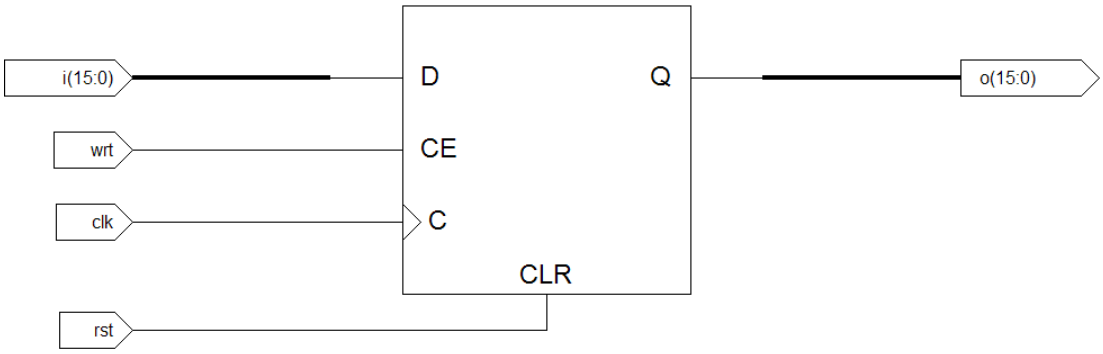

Figure 3.2 Program Counter

3.2.1.2. Incrementer

If there is no branching or exception, in other words if the processor executes the instructions in order, the address in the PC is increment. This operation is done by an incrementer circuit in Figure 3.3. This unit is increments the output of the register. Incremented output is again connected to the input of the PC.

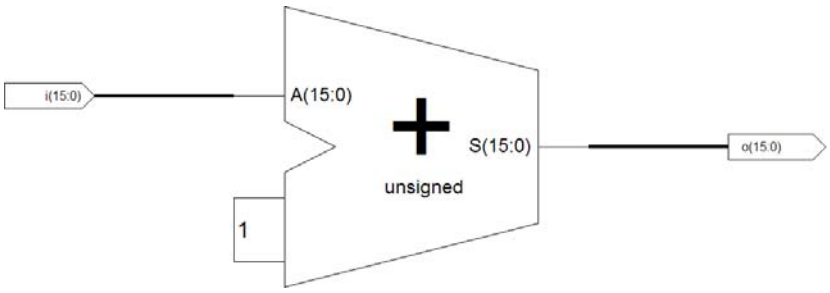

Figure 3.3 Incrementer Circuit

3.2.1.3. Branch Mux

Multiplexers are the logic elements that select the one of its inputs and transfers this input to the output. 16-bit 4-input multiplexer is shown in the Figure 3.4. Branch Mux is connected to the input of the PC.

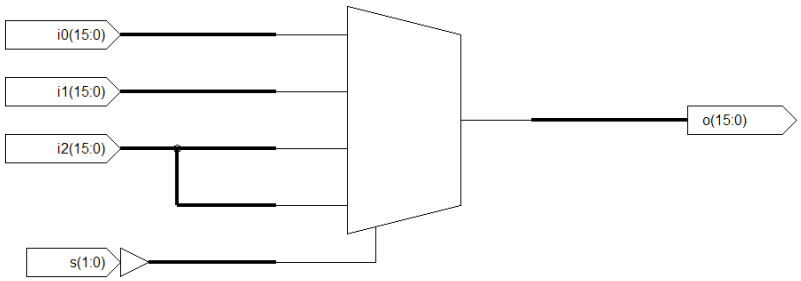

Figure 3.4 16-bit Four Input Multiplexer

Instruction fetch sequence can be broken with a branch/jump instruction or with an interrupt/exception. If these events happen, the correct address is sent via a multiplexer which is connected to the input of the PC. Fetch stage becomes as in the Figure 3.5. Branch mux selects the incremented PC output, jump address or exception address. Multiplexer selection inputs are controlled by branching and exception units. These are explained in the following sections. Instruction address is connected to the address port of the instruction memory.

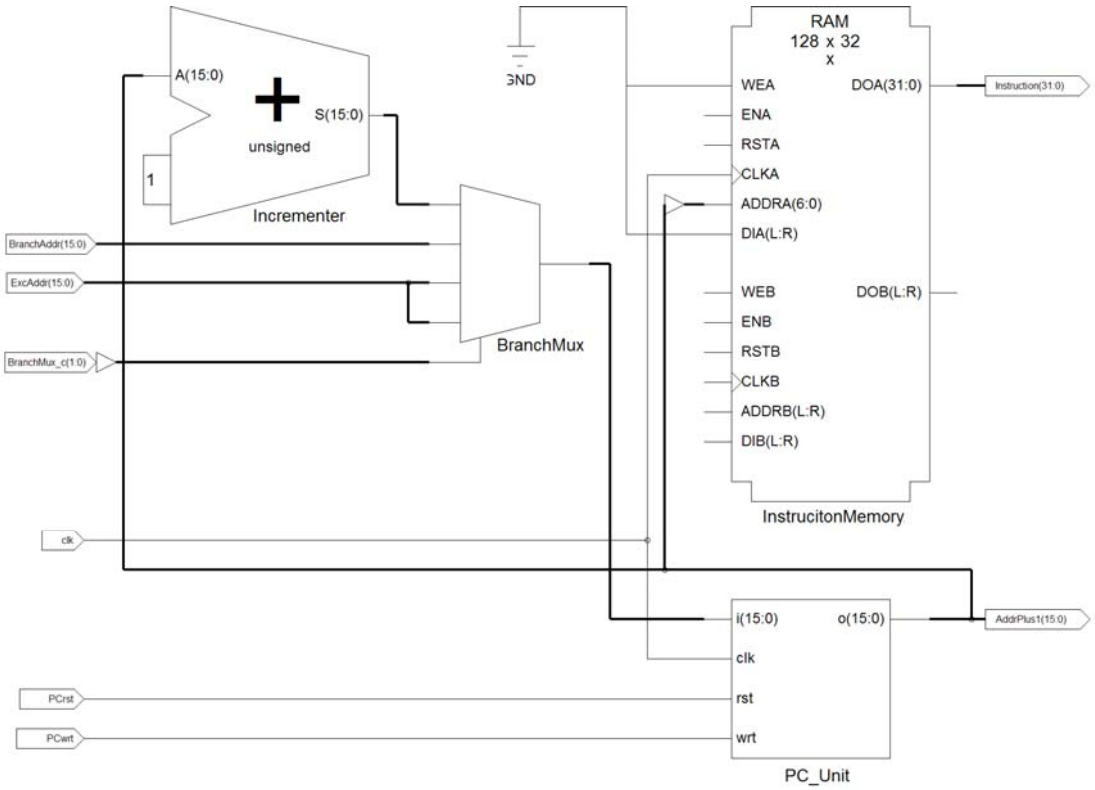

Figure 3.5 Fetch Stage

3.2.1.4. Instruction Memory

There are two memories in the processor. One is instruction memory which is used for store instructions. The other is data memory which is in the 4^{th} stage. The instruction memory can be constructed as 2^{16} blocks. But in the processor it is designed as 128 blocks.

3.2.2. Instruction Decode

3.2.2.1. Control Unit

Any instruction goes through pipeline must also carry control signals belong to itself synchronously. This control signals are generated by control unit. Op and Funct fields of the instructions are the inputs of the Control Unit. The units are shown as blocks in the Figure 3.6.

Branch control signals following logic expressions are given in the Table 3.1. These branch signals are connected to the branch unit at the 3^{rd} stage. Branch unit decides whether there is any branching operation or not. Required branching signals are generated by the branch unit.

Table 3.1 Branch Control Signals

Instruction	Op field	Logic Expressions
Be	1001	BranchEq = Op(3) and $\overline{\text{Op(2)}}$ and $\overline{\text{Op(1)}}$ and Op(0)
Bne	1010	BranchNEq = Op(3) and $\overline{\text{Op(2)}}$ and Op(1) and $\overline{\text{Op(0)}}$
Ba	1011	BranchAlw = Op(3) and $\overline{\text{Op(2)}}$ and Op(1) and Op(0)
BL	1100	BranchAndLnk = Op(3) and Op(2) and $\overline{\text{Op(1)}}$ and $\overline{\text{Op(0)}}$

DataMRead and DataMemwrite signals are used for reaching the main memory. These signals are generated if the Lw or Sw instructions are executed. Logic expressions of the circuit are shown in the Table 3.2

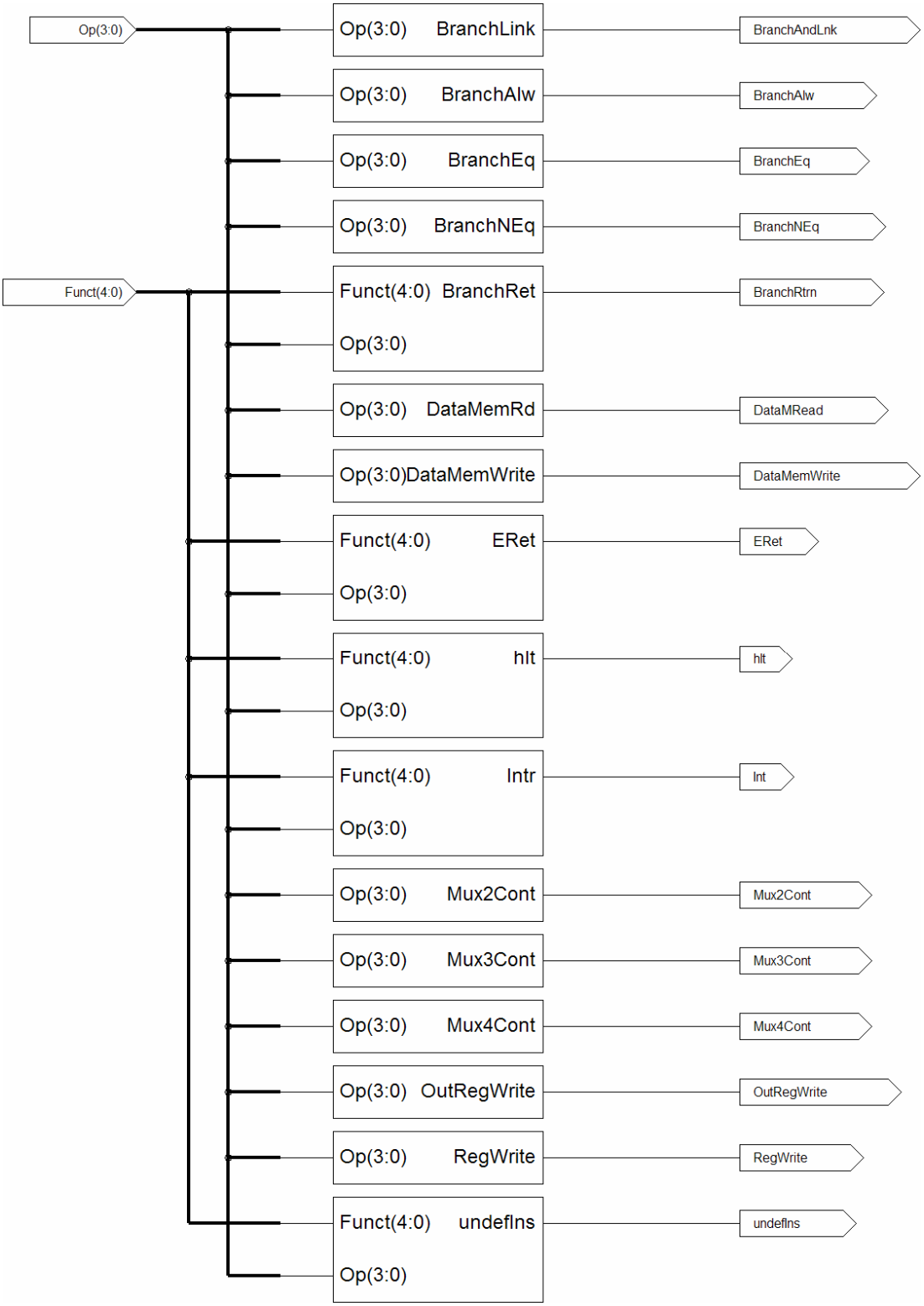

Figure 3.6 Control Unit Block Diagram

Table 3.2 Data Memory Write and Read Control Signals

Instruction	Op field	Logic Expressions
Lw	0111	DataMRead = $\overline{Op(3)}$ and Op(2) and Op(1) and Op(0)
Sw	1000	DataMemWrite= Op(3) and $\overline{Op(2)}$ and $\overline{Op(1)}$ and $\overline{Op(0)}$

