Politecnico di Torino

III Facoltà di Ingegneria

DLX Project for the course Microelectronics Systems

Master degree in Electronic Engineering

Authors:

Yang Yu (s213850), Bryan D. Lara Tovar (s217156)

October 9, 2015

Summary

This report contains the functionality and details of the implementation of a 5-stage-pipelined DLX processor presented as our final project for the course Microelectronics System.

This implementation contains many features which include:

- A modified Booth's algorithm-based multiplicaton unit for supporting signed and unsigned multiplication.
- A modified Non-Restoring algorithm-based division unit for supporting signed and unsigned division.
- Square Root operation support performed by the modified division unit.
- Branch prediction feature based on 2-bit saturating counter.
- Hazard handling by forwarding techniques.

The synthesis result shows that the final critical path delay of our processor is sufficient to $T_{ck} = 2.5$. The calculated area after optimizing the delay of critical path was around 31421.

Contents

CHAPTER 1

Introduction

Our DLX processor is a 5-stage pipelined in-order microprocessor which fully supports general integer instructions and some long latency special integer instructions, like signed and unsigned multiplication and division and unsigned square root.

With maximum forwarding technology, the processor only suffers 1 type of logic hazard; Read After Load (RAL).

In order to eliminate or reduce the delay caused by branching, the Branch unit is organized in the 2nd stage (ID), which allows it to make the decision as soon as possible. Furthermore, a 2-bit-saturating-counter-based Branch Prediction unit is embedded in the Branch unit and is used to predict the result of branch in case of a Branch After Load (BAL) hazard.