Mux2 is in the 3rd stage and selects the destination addresses. If the instruction is R-type, destination address is hold in the RD field and if it is I-type destination address is hold in the RT field. Because the OP field of the R-type is 0000 Mux2Cont signal is generated according to the below expression.

$$Mux2Cont= \overline{Op(3)} \text{ and } \overline{Op(2)} \text{ and } \overline{Op(1)} \text{ and } \overline{Op(0)}$$

Mux3 is in the 3rd stage and selects the data from the register file if the instruction is R-type or selects the data in the instruction if the instruction is I-type. Because the R-type instructions OP field is 0000 the logic circuit for Mux2Cont is set as

$$Mux3Cont= \overline{\overline{Op(3)} \text{ and } \overline{Op(2)} \text{ and } \overline{Op(1)} \text{ and } \overline{Op(0)}}$$

Mux4 is in the write back stage and is used for the selection of the obtained data from the memory or the result of the ALU unit which is written to the register file. Mux4Cont expression is:

$$Mux4Cont= \overline{\overline{Op(3)} \text{ and } Op(2) \text{ and } Op(1) \text{ and } Op(0)}$$

Branch instructions, system instructions and Sw (store word) instruction's results are not related to the register file. So when these instructions reach the write back stage register file's write control input must be disabled. OP fields of these instructions are 8, 9, A, B, C, E, F in hexadecimal. In another expression if instruction is Sw or Beq or Bne or Ba or BL or Nop or Hlt or Syscall or Lret then RegWrite is disabled. The circuit for register write enable control is shown in the Figure 3.7.

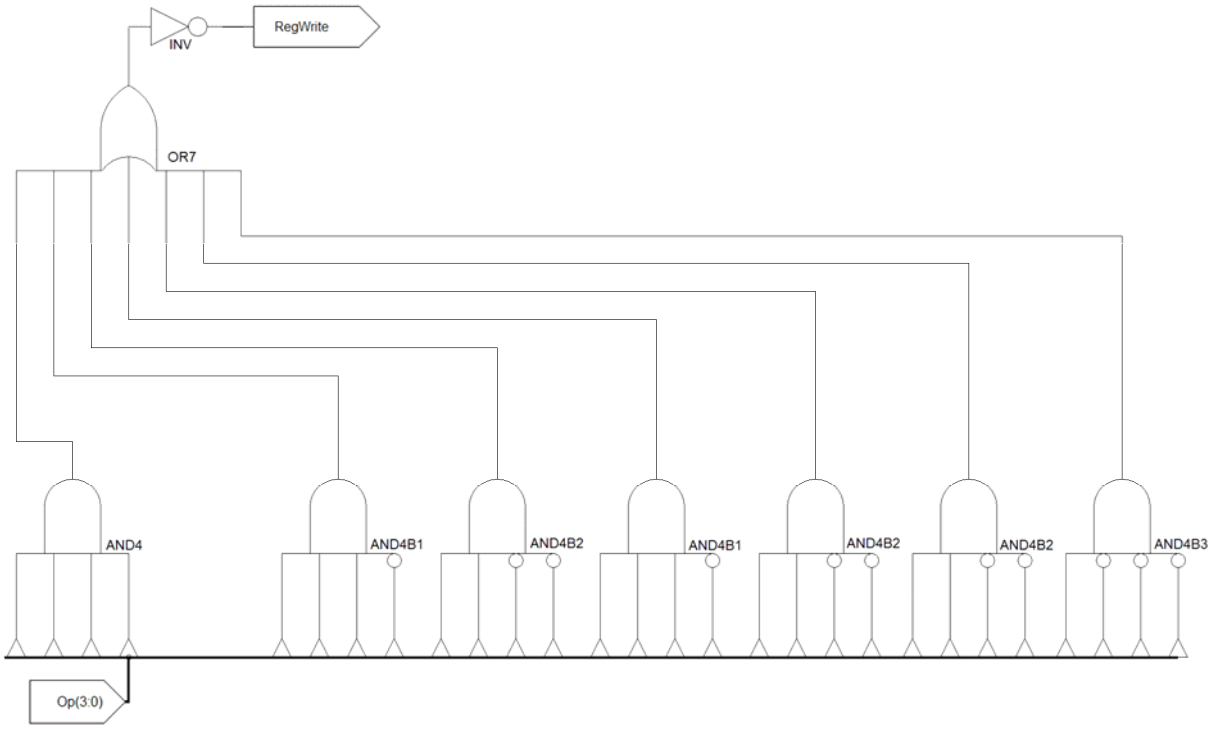

Figure 3.7 Register Write Enable Control Circuit

S-type instructions are recognized with both the OP field and FUNCT field. FUNCT and OP bits of the Syscall, Hlt, Lret and Eret instructions are used for generate control signals. Logic Expressions of the control signals are shown in the Table 3.3, F is used for FUNCT.

Table 3.3 Control Signals of the S-type Instructions

Instruction	OP	FUNCT	Logic Expressions
Syscall	1110	00010	$\text{Int} = \left(\text{Op}(3)\ \text{and}\ \text{Op}(2)\ \text{and}\ \text{Op}(1)\ \text{and}\ \overline{\text{Op}(0)}\right)$ or $\left(\overline{\text{F}(4)}\ \text{and}\ \overline{\text{F}(3)}\ \text{and}\ \overline{\text{F}(2)}\ \text{and}\ \text{F}(1)\ \text{and}\ \overline{\text{F}(\text{O})}\right)$
Hlt	1110	00001	$\text{Hlt} = \left(\text{Op}(3)\ \text{and}\ \text{Op}(2)\ \text{and}\ \text{Op}(1)\ \text{and}\ \overline{\text{Op}(0)}\right)$ or $\left(\overline{\text{F}(4)}\ \text{and}\ \overline{\text{F}(3)}\ \text{and}\ \overline{\text{F}(2)}\ \text{and}\ \overline{\text{F}(1)}\ \text{and}\ \text{F}(0)\right)$
Lret	1110	00011	$\text{BrRtrn} = \left(\text{Op}(3)\ \text{and}\ \text{Op}(2)\ \text{and}\ \text{Op}(1)\ \text{and}\ \overline{\text{Op}(0)}\right)$ or $\left(\overline{\text{F}(4)}\ \text{and}\ \overline{\text{F}(3)}\ \text{and}\ \overline{\text{F}(2)}\ \text{and}\ \text{F}(1)\ \text{and}\ \text{F}(0)\right)$
Eret	1110	00100	$\text{ERet} = \left(\text{Op}(3)\ \text{and}\ \text{Op}(2)\ \text{and}\ \text{Op}(1)\ \text{and}\ \overline{\text{Op}(0)}\right)$ or $\left(\overline{\text{F}(4)}\ \text{and}\ \overline{\text{F}(3)}\ \text{and}\ \text{F}(2)\ \text{and}\ \overline{\text{F}(1)}\ \text{and}\ \overline{\text{F}(0)}\right)$

There is one more logic sub-unit in the control unit. This sub-unit controls if the instruction is undefined. Defined instruction conditions are:

1) When OP=0_{hex} FUNCT must be in the interval 1 to 13 (decimal)
2) When OP=E_{hex} FUNCT can only be 1,2,3,4 (decimal)

Then

- First undefined condition is tested as

  if OP=0 and (Funct=0_{hex} or Funct[3 downto 1]=7_{hex})

- Second undefined condition is tested as

  If OP=0 and (not(Funct=1_{hex} or Funct=2_{hex} or Funct=3_{hex}))

The circuit that performs these tests is shown in the Figure 3.8.

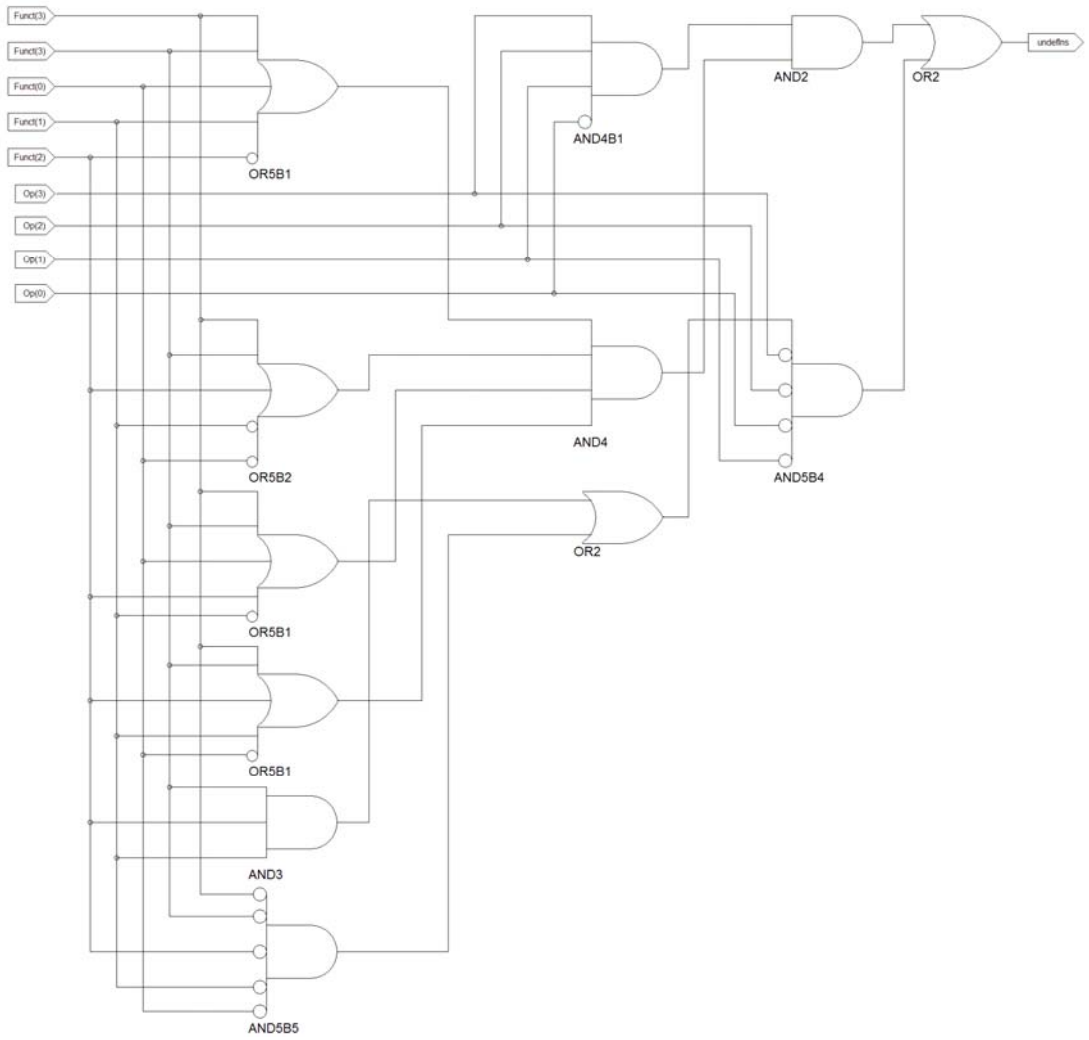

Figure 3.8 Undefined Instruction Control Subunit

3.2.2.2. Register File

Registers are the fast memory units located inside of the processor. One of the RISC processor design principle is RISC processors have large number of registers (Dandamudi, 2004) This principle supports register to register operations and reduces memory accesses. In our design instruction 6-bit RT, RS and RD fields address the registers. 64 registers constitutes the memory unit named as register file.

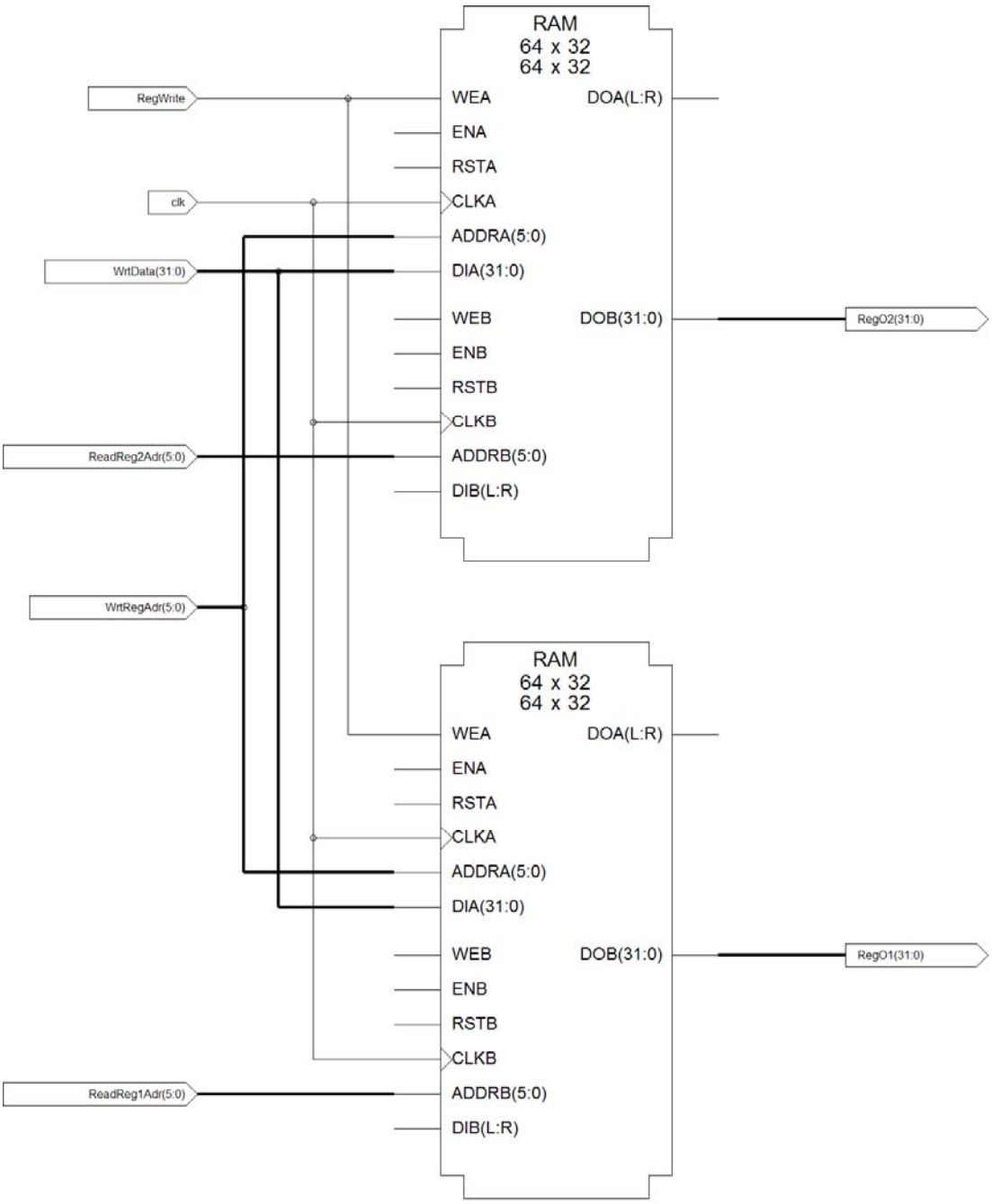

Figure 3.9 Register File

3.2.2.3. Sign Extend

Immediate instructions have 16 bit data on it. 16-bit data is executed with a register which is 32 bit. This operation can be implemented if the 16-bit immediate

data is converted to 32-bit data. Sign extend unit converts 16 bit to 32 bit. Output's 16 to 31st bits are connected to the 15th bit of the input. This method operates as filling the output's 16 to 31st bits are filled with the left most bit of the input. Magnitude and sign of the number is conserved by this method.

3.2.3. Instruction Execute

After obtaining the data and control signals, arithmetic and logic operations, branch address calculations are performed in the instruction execute stage. Instruction execute stage of the pipeline is shown in the Figure 3.1. Sign extend input of the execute stage holds either the 32-bit immediate data or funct, rd and shift amount data. Sign-extend data's 5-0th bits (func field of the instruction) and opcode of the instruction are used in the AluOp unit to select the required result of the ALU's sub units. ALU has two data inputs. First input is the data obtained from the register file. Second data is either the data from the register file or the data on any immediate field. The selection of these two data carry out by a multiplexer named as ALUin2Mux. The result of the ALU is written to the register file according to the register file address fields. If the instruction is R-type the destination address is RD field which is in the sign-extend data. If the instruction is I-type the destination address is RT field. The selection of the destination addresses is done by DestAddrMux. Branch and link instruction requires a register to hold the return address. The next address of the instruction is hold in the ReturnAddress register if this instruction is used. The next instruction address and 15 to 0th bits of the sign extend input (jump amount) are added in the JumpAddAddress unit. The block diagram of the execute stage is shown in the Figure 3.10. The details of the each subunit are given in the following subsections.

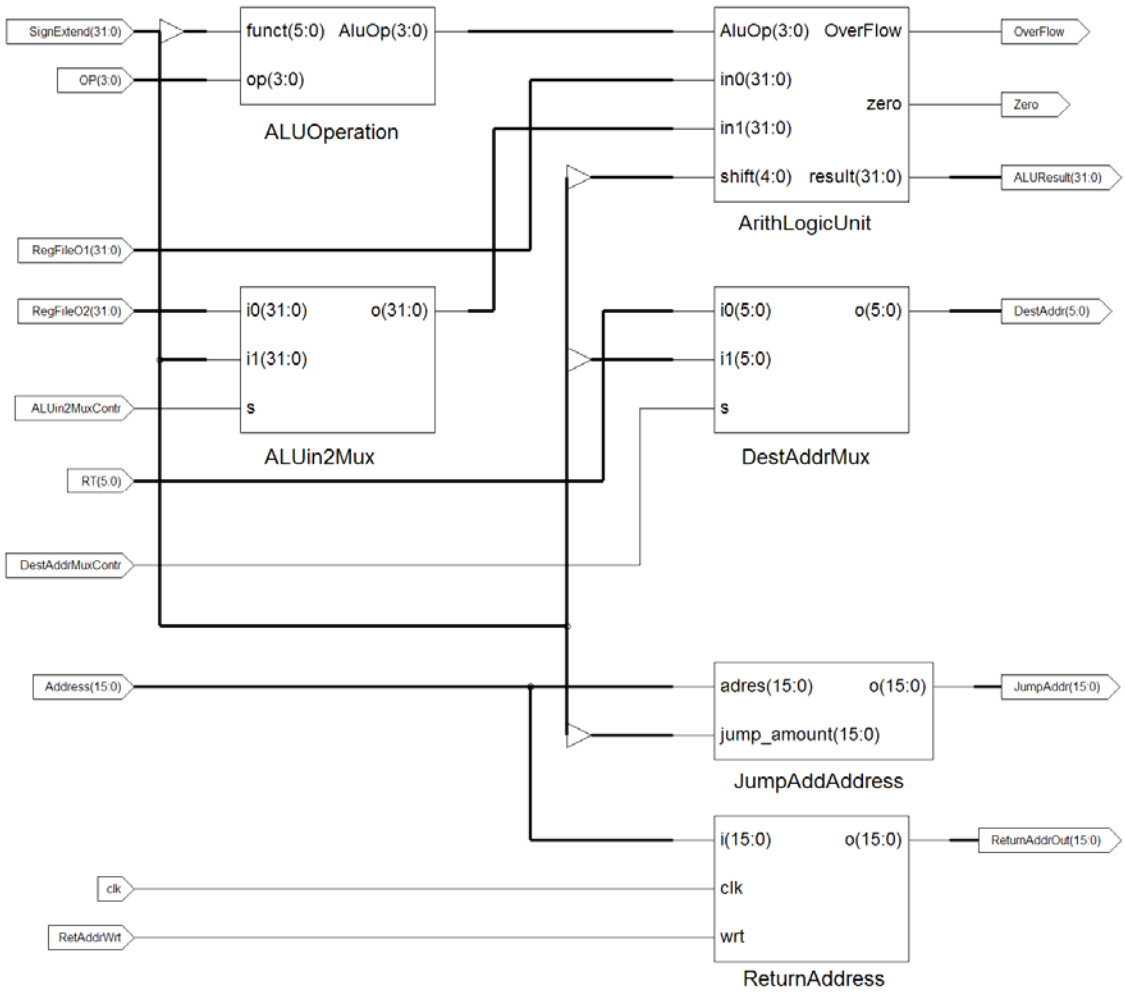

Figure 3.10 Execute Stage

3.2.3.1. Arithmetic Logic Unit

In the arithmetic logic unit (ALU) several arithmetic and logic functions are performed. In0 and in1 are the 32-bit inputs of the ALU. Shift input is required to specify shift amount for arithmetic and logical shift operations. There are 3 output ports. One is for the 32-bitresult. The OverFlow output is generated by the overflow detection unit, which may occur in the addition or multiplication operations. The zero output is used for conditional branch instructions. The block diagram of the ALU is shown in the Figure 3.11.

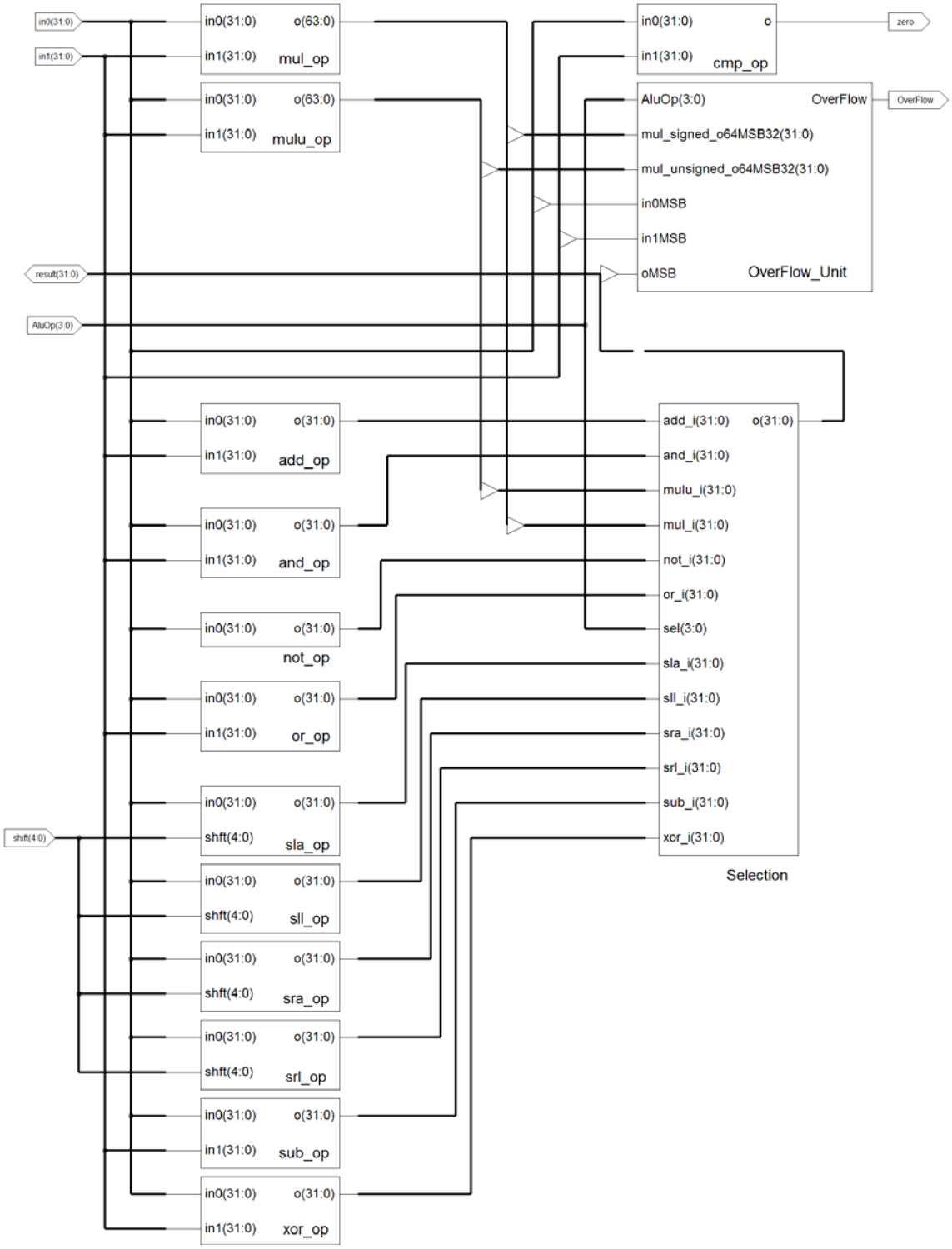

Figure 3.11 Arithmetic Logic Unit Block Diagram

In the Figure 3.11 and, or, not, sra, sll, srl, xor blocks are the logical operation blocks. Mul, mulu, add, sub blocks are the arithmetical blocks. Selection block is a

multiplexer that selects the desired result. Cmp block compares the inputs for equality.