CHAPTER 2

Functional Schema

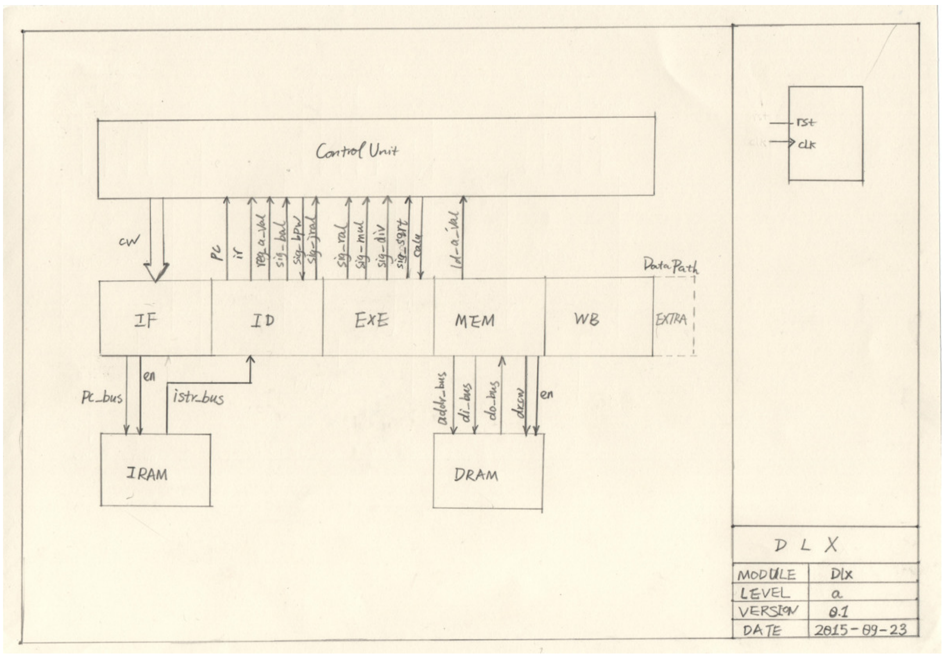

Figure 2.1: DLX overall structure

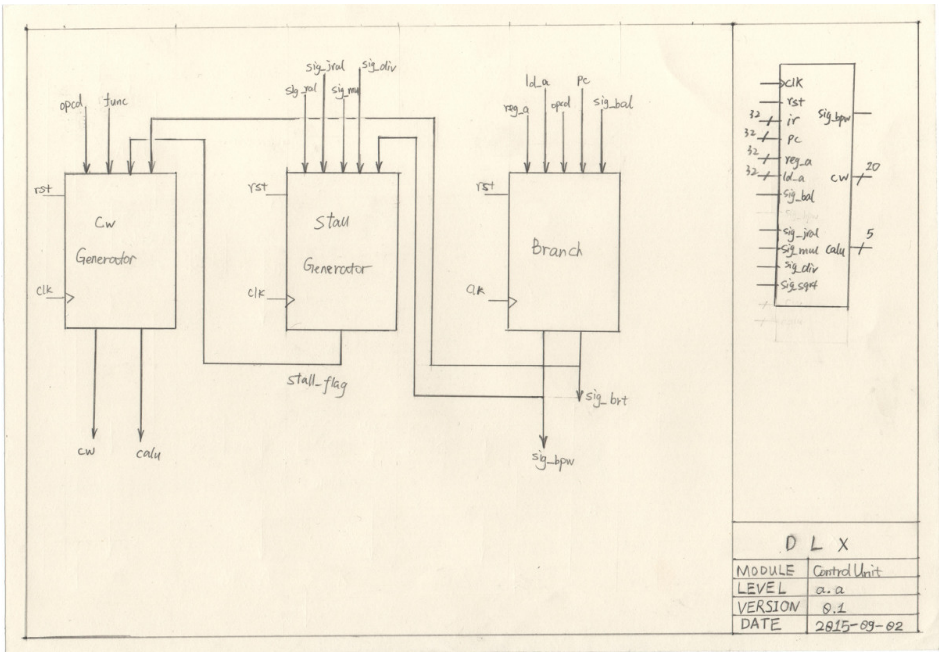

Figure 2.2: Control Unit

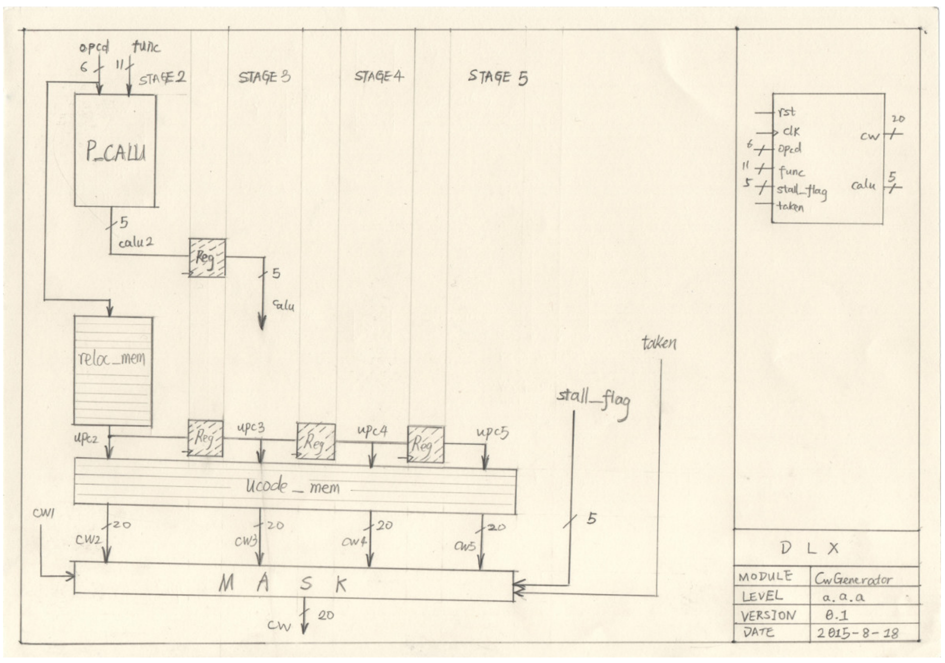

Figure 2.3: Control Word Generator

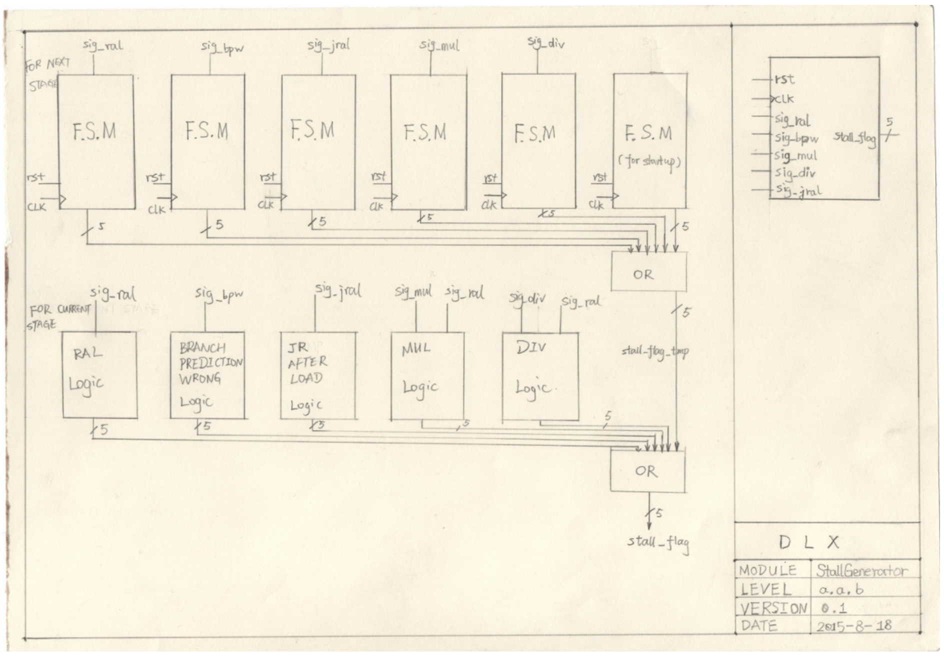

Figure 2.4: Stall Generator

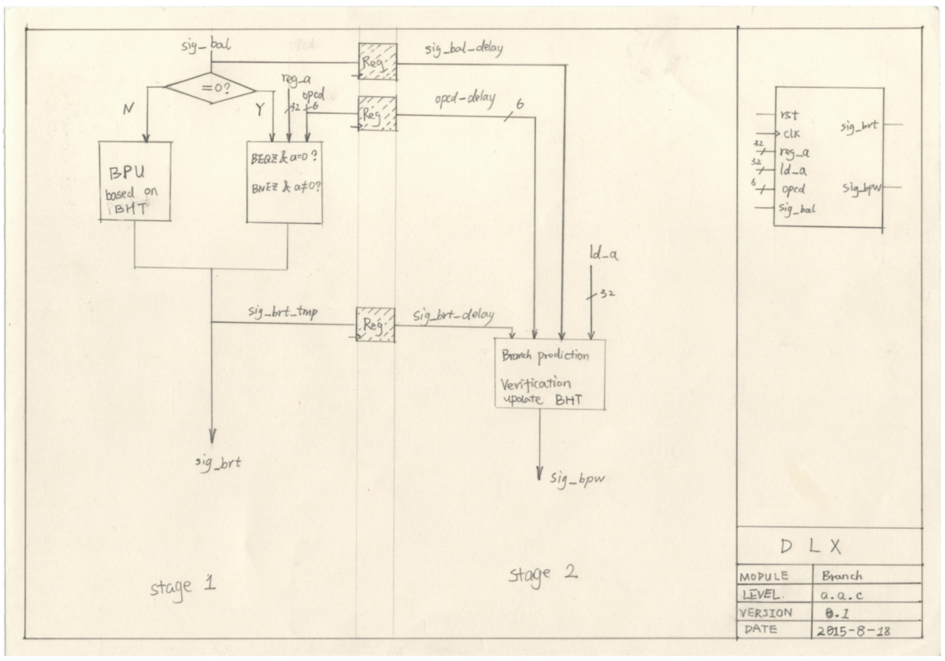

Figure 2.5: Branch Unit

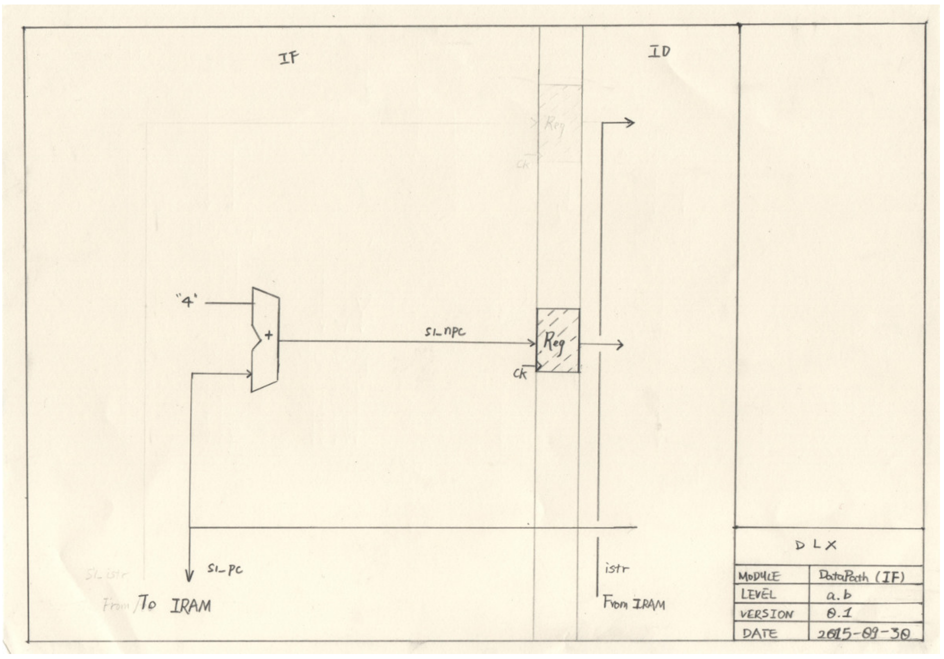

Figure 2.6: Datapath (IF)

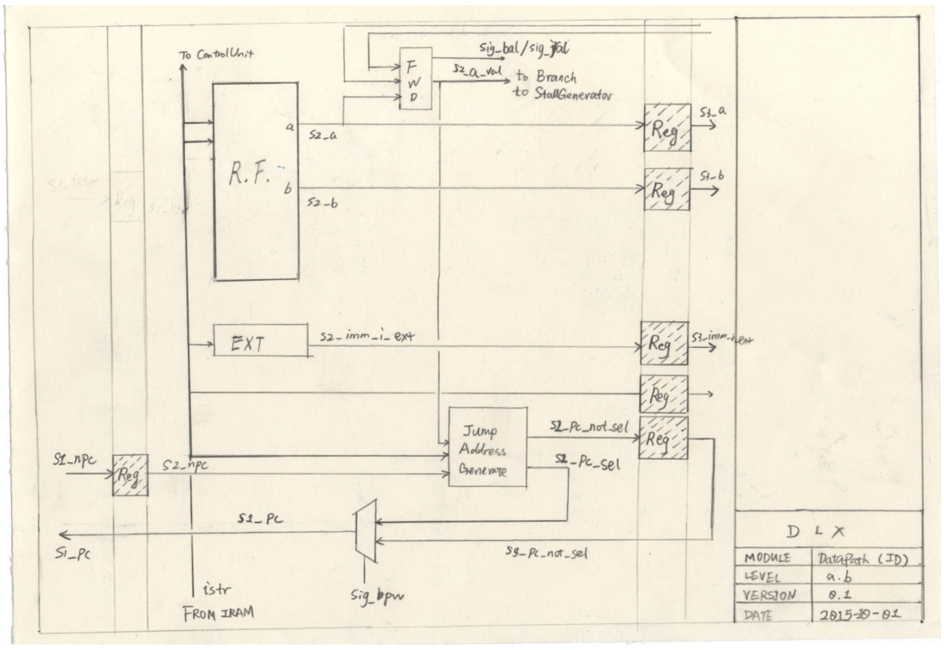

Figure 2.7: Datapath (ID)