3.2.3.2. Overflow Unit

Overflow detection conditions are designed as follows:

1. Addition overflow condition:
    a. If the inputs' most significant bits are 0s (numbers are both positive) and if the result's most significant bit is 1 result is negative then an overflow exists.

    Example:

    In1: 01101001 $(105)_{10}$
    In2: 01001011 $(75)_{10}$
    Result: 10110100 $(-76)_{10}$

    b. If the inputs' most significant bits are 1 (numbers are both negative) and if the result's most significant bit is 0 then an overflow exists.

    Example:

    In1: 10100000 $(-96)_{10}$
    In2: 10010010 $(-110)_{10}$
    Result: 00110010 $(50)_{10}$

If the inputs most significant bits are 1 and 0 (i.e. they are opposite signed numbers) overflow does not exist.

2. Unsigned multiplication overflow condition

    The product of 32-bit multiplication is 64 bits. So we can determine if there is any overflow by testing the most significant half of the product. If one of the bit is 1 than there exists an overflow in the unsigned multiplication operation. So this circuit can be implemented with an three OR gates.

3. Signed Multiplication overflow condition
    a. If multiplier and multiplicand both are positive or negative, the result is positive. In this condition, if any of the bits between 63 and 31 is 1 then there is an overflow.

b. If multiplier and multiplicand have opposite signs, the result is negative. So 31 to 63th bits must be 1s. These bits are ANDed if there are any 0s the result of the AND operation is zero.

Signed multiplication overflow detection circuit is shown in Figure 3.12.

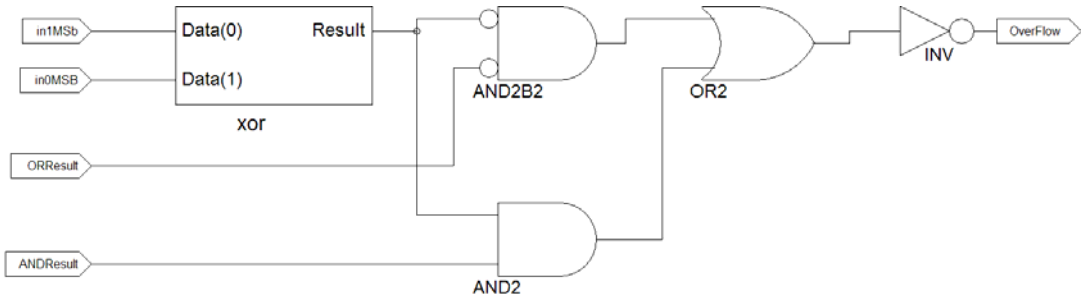

Figure 3.12 Signed Multiplication Overflow Detection Circuit

3.2.3.3. Compare Unit

Compare unit is used for conditional branch instructions, Beq and Bne. Compare operation is formed with a 32 bit bitwise xor operation and the output bits of the operation are NORed. If result of NOR is 1, operands are equal.

3.2.3.4. And, Or, Not, Xor Subunits

And, Or, Xor subunits perform logical bitwise operations. The subunits consist of arrays of 32 AND, 32 OR and 32 XOR gates. Not operation has one input and it inverts each operand.

3.2.3.5. Shifting Units

a. The Shift Left Logical Unit

This unit shifts the input to the left logically, i.e. emptied bits are filled with 0s after shifting to the left. Shifting amount is specified with the input shft which is 5

bits, so 31 level digits is possible. The synthesized schematic is shown in the Figure 3.13

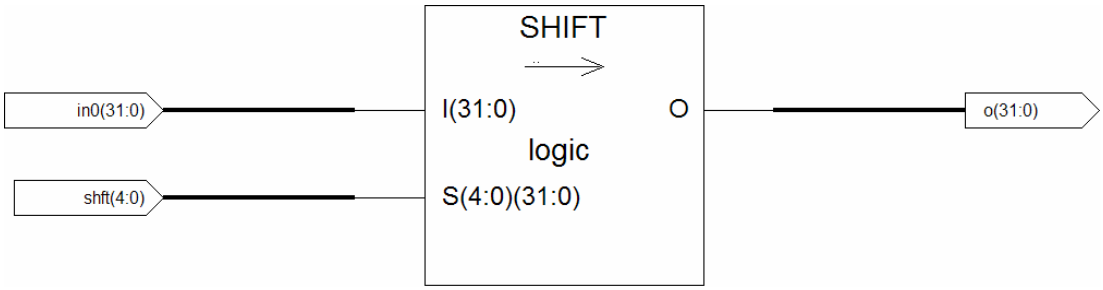

Figure 3.13 The Shift Left Logical Unit

b. The Shift Right Logical Unit

This unit shifts the input to right logically, i.e. after shifting operation the emptied bits are filled with 0s. Shift amount is again 5 bits so the input can be shifted 31 times to the right. The synthesized schematic is shown in the Figure 3.14.

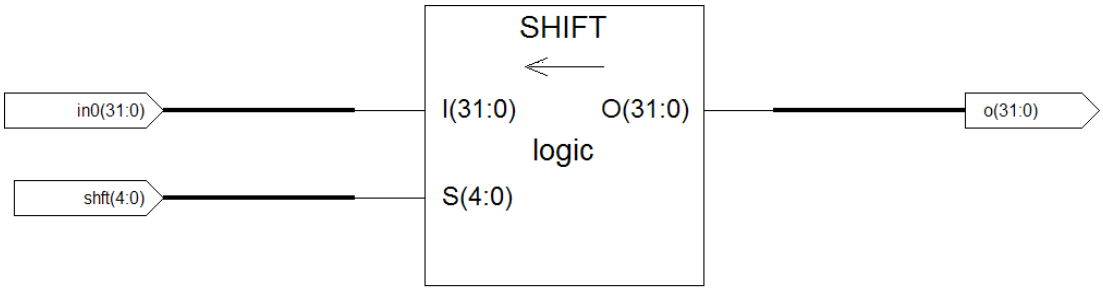

Figure 3.14 The Shift Right Logical Unit

c. The Shift Right Arithmetic Unit

Shift right arithmetic unit fills the emptied bits with the least significant bit after shifting. Shift amount input is 5-bits. The synthesized circuit is shown in the Figure 3.15.

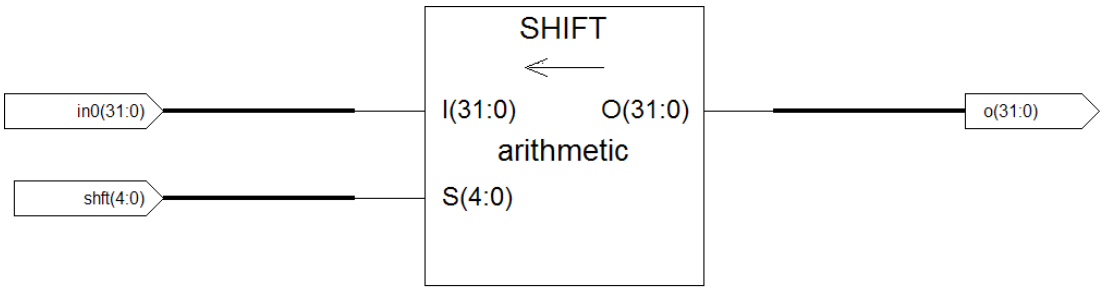

Figure 3.15 The Shift Right Arithmetic Unit

3.2.3.6. The Adder Unit

Addition operation is performed by adder unit. The adder unit performs the signed addition operation. The inputs and output of the addition are 32-bits. Tte rtl shematic of this unit is shown in the Figure 3.16.

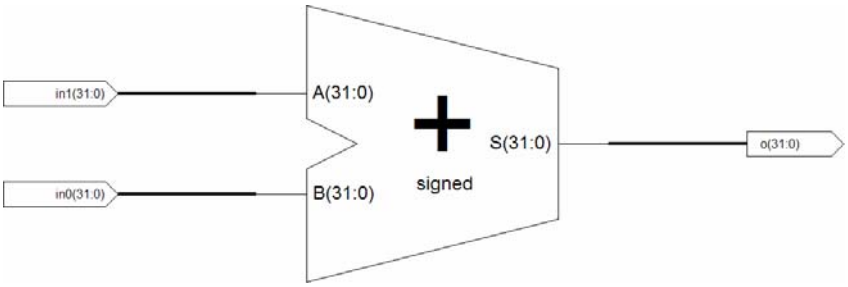

Figure 3.16 The Adder Unit

3.2.3.7. The Unsigned Multiplication Unit

This unit multiplies two unsigned 32-bit numbers. The result of the multiplication is 64-bit. The least significant 32 bits are considered as the result and the most significant 32-bits are checked for overflow. The synthesized circuit is shown in the Figure 3.17.

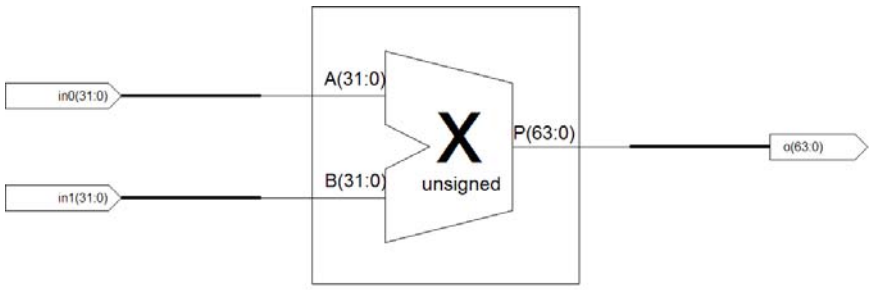

Figure 3.17 The Unsigned Multiplication Unit

3.2.3.8. The Signed Multiplication Unit

The signed multiplication unit multiplies two 32-bit two's complement numbers. Product is 64-bit. The least significant 32 bit is retuned as result and the most significant 32 bits are tested for overflow. The synthesized circuit is shown in the Figure 3.18.

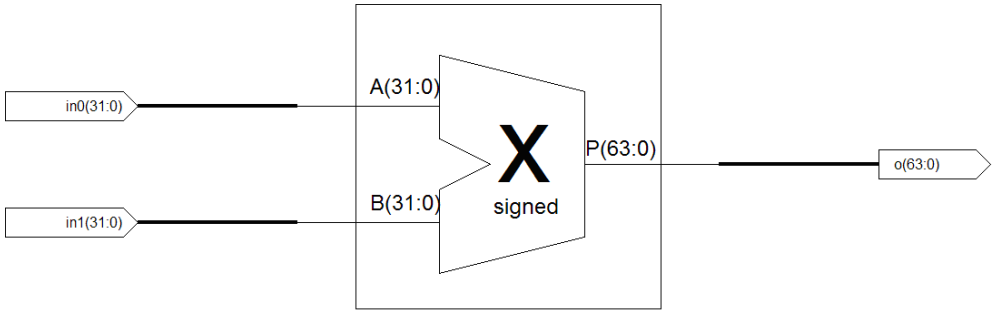

Figure 3.18 Signed Multiplication Unit

3.2.3.9. The ALUOp Unit

In the ALU, all operations are performed in parallel. The result of the required operation is sent out via a multiplexer. This multiplexer selects the ALU subunits results according to the signals generated in the ALUOp unit.