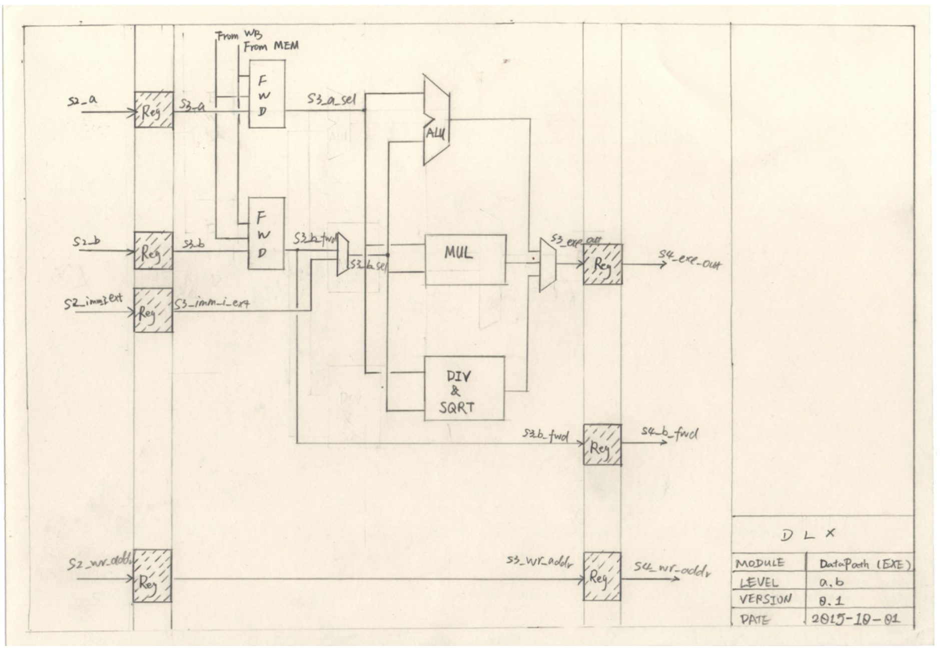

Figure 2.8: Datapath (EXE)

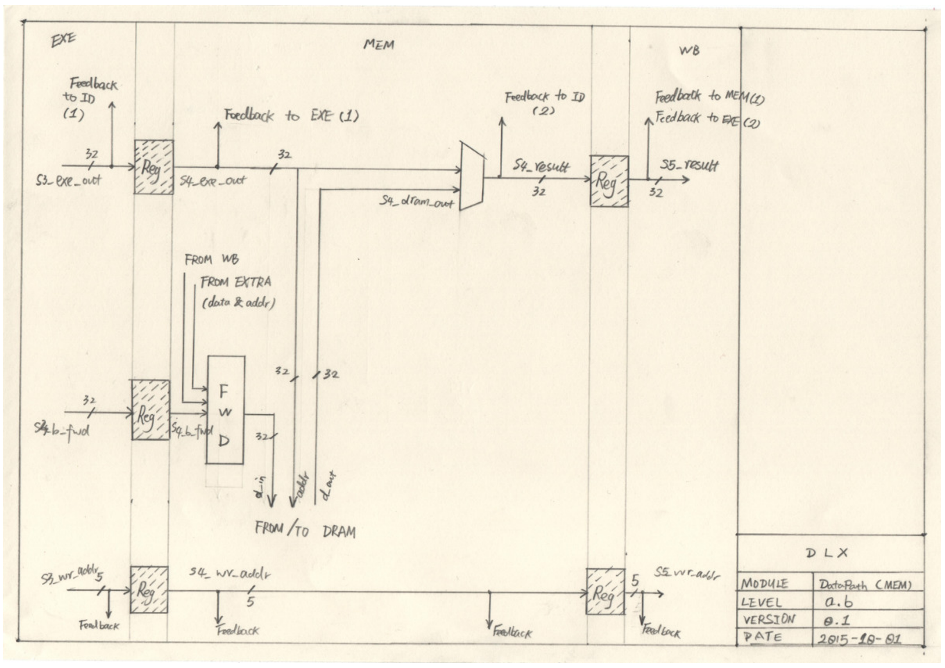

Figure 2.9: Datapath (MEM)

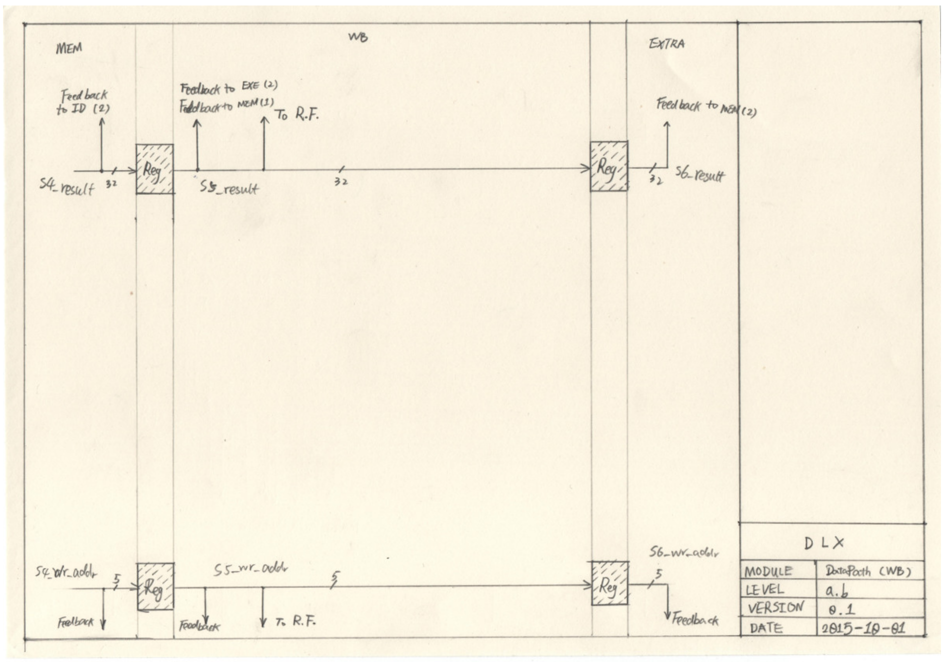

Figure 2.10: Datapath (WB)