ALUOp unit gets the instructions' OP and FUNCT fields as inputs. If the instructions' OP field is 0000, the instruction is R-type. So FUNCT field must be controlled to determine which operation is required. ALUOp Signals and conditions are shown in the Table 3.4.

Table 3.4 ALUOp Signal Generation Condiditons

ALU Operation	ALUOp (4-bit hex)	R-type		I-type
		OP(4-bit hex)	FUNCT	OP(4-bit hex)
And	0	0	00100	4
Or	1	0	00101	5
Not	2	0	01100	-
Xor	3	0	00110	6
Sll	4	0	01010	-
Srl	5	0	01011	-
Sla	6	0	01100	-
Sra	7	0	01101	-
Add	8	0	00001	1
Sub	9	0	00010	-
Mul	A	0	00011	3
Mulu	B	0	00111	-

3.2.4. Address Computation Unit

Branch instructions require an adder unit. Adder unit adds the address and the jump amount data on the branch instruction. Relative branching operation is provided by this method. This operation is explained in detail in branch unit section.

3.2.5. Memory Stage

RISC processors access the memory with load/store instructions. Referring to this characteristic the RISC architecture is named as load/store architectures as well. (Dandamudi, 2004). Lw (Load Word) and Sw (Store Word) instructions are detected in the control unit and, control signals of these instructions are generated. The control signals are transferred in the pipeline parallel with the instruction. In the memory stage transferred memory control signals named as MemRead or MemWrite enables

the memory for reading or writing respectively. Address bus is 16 bit and transferred from the previous stage's sign extend output. Again if any data is stored in the data memory, data is sent in the pipeline and reaches the data memory with the name DataIn to the port of the memory WriteData. Any data in the memory is obtained from the Read data output according to the address input. The data memory's RTL schematic is shown in the Figure 3.19.

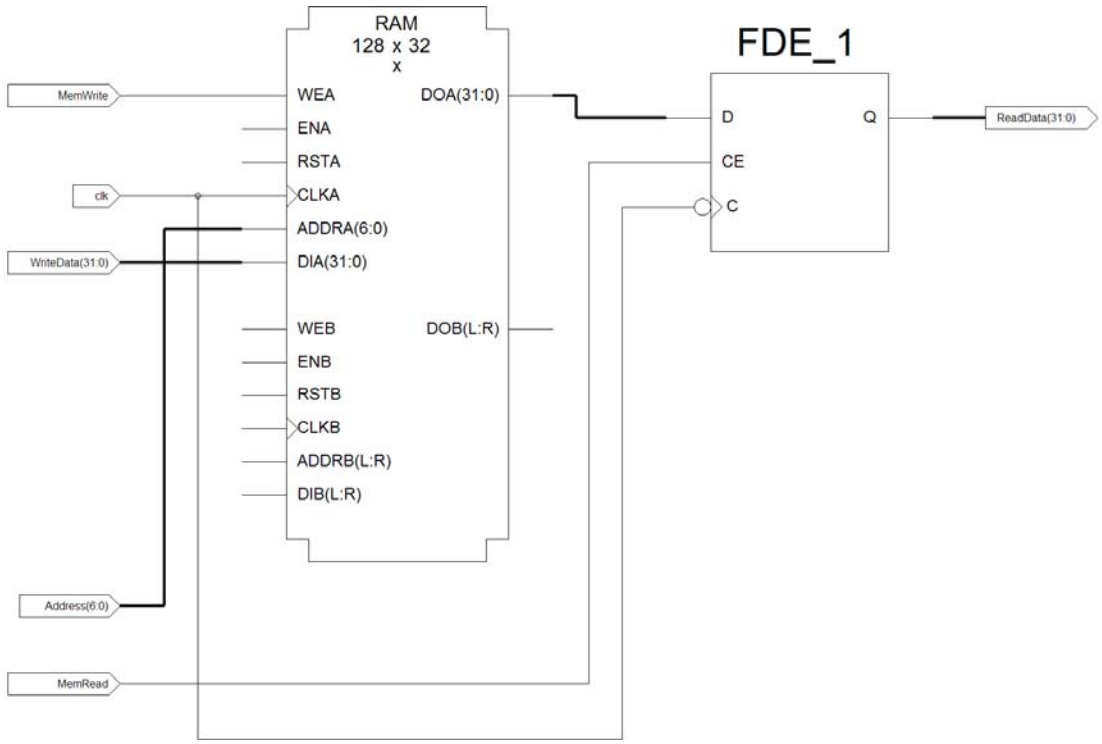

Figure 3.19 Data Memory

3.2.6. Write Back Stage

This stage is constructed to write the result of the ALU or the data obtained from the memory to the register file. The address hold in the RD or RT fields is selected in the 4^{th} stage via a multiplexer. In this stage the data is selected. If the instruction is Lw, the selection input of the multiplexer is 0. If the instruction is related to the ALU the selection input is 1. The data selected by mux, address of the data and the wrt enable signals are sent to the register file.

4. PIPELINE HAZARDS

Pipeline method divides the subunits of the processor into categories and different categories are used by different instructions at a time. The advantage of the pipeline is shown in the Figure 3.1. However pipeline structure can cause some problems. In this section these problems and the solutions are examined.

4.1. Structural Hazards

Structural hazards occur when a unit of the processor is attempted to use by more than one instruction. This situation could occur in memory access if the memory unit is not separated as data memory and instruction memory. If one memory is used, the structural hazard could be eliminated by stalling. Because instruction address is sent to the memory almost every clock cycle, data write could have priority to the instruction read address. Stalling reduces the performance of the processor.

4.1.1. Data Hazards and Forward Unit

The purpose of the pipeline design is reducing instruction execution rate to the one instruction per clock period. In our design we can examine this situation with independent instructions. Consider the following instruction sequence:

```
Movi R2,15
Movi R3,21
Movi R5,13
Movi R4,15
Movi R6,20
Movi R7,15
Movi R8,32
```

These instructions store 15, 21, 13, 15, 20, 15, 32 to the 2, 3, 5, 4, 6, 7,8th registers respectively. The result of the simulation is shown in the Figure 4.1. pc_o signal represents the output of the program counter in the fetch stage. Instructions are

written to the instruction memory at the beginning of the simulation. Figure 4.1 shows only units that contribute the execution of these instructions. Related units of the instructions are shown in the simulation figure. Each instruction is obtained from the instruction memory in the first stage (inst), signext_o is used for the output of the sign extend unit and in the second unit. Aluresult is the result of 3rd stage (execution stage). aluresult_o is the output of the EX/MEM register in he 4th stage. regfiledata_i is the output of the last stage. Rgfile (and subsignals) shows the register file content. At the end of the 5th clock the first result is written to the register files related register. The following results are written to the register file each are one clock period later. As seen in the simulation completion of one instruction per clock target is achieved for this instruction sequence.

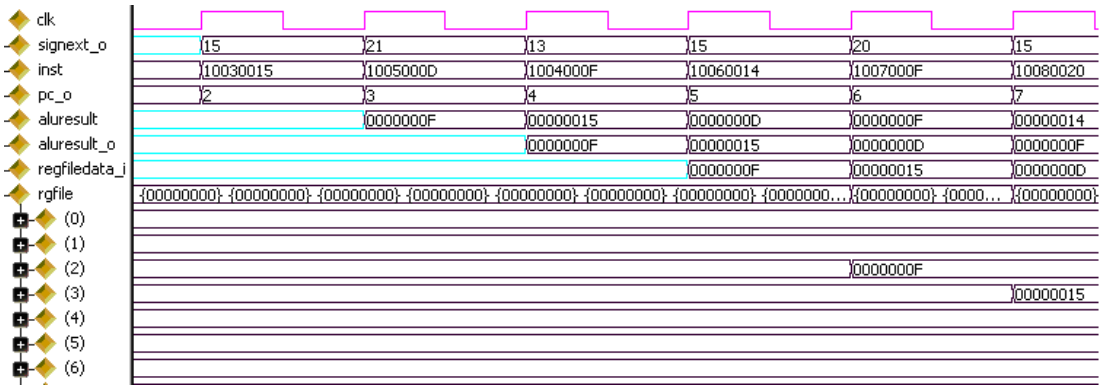

Figure 4.1 Simulation of the Independent Instructions.

Let's consider the execution of the dependent instructions.

Example:

Movi R2, 15
Addi R4, R2, 22 # Reg4=Reg2+22
Movi R3, 21

In this sequence Addi instruction is dependent to movi since it uses register 2 which is written by Movi.

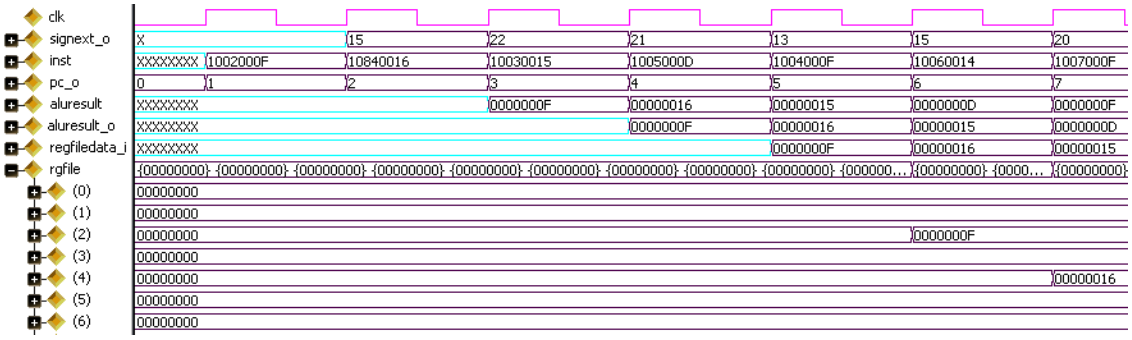

Figure 4.2 First Type Data Dependency Simulation

As shown in the Figure 4.2 Movi R2, 15 instruction writes its result in the 6th clock cycle. Following Addi R4, R2, 22 instruction needs the result of the previous movi instruction in the 3rd clock's rising edge. So there occurs a data hazard.

If we change the second and third instructions sequence as below, the simulation result is as in the Figure 4.3.

Movi R2, 15
Movi R3, 21
Addi R4, R2, 22

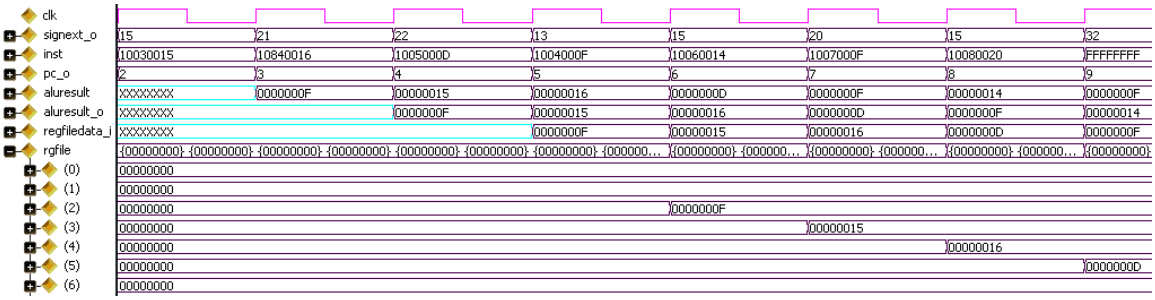

Figure 4.3 Second Type Data Dependency Simulation

As seen in the Figure 4.3, first instruction writes its result in the 6th cycle, and second instruction writes its result in the 7th cycle. 3rd instruction attempts to use 1st instruction's result in the 4th cycle. So the first cycle is late for 2 cycles. So even after rescheduling, there exists a data dependency. the correct execution can be achieved by stalling the pipeline until Movi generatetes the result. However stalling

reduces the performance so this processor uses forwarding technique to solve data dependency.