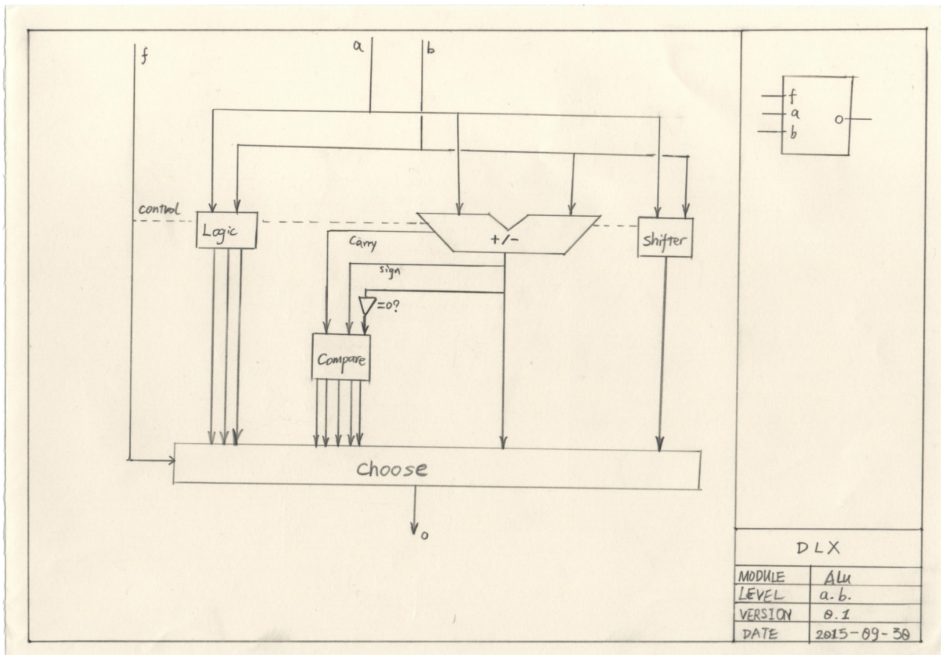

Figure 2.11: ALU

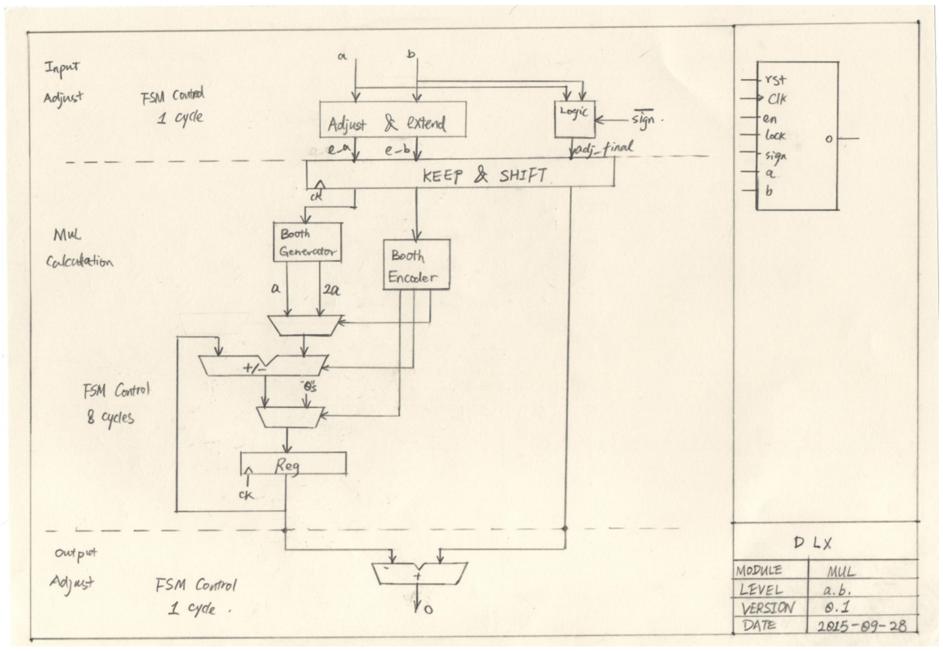

Figure 2.12: Multiplier

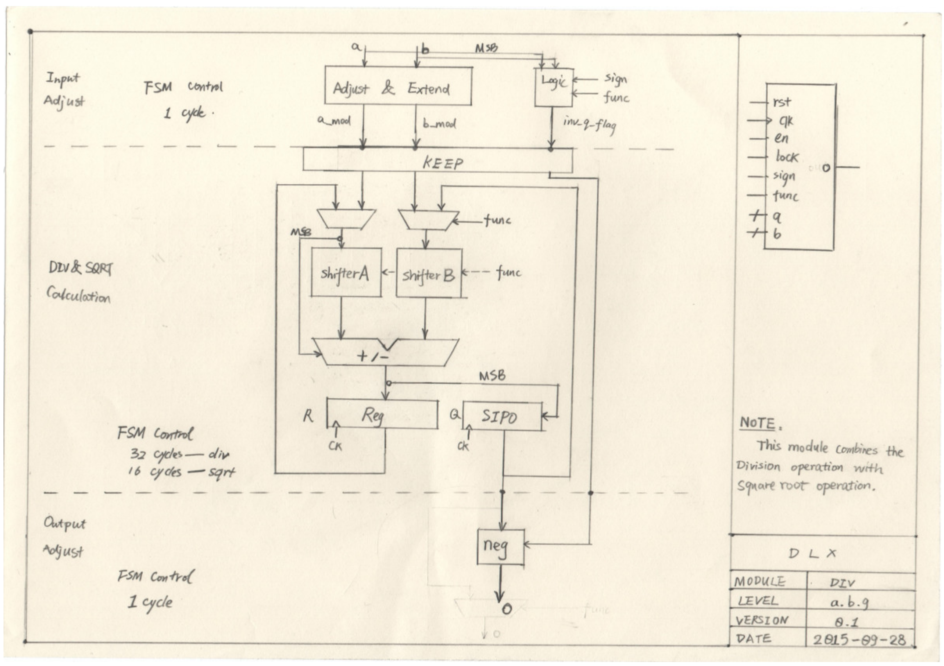

Figure 2.13: Divider and SQRT unit

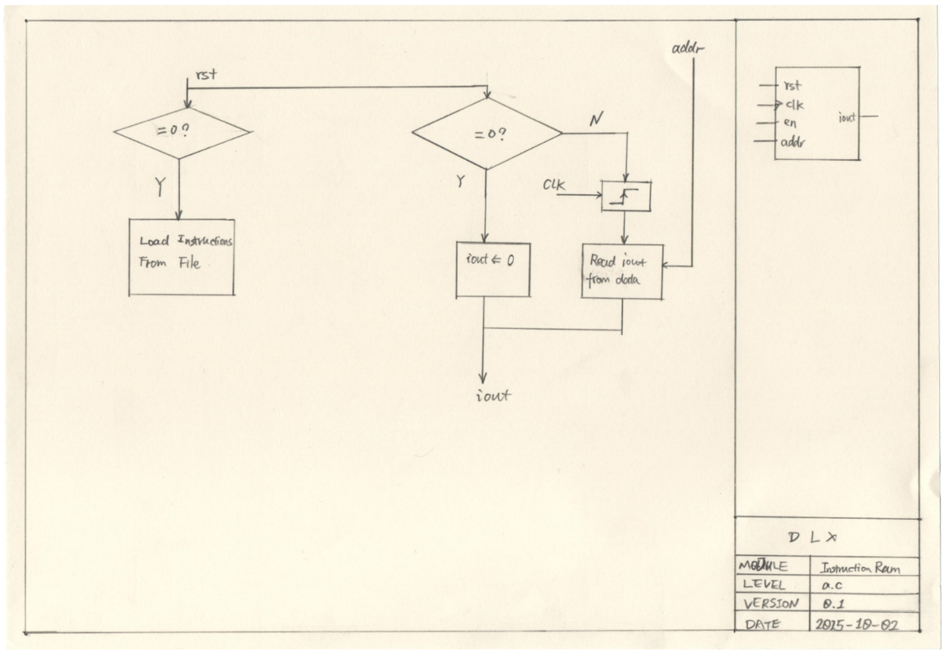

Figure 2.14: Instruction RAM

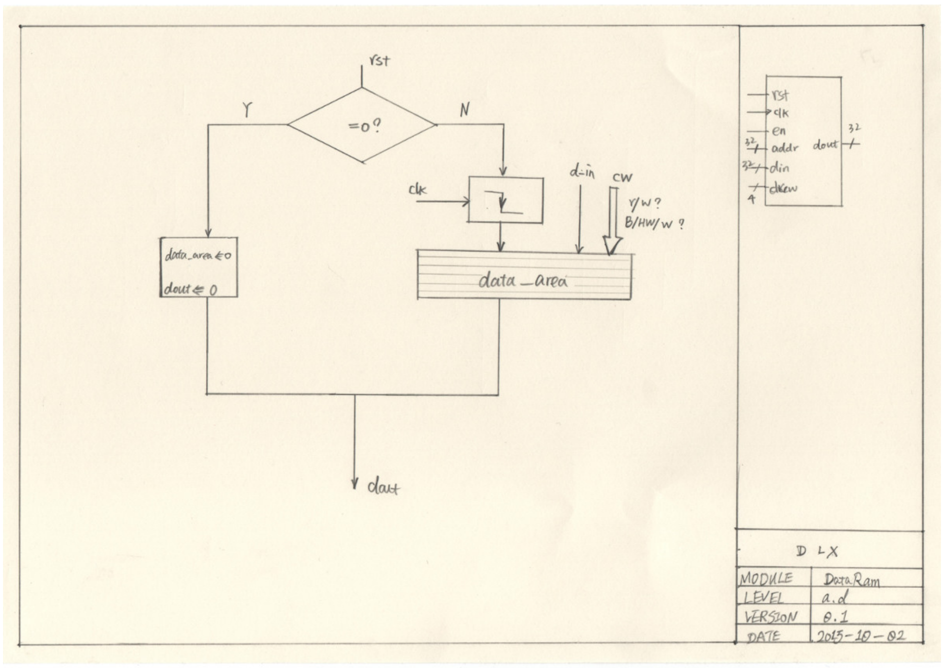

Figure 2.15: Data RAM

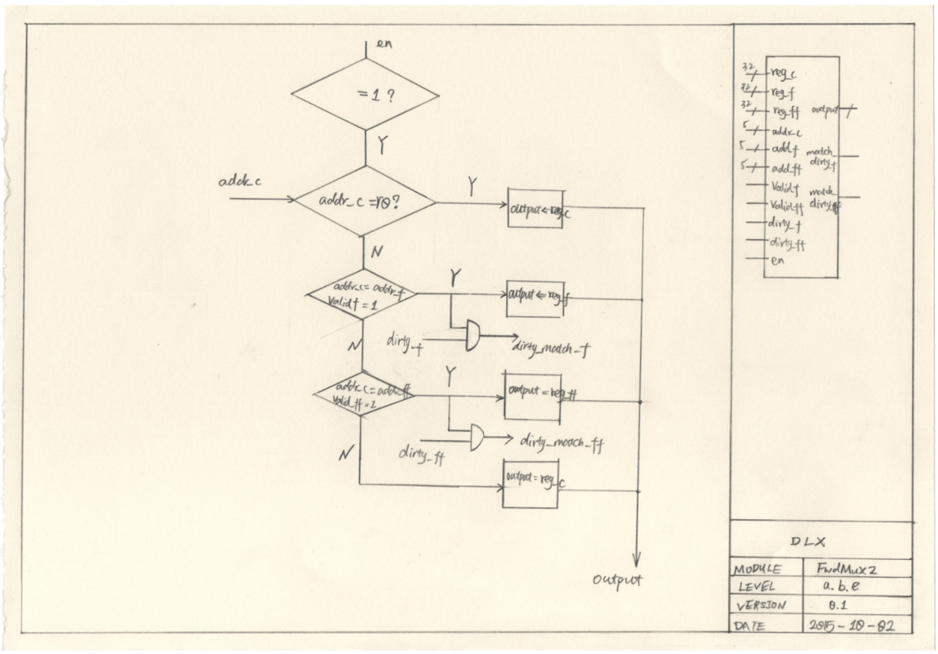

Figure 2.16: Forwarding Unit

CHAPTER 3

Implementation

The whole DLX microprocessor is organized in 4 parts, which are Control unit, Datapath, Instruction RAM and Data RAM. As shown in figure 2.1

3.1 Control Unit

Control Unit mainly produces the control words to control the behavior of our microprocessor's Datapath. It also needs to decide how a stall signal affects the pipeline and whether a branch is taken or not.

Our microprocessor's Control Unit consists of a Control word generator, a Stall generator and a Branch unit. As shown in figure 2.2

3.1.1 Control Word Generator

The Control word generator (CwGenerator) is a tiny-microprocessor-based module with 2 memory levels; a relocated memory and a microcode memory.

It first looks up a signal called *upc2* corresponding to the current instruction in the *stage 2 (ID)* and then gets the control words from the microcode memory based on this *upc2* signal. *Upc2* can be delayed with offset in each clock cycle, which makes it *upc3*, *upc4* and *upc5*.

For R-type and F-type instructions, all the control words are the same except for the part controlling the execution units (ALU, MUL, DIV). So a single control word that consists of many bits is introduced to solve this sharing problem among R-type and F-type instructions.

At the end of process, depending on *branch flag* and *stall flag*, the control words will be masked in a specific way to form the final control words.

The structure of Control word generator is shown in figure 2.3

3.1.2 Stall Generator

The Stall Generator deals with different signals which will lead to stall any stage of the pipeline. The signals include:

RAL Read After Load, 3rd stage (EXE).

BPW Branch Predict Wrong, 3rd stage (EXE).

JRAL JR After Load, 2nd stage (ID).

MUL Multiplication, 3rd stage (EXE).

DIV Division, 3rd stage (EXE).

SQRT Square Root, 3rd stage (EXE).

The detail of hazard handling will be covered in Section 3.4.

For each kind of hazard, we use a F.S.M. and a logic block to determine the behavior of Datapath, the former is for following clock cycles and the latter is for the current clock cycle. Finally, the *stall_flag* signal is produced by performing OR operation between the resulting signals obtained from these blocks.

The structure of Stall generator is shown in figure 2.4

3.1.3 Branch

The Branch Unit calculates the result of branch instructions by checking the value of the involving register. It will make predictions when the value of this register is not ready.

The detail of Branch After Load (BAL) hazard will be covered in section 3.4. And the detail of Branch Prediction will be covered in section 3.5

The structure of Branch unit is shown in figure 2.5

3.2 Datapath

The Datapath contains the basic 5 stages and one additional stage (EXTRA); it connects to Instruction RAM in stage 1 and 2; data RAM in stage 4.

3.2.1 IF (Stage 1)

Instruction Fetch sends PC to Instruction RAM and calculates NPC. As shown in figure 2.6.

3.2.2 ID (Stage 2)

Instruction Decode receives an instruction from Instruction RAM and depending on its type, the processor in this stage reads operands from Register File, extends the immediate value and calculates the PC for the next instruction.

The Register File works at the falling edge of the clock signal. It has 2 read ports and 1 write port. There are 32 4-byte integer registers inside the data area.

The Extend Unit works in both signed and unsigned modes.

The address which is not selected by the branch prediction unit is kept and delayed in case of a wrong prediction occurs, if so, this address could be restored.