The forwarding is shown in Figure 4.4. The forwarding operation can be done with two data selectors. The conditions of these data dependencies can be formulated with comparing the fields of the destination addresses of the preceding instruction and sources of the following instruction (Patterson, 2005).

Notation for the register file addresses fields in the pipeline registers as follows:

EXMEM.RegRD: The Register's destination address field in the Execute/Memory pipeline register.

MEMWB.RegRD: The Register's destination address field in the in the Memory/Write Back pipeline register.

IDEX.RegRS: The address field of the register in the Instruction Decode/Execute pipeline which is used as source.

IDEX.RegRT: The address field of the register in the Instruction Decode/Execute pipeline which would be the second source of the ALU.

The conditions that required to forward data are:

1. If the RD field in the EX/MEM register is equal to the RS field in the ID/EX register then forward the ALU result in the EXMEM to the first input of the ALU.
2. If the RD field in the EX/MEM register is equal to the RT field in the ID/EX register then forward the ALU result in the EXMEM to the second input of the ALU.
3. If the RD field in the MEM/WB register is equal to the RD field in the ID/EX register then forward the write back data mux's output to the first input of the ALU.
4. If the RD field in the MEM/WB register is equal to the RT ield in the ID/EX register then forward the write back data mux's output to the second input of the ALU.

With the above controls there must be additional controls. These controls are related to the instruction type. Namely the instruction in the EM/MEM or MEM/WB pipeline register must write the result to the register file. Otherwise there is not any data dependency. Destination address must not be the 0^{th} register which is set to 0. This condition is added to the forward unit too. Last condition that must be control is that if the data dependency occurs between ID/EX – EM/MEM and ID/EX – MEM/WB at the same time. ID/EX – MEM/WB data dependency must have priority. Because the instruction is depend primarily to the instruction which is going into the execution first.

After considering all these conditions, forward unit control is adapted as (Patterson, 2005):

1. If {(EX/MEM.RegWrite=1) And (EX/MEM.RegRD!=’000000’)And (EX/MEM.RegRD= ID/EX.RegRS)} Then forward ALU result at the EXMEM Register to first ALU input.
2. If {(EX/MEM.RegWrite=1) And (EX/MEM.RegRD!=’000000’)And (EX/MEM.RegRD= ID/EX.RegRT)} Then forward ALU result at the EXMEM Register to second ALU input.
3. If {(MEM/WB.RegWrite=1) And (MEM/WB.RegRD!=’000000’)And (EX/MEM.RegRD!= ID/EX.RegRS) And (MEM/WB.RegRD=ID/EX.RegRS)} Then forward writeback mux output to first ALU input.
4. If {(MEM/WB.RegWrite=1) And (MEM/WB.RegRD!=’000000’)And (EX/MEM.RegRD!= ID/EX.RegRT) And (MEM/WB.RegRD=ID/EX.RegRT)} Then forward writeback mux output to second ALU input.

The circuit of the forward unit is constructed as in the Figure 4.4. In the circuit there are 6 comparators are used. 4 are 6-bit equal comparators and 2 are 6-bit not equal comparators.

Forward unit generates the control signals for the forwarding multiplexers. These multiplexers are added to the inputs of the ALU. Second data input of the ALU accepts data either from the register file or from the sign extend unit. If the instruction is I-type, there is no need to forward any data. Because of this reason forward mux is connected to the input of this register file – sign extend data selector. Arithmetic logic unit and its input side becomes as shown in the Figure 4.5.

After addition of the forward unit and forward unit multiplexers, the simulation of

```
Movi R2, 15
Addi R4, R2, 22
Movi R3, 21
```

instructions are shown in the Figure 4.6. When the destination address in the EX/MEM register is equal to the RS field in the ID/EX register at the 4^{th} clock period, Forward unit generates the control of the Forward unit mux “10” and the ALU result in the EX/MEM register is forwarded to the first ALU input. The result of the instruction is written to the register file in the 7^{th} clock as 35 decimal which is sum of 15 and 22.

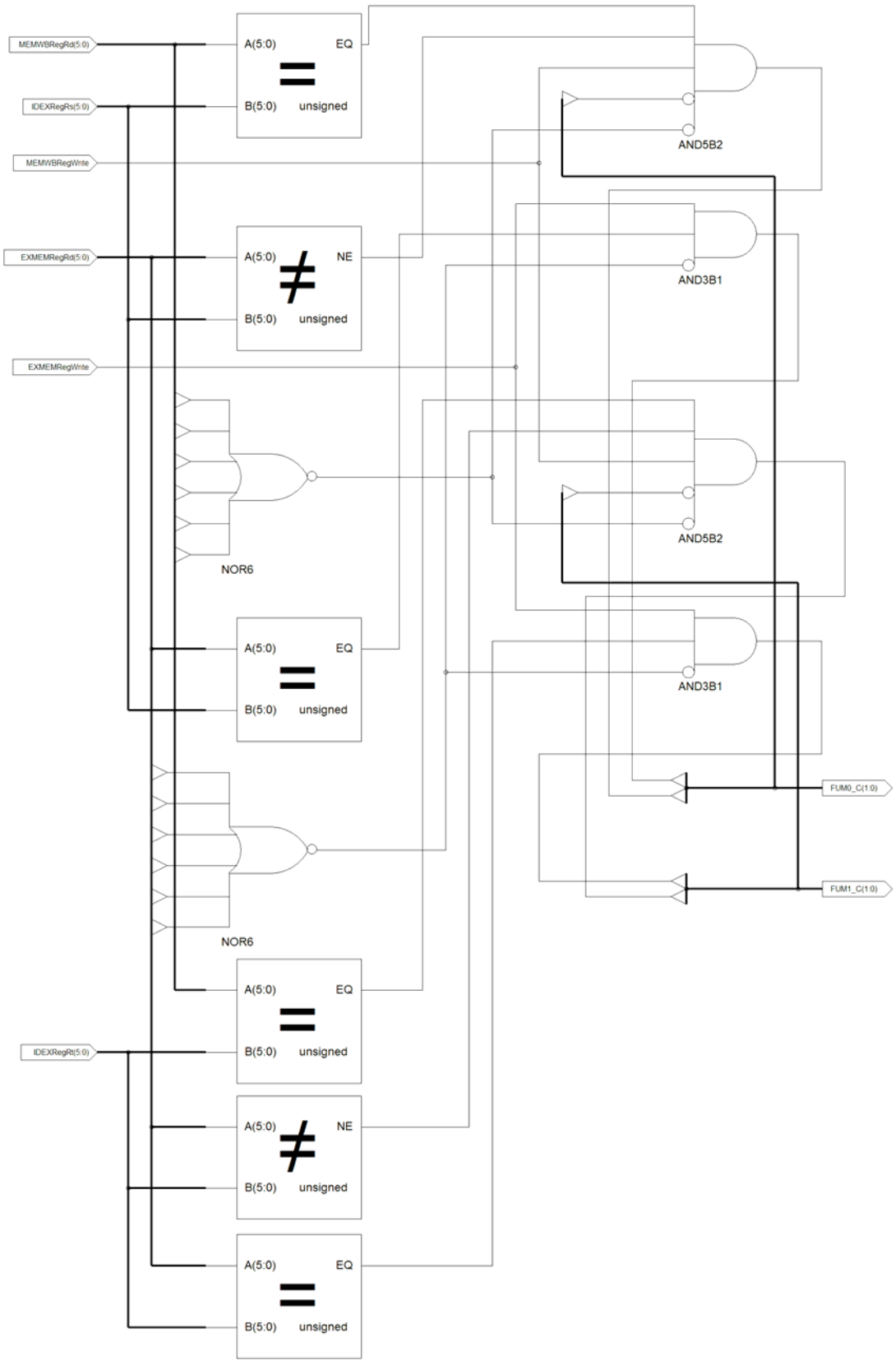

Figure 4.4 The Forward Unit

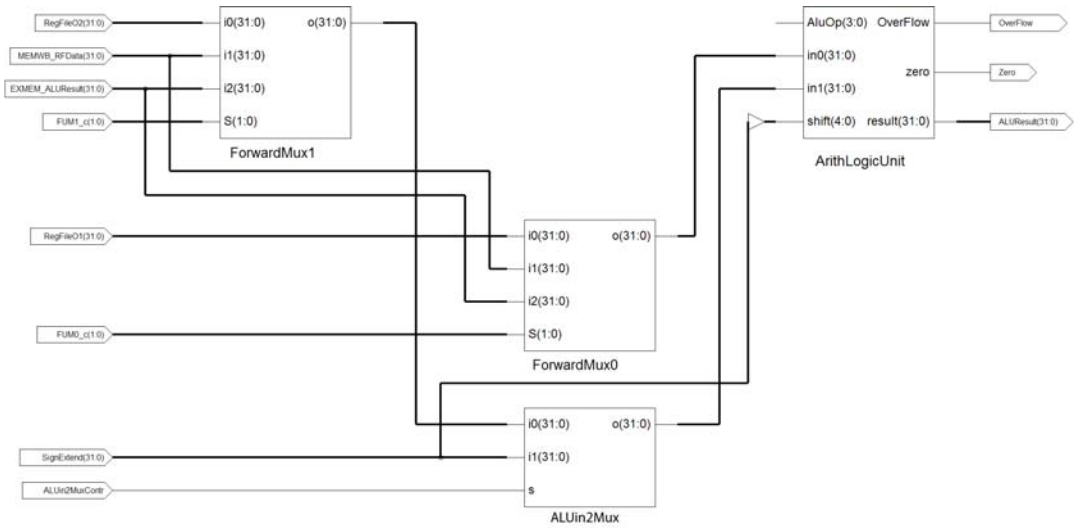

Figure 4.5 ALU with the Forward Unit Multiplexers Connected

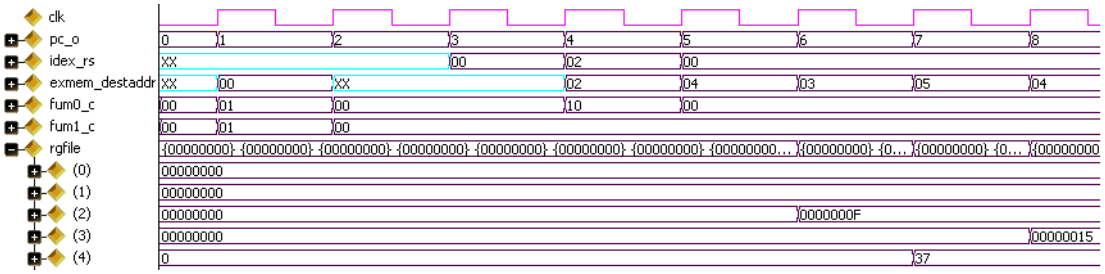

Figure 4.6 EX/MEM Forward Simulation

To demonstrate of second forward type, the above instruction sequence can be considered.

Movi R2, 15
Movi R3, 21
Addi R4, R2, 22

In this example when the destination address in the MEM/WB register is equal to the source address in the ID/EX register, forward operation must be done to accurate result. The simulation of this situation is shown in the Figure 4.7.

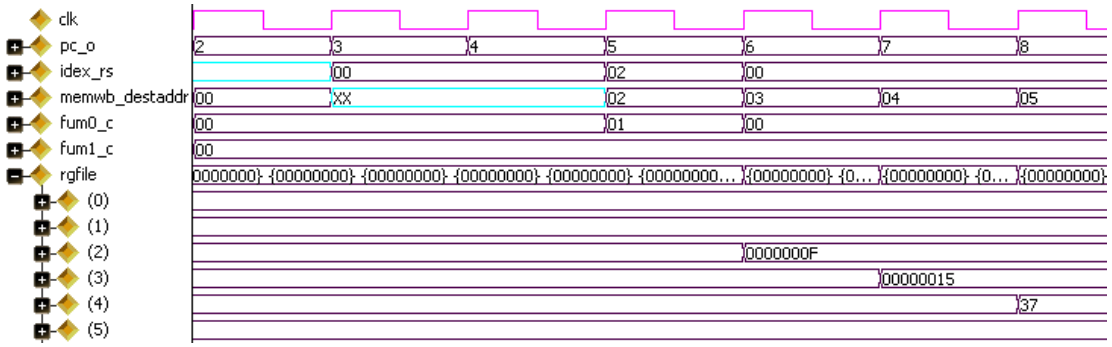

Figure 4.7 Simulation of the Instruction Sequence of Second Type Forwarding

MEM/WB resiter destination location and ID/EX register RS field becomes same at the 5th clock. Forward Unit generates the control signal as "01" for the first input of the ALU. The result of the instruction is written to the Register File (35 in decimal) in the 8th clock accurately.

4.1.2. Data Memory Dependency Hazard

The designed processor has 2 instructions that access the memory. These are Lw (Load Word) and Sw (Store Word). Lw instruction reads the data memory and writes the data to the register file. The following instruction sequence is used to demonstrate data memory dependency hazard.

```
Movi R2, 10
Lw   R3, 3
Add  R4, R2, R3
```

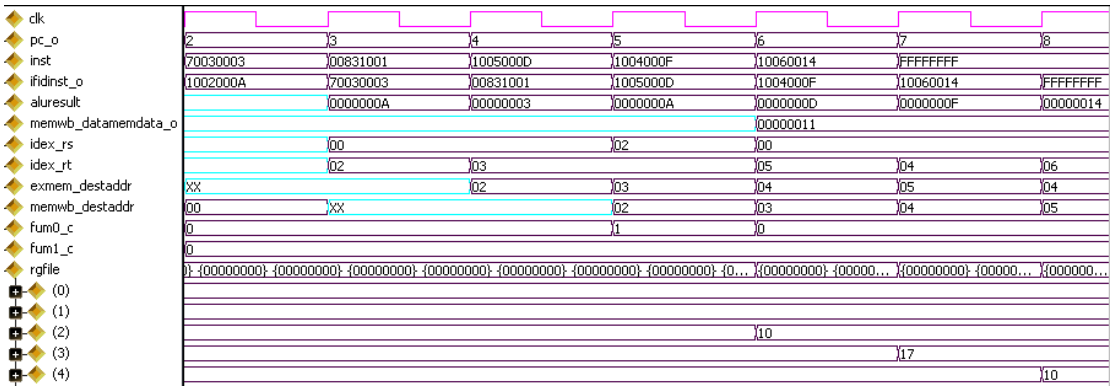

Figure 4.8 Data Memory Dependency Hazard

As seen in the Figure 4.8, the data memory data is in the MEM/WB register in 6th clock. But following Add instruction requires the data in the 5th clock cycle for the input of ALU. Because the required data is not obtained yet, forward unit is not sufficient for this operation. The result is written as 10 to the 4th register which is the sum of 10 and 0 in the 8th clock. But the result has to be 10+17=27.

This problem can be solved with delaying the following Lw related instruction for one clock cycle period. Delay operation can be done with disabling write input of the PC and installing a Nop instruction to the pipeline. This operation is named as stall or bubble insertion to the pipeline.

This hazard condition can be recognized from the ID/EX and ID/ID registers. Data memory read bit of the ID/EX register must be 1. RT field of the ID/EX register (destination address of the Lw instruction) have to be controlled with the previous pipeline register's source address fields, i.e. IF/ID RS and RT fields.

Data memory dependency hazard detection unit is constructed as in the Figure 4.9.

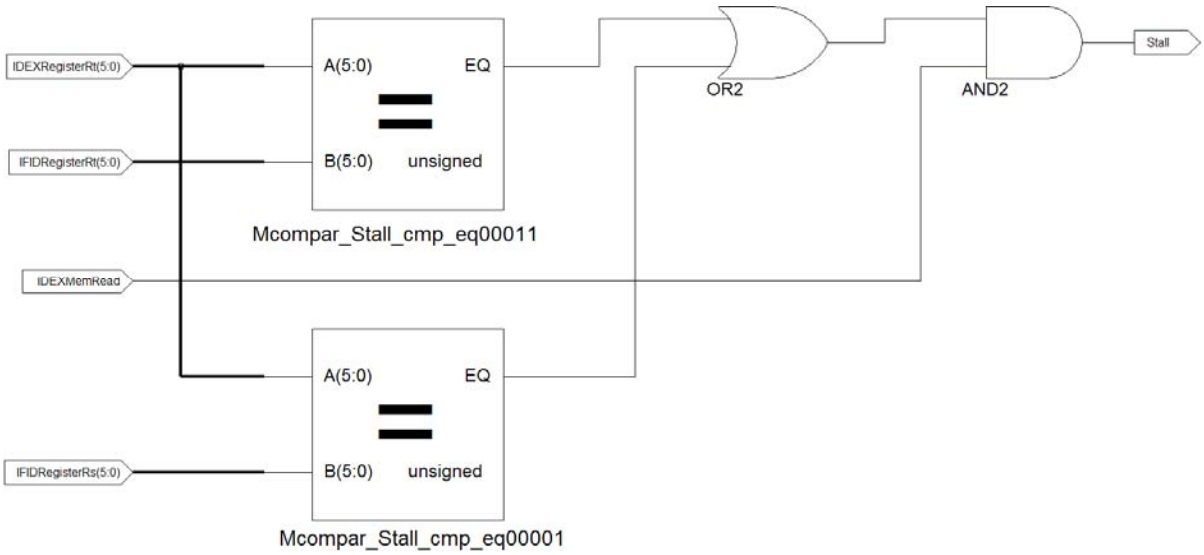

Figure 4.9 Data Memory Data Hazard Detection Unit

After adding the memory data hazard unit, at the 4th clock Add instruction is in the IF/ID register while the Lw instruction is in the ID/EX instruction. Data hazard

detection unit detects there is an instruction follows the Lw which is using the destination address of the Lw and generates the required stall control. Simulation is shown in the Figure 4.10.

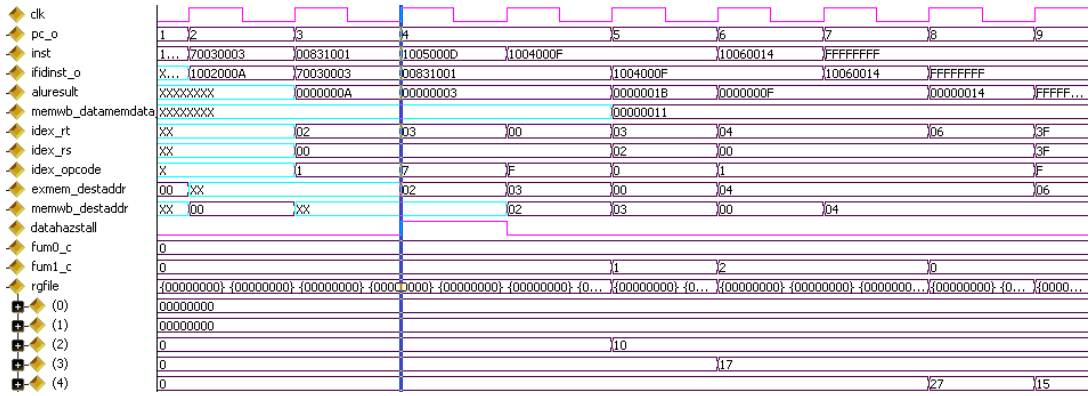

Figure 4.10 Simulation of the Instructions with Data Hazard Detection Unit

Datahazstall signal is connected to the flush input of the ID/EX register , inverse of the signal is connected to the write input of the PC and IF/ID register. When datahazstall signal is enabled, PC and IF/ID register hold the data, while ID/EX register accepts a Nop instruction which causes bubble in the pipeline. This causes increasing the gap between two following instructions to 2 clock cycle periods, so when Add instruction requires the data forward unit can provide. Add instruction writes the result as 10+17=27 to the register file at the 8th clock which is one clock cycle later than normal operation time.

4.2. Branch Hazards

The result of the branch unit controls the mux connected to the PC. Branch unit is shown in the Figure 4.11.

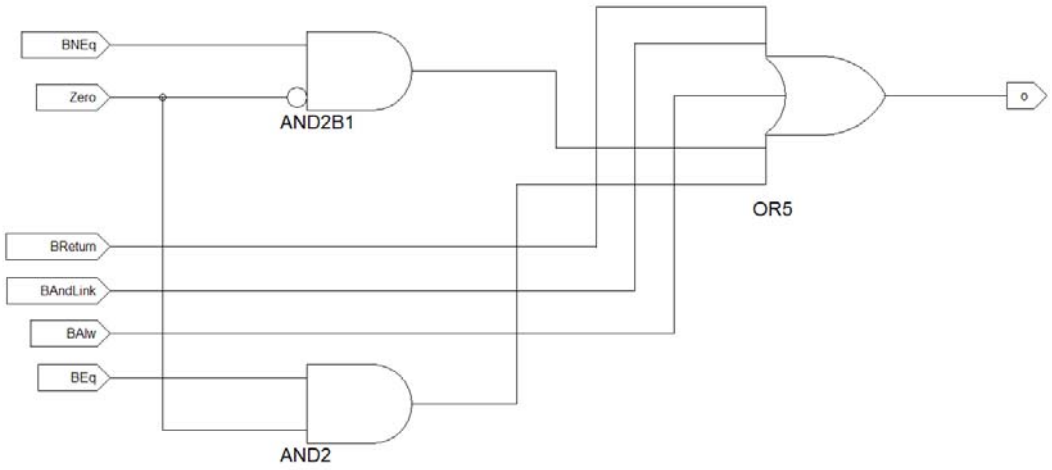

Figure 4.11 Branch Unit

There are two conditional branch instructions in the instruction set architecture. When these instructions are used either branching can occur or not. Branch address calculations are done in the execute stage. These calculations are involved address addition and testing the registers for equality. The comparison operation is done by ALU's compare subunit. If it is the address calculated by branch adder is sent to the mux connected to the PC. So PC can hold the branching address at the next clock. But there another problem occurs in this case. Since there are 2 instructions have already entered the pipeline before branch instruction decision is made then there are instructions that must not be executed in the pipeline.

The problem can be solved in two ways:

1. When any branch instruction is fetched from instruction memory; processor can wait for the result of the branch instruction. The following instructions execute in the processor after result instructions.
2. Branch instructions and following instructions are executed in ordinary sequence. If branching is needed two instructions are removed from the pipeline.

First method reduces the performance, because if branch instruction does not require branching, 2 clock period unnecessary delay occurs.

We can examine the dataflow in the design with the following instructions:

```
Movi R5, 10
Movi R3, 12
Ba 3
Movi R5, 21
Add R2, R3, R3
Subi R5, R1, 7
Mul R4, R3, R5
Movi R7, 12
Movi R6, 5
Movi R8, 20
```

After executing first 3 instructions, it is expected that 7th instruction have to be executed. But while branching operation is expecting in the 3rd stage of the pipeline, there executed more 3 instructions. This situation is simulated as in the Figure 4.12.

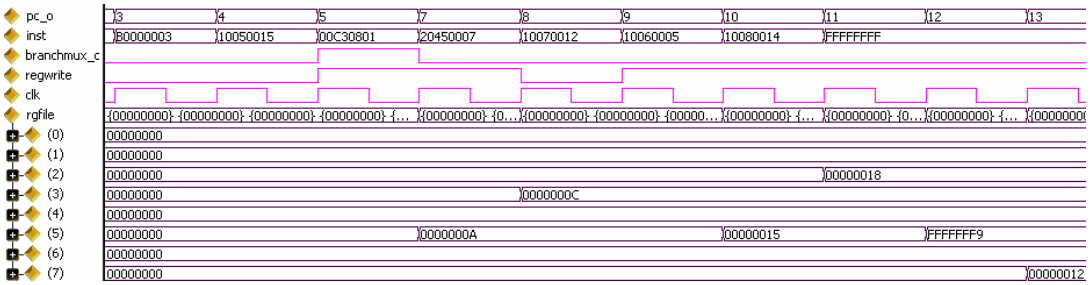

Figure 4.12 Simulation of the Instructions with a Branch Instruction

Results of the instructions are saved in the register file (rgfile in the Figure 4.12). First and second instructions' results are written when the PC holds 7 and 8 in decimal. But after branch instruction 4, 5 and 6th instructions are written when PC holds 10, 11, 12. These instructions have to be discarded from the pipeline before execution because they cause data errors. This discarding is named as flushing instructions.

When the branching signal is generated there are 2 instructions existing at the output of the IF/ID register and instruction memory. The output of the PC is also cause one more instruction to enter pipeline. The flushing can be done connecting the branch output signal to the flush inputs of the ID/EX and IF/ID registers. So following two instructions can be flushed. But for the 3rd instruction branch signal

must be delayed for one clock cycle period. So this signal is sent to the EX/MEM register and again connected to the IF/ID register. At the next period the 3rd instruction can be discarded by this way. The result is shown in the Figure 4.13.

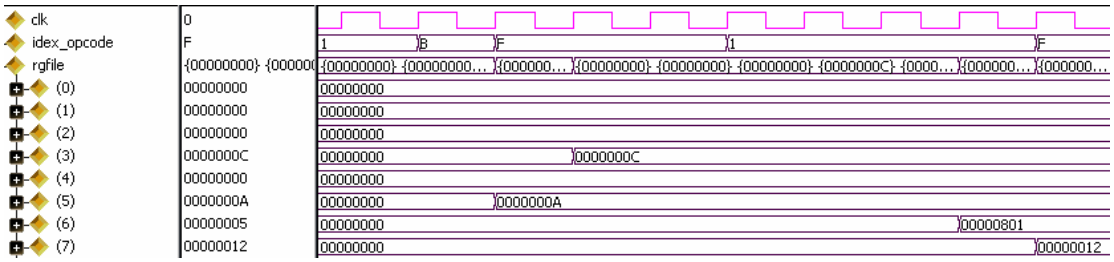

Figure 4.13 Simulation of the Instructions to Show Solved Branch Hazard

4.3. Exceptions

Exceptions are the unexpected situations while program execution continues. The exceptions are defined as overflow and undefined instruction in this thesis. Syscall instruction is another exception since this instruction is used for transferring control to the operating system. The control signals of the undefined instruction and Syscall is generated at the control unit. Overflow is detected in the ALU. So the exception unit is located in the 3rd stage. When exception occurs, the processor stores the instruction which causes exception in the exception program counter (EPC) and jumps to the addresses which are defined for handling exceptions. Since the address+1 is transferred with the instruction parallel and exception is detected in the 3rd stage, the address from ID/EX register will be address+2. So the problem can be solved with transferring the address to the EX/MEM and MEM/WB pipeline registers and connecting the address in the MEM/WB to the EPC register. The instructions following the instruction which causes exception also have to be flushed. The determined addresses are sent to the mux which is connected to the PC. The type of the exception is determined by the addresses that processor jumps. Required handling instructions are determined by the operation system, in the predetermined locations. The control signal of the mux connected to the PC is united with the signal from exception unit. Exception unit provides the jump address of the exception. ERet instruction is used to return the normal execution point after handling exception.

5. SIMULATION RESULTS

The designed processor is run with a simple program which computes the first 10 Fibonacci numbers. Fibonacci numbers are computed according to the following equation:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases} \quad (1)$$

The program is as follows:

```
Movi R5, 9          #1
Movi R3, 0          #2
Movi R1, 0          #3
Movi R2, 1          #4
Movi R3,1           #5
Addi R5, R5, -1     #6
Add R3, R1, R2      #7
Mov R1, R2          #8
Mov R2, R3          #9
Bne R4, 0, -5       #10
Movi R3, 0          #11
```

R5 is used for count 9 down to 0 to compute first 10 Fibonacci numbers. R3 is the register which Fibonacci numbers are stored in order. 6th instruction is used for decrementing. 10th instruction branches to the 6th instruction if 4th and 0th registers are not equal. Finally when the equality occurs, 11th instruction writes 0 to the 3rd register. The results are shown in the Table 5.1.

Table 5.1 Simulation Timing

Fibonacci Number (R3 register)	Period
0	6
1	9
1	11
2	19
3	27
5	35
8	43
13	51
21	59
34	67
55	75

Branch instruction stalls pipeline for 3 periods. First 10 Fibonacci numbers are computed in 75 clock periods while 50 instructions are executed. 45 is executed in the loop 5 is out of the loop. For this example, clock per instruction is 1,5. The simulation is shown in the Figure-5.1. The registers are shown in decimal. The simulation starts at 5 ns and period is 10 ns. 10^{th} number is written to the register file at 755 ns.

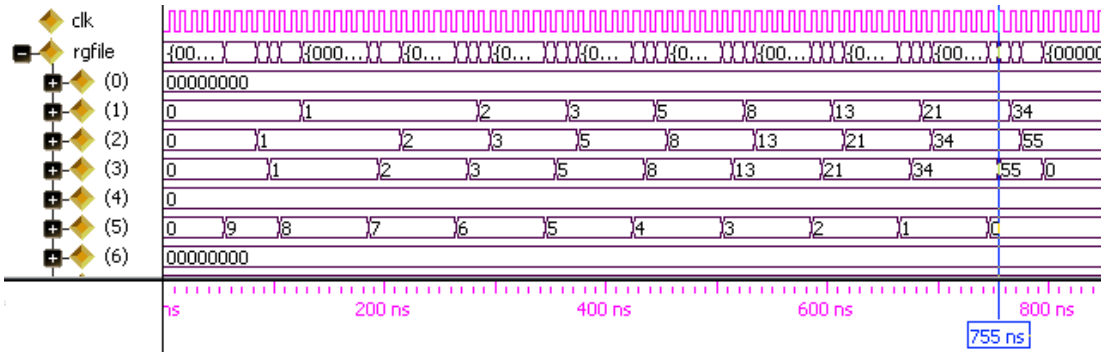

Figure 5.1 Fibonacci Program Simulation

6. SYNTHESIS RESULTS

The target device for synthesis is selected as xc3s250e-4vq100 which is a member of Xilinx Spartan-3E™ FPGA family. This device costs approximately 13.5$ (Nu, 2008). The resource usage of the design is shown in the Table 6.1.

Table 6.1 Device Utilization Summary

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	783	2448	31%
Number of Slice Flip Flops	527	4896	10%
Number of 4 input LUTs	1417	4896	28%
Number of bonded IOBs	34	66	51%
Number of BRAMs	4	12	33%
Number of MULT18X18SIOs	8	12	66%
Number of GCLKs	4	24	16%

The number of logic items used in the processor is shown below:

```
# RAMs                          : 4
        128x32-bit dual-port block RAM    : 1
        128x32-bit single-port block RAM  : 1
        64x32-bit dual-port block RAM     : 2
# Multipliers                   : 2
        32x32-bit multiplier              : 2
# Adders/Subtractors            : 4
        16-bit adder              : 2
        32-bit adder              : 1
        32-bit subtractor         : 1
# Registers                     : 494
        Flip-Flops                : 494
# Latches                       : 6
        1-bit latch               : 1
        16-bit latch              : 1
        32-bit latch              : 3
        4-bit latch               : 1
# Comparators                   : 10
        6-bit comparator equal    : 8
        6-bit comparator not equal : 2
# Multiplexers                  : 5
```

```
        16-bit 4-to-1 multiplexer        : 2
        32-bit 12-to-1 multiplexer       : 1
        32-bit 3-to-1 multiplexer        : 2
# Logic shifters                 : 4
        32-bit shifter arithmetic left   : 1
        32-bit shifter arithmetic right  : 1
        32-bit shifter logical left      : 1
        32-bit shifter logical right     : 1
# Xors                           : 4
        1-bit xor2                       : 2
        32-bit xor2                      : 2
```

The synthesize result shows that the minimum period is 20.278ns for the clock signal so the maximum frequency is 49.315MHz.

7. CONCLUSIONS

This thesis presented a design for a RISC processor which can be implemented on an FPGA platform. The purpose of this study is to provide a basis for more complicated processor design and offers a test system for computer system organization education.

The processor instruction set consists of thirty instructions. These instructions perform basic arithmetic, logic, data transfer, branch and system functions. The designed processor has a five-stage pipeline architecture. These stages are named as instruction fetch, instruction decode, execution, memory access and write-back stages. The control unit is hardwired and placed in the decode section. The processor contains sixty four general purpose registers.

The processor eliminates pipeline data and structural hazards by using hardware techniques. Data hazards are eliminated by using forwarding method. Memory structural hazard is resolved by using separate instruction and data memory. Branch hazards are eliminated by flushing the pipeline registers.

The processor is modeled by using VHDL and functionality of the processor is tested by ModelSim simulation tool. Real life programs are executed to observe the correct execution of the processor. The processor is mapped on Xilinx's xc3s250e-4-vq100 FPGA which costs approximately 13.5$. The affordable cost of the design makes it a perfect choice for a test system.

The presented processor is one of the first RISC processor design implemented in Turkey. The planned future work will focus on enhancing the performance of the processor by adding instruction and data caches in the design.

REFERENCES

ABD-EL-BARR, M. and EL-REWINI, H., 2005. Fundamentals of Computer Organization and Architecture, John Wiley & Sons, Inc Publication, New Jersey.

ALPERT D. AND AVNON D., 1993. Architecture of the Pentium Microprocessor, IEEE Micro, 3:13, pp 11 - 21

ANONYMOUS, 2008a. Computer, http://en.wikipedia.org/wiki/Computer (1.10.2008)

ANONYMOUS, 2008b. Classic RISC pipeline, http://en.wikipedia.org/wiki/Classic_RISC_pipeline (1.11.2008)

BODUR, M., 2005. Computer Organization: An Introduction to RISC Hardware. 2nd Edition, Bileşim Yayınevi, İstanbul.

COLWELL, P.C., HITCHCOCK , C. III, JENSEN E.D., JENSEN, E.D., BRINKLEY SPRUNT, H.M. and KOLLAR, C.P.,1985. Instruction Sets and Beyond: Computers, Complexity, and Controversy. Computer 9:18 Pg:8-19

DANDAMUDI, S.P., 2003. Fundamentals of Computer Organization and Design, Springer, New York.

DANDAMUDI, S.P., 2004. Guide to RISC Processors. Springer, New York.

HENNESSY, J.L. and PATTERSON, D.A., 2003. Computer Architecture A Quantitative Approach. 3rd Edition, MORGAN Kaufmann Publishers, San Francisco.

MANO, M.M., 1993. Computer System Architecure. 3rd Edition, Prentice Hall, New Jersey.

MENTOR GRAPHICS, 2008. ModelSim - A Comprehensive Simulation and Debug Environment for Complex ASIC and FPGA designs, http://www.model.com/

NU HORIZONS ELECTRONICS Corp., 2008. Electronic Component Distrubutor, http://www.nuhorizons.com/

PARHAMI, B., 2005. Computer Architecture From Microprocessors to Supercomputers. Oxford University Press, New York.

PATTERSON D. and DITZEL R., 1980. The Case for the Reduced Instruction Set Computer. Computer Architecture News, 6:8, pp 25-33.

PATTESON, D.A. and HENNESSEY J.L., 2005. Computer Organization and Design, The Hardware / Software Interface. 3rd Edition Morgan Kaufmann Publishers, San Francisco.

PEDRONI, V.A., 2004. Circuit design with VHDL. MIT Press, London.

TORRES, G.,2006. Inside Pentium M Architecture, http://www.hardwaresecrets.com/article/270/4

XILINX, 2008. FPGA and CPLD Solutions from Xilinx, Inc., http://www.xilinx.com/

YEAGER, K.C.,1996. The MIPS R10000 Superscalar Microprocessor, IEEE Micro, 2:16, pp. 28-40.

BIOGRAPHY

Ali ŞENTÜRK was born in Afyonkarahisar, Turkey, in 1983. He has completed high school education in 2001 at Afyon Lisesi. He received the B.S. degree in Electrical and Electronics Engineering, Çukurova University in 2006. He started MSc program of the department of Computer Engineering, Çukurova University in 2006. He has been working as a Research Assistant at the department of Computer Engineering, Çukurova University since 2007. His interest areas are logic design, computer system architectures, computer arithmetic, web programming. He is a member of Turkish Chamber of Electrical Engineers.