The structure is shown in figure 2.7

3.2.3 EXE (Stage 3)

Execution stage receives operands and control words to perform different types of operations. For the ALU, all the operations are completed within 1 clock cycle. For MUL and DIV, the operations will take more than 1 clock cycle to be completed. The long latency operations will be covered in section 3.6

The structure is shown in figure 2.8

3.2.4 MEM (Stage 4)

Memory stage receives the resulting signals from the operations performed in the previous stage and based on the given instruction it will either store the resulting value in Data RAM or bring a value needed from Data RAM.

If there is no need of memory access, the output of execution unit will be directly send to the next stage.

The structure is shown in figure 2.9

3.2.5 WB (Stage 5)

WriteBack stage takes the final value given by the Memory stage and send it to the Register File (RF). An important point here is that the address in which this value is going to be set has been passed along from stage 2 (ID) to stage 5 (WB).

The structure is shown in figure 2.10

3.2.6 EXTRA (Stage 6)

Extra stage will basically take the same output of stage 5 and send it back to the memory stage in orded to handle hazards. this will be explained in details in the section 3.4

3.3 RAM

The Instruction RAM and Data RAM are implemented like register files. They perform either a read operation or a write operation within a single clock cycle without data miss.

The structure of IRAM is shown in figure 2.14 and the structure of DRAM is shown in figure 2.15.

3.4 Hazard Handling

3.4.1 Forwarding

Many kinds of Hazards can be eliminated by introducing forwarding. Our forwarding unit (Fwd2) is a 2-stage forwarding module with unwanted match indication.

The forwarding unit recieves the value of the current stage, the feedback value of the next stage and the feedback value of the next-next stage. It also receives enable signals to indicate the possiblity to perform the match operation with each feedback value. Two additional inputs *dirty_f* and *dirty_ff* are used to indicate the avalability of the feedback values. The output will be a value chosen from these 3 values based on their addresses and enable signals. If a match is labeled as ”dirty match”, the corresponding output signal will be ’1’ in order to notify other components that the output is not correct. The ”dirty match” will usually lead to a stall process.

The structure of Forwarding unit is shown in figure 2.16

3.4.2 Read After Load (RAL)

The Read After Load (RAL) hazard arises when the value of a register is needed by one of the two operands of execution units while the same register should be updated by a Load instruction which is 1 clock cycle in front. Since the value will not be available until the end of stage 4 (MEM), even with 2 stage forwarding, the problem still exists.

To solve the problem, we need to stall the pipeline. The sequence of stall flag is shown in table 3.1.

Table 3.1: Stall Flag for RAL

IF	ID	EXE	MEM	WB
1	1	1	0	0
0	0	0	1	0
0	0	0	0	1
0	0	0	0	0

Unfortunately, the solution is still not perfect. There is a situation when one operand (assuming A) is regularly matched with a double-forwarding feedback (from stage 5) while the other operand (assuming B) triggers a RAL stall by dirty matching the feedback of stage 4. After the 1 clock cycle stall, the value of B becomes correct, but the value of A becomes wrong since forwarding is not valid for A at the new clock cycle. Therefore, the value for A and B should be kept for 1 clock cycle. In case of a RAL stall happens, the correct kept value for the other operand can be restored if needed.

3.4.3 JR After Load (JRAL)

JR After Load (JRAL) hazard happens when a JR or JALR instruction follows a Load instruction which will update the register requested by JR/JALR instruction. Similar to RAL hazard, it will trigger a stall for 1 clock cycle, the stall flag sequence is shown in table 3.2.

Table 3.2: Stall Flag for JRAL

IF	ID	EXE	MEM	WB
1	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1
0	0	0	0	0

3.4.4 Branch After Load (BAL)

Similar to JRAL, Branch After Load hazard should request a pipeline stall too. However, the problem is solved by introducing the Branch Prediction Unit. The detail of Branch Prediction will be covered in section 3.5

3.4.5 Store After-After Write (SAAW)

A Store After-After Write hazard happens when a Store instruction is 2 stages later than an instruction writing to the same register that requested by the Store instructions. This will becomes a problem when there are only 5 stages. The Store instruction which is in the stage 4 cannot get the double-forwarding feedback since there is no stage 6. To solve the problem, we add a stage 6 (EXTRA) which simply delays the value of stage 5, and feedback to stage 4 in case of SAAW.

3.4.6 Long Latency Execution

Our DLX processor support some long latency execution like DIV, MUL and SQRT. These operation can not be completed within 1 clock cycle. While executing these instructions, the stage 3 will be occupied for a long time and the pipeline should be stalled. The stall flag sequence is shown in table 3.3.

Table 3.3: Stall Flag for Long Latency Execution

IF	ID	EXE	MEM	WB
1	1	1	0	0
1	1	1	1	0
1	1	1	1	1
....				
1	1	1	1	1
0	0	0	1	1
0	0	0	0	1
0	0	0	0	0

3.5 Branch Predition

The information for each branch prediction is stored inside a branch history table (BHT).

The BHT is organized like a direct-mapped cache. Each entry contains 2 fields, TAG and VALUE. TAG is determined by the first part of branch address (Default 25 bits) and VALUE is the current 2-bit state of the branch instruction. The index of each entry is determined by the middle part of address (Default 5 bits).

Table 3.4: Address of branch instruction

TAG (25 bits)	INDEX (5 bits)	Always 0 for any instruction (2 bits)

Table 3.5: BHT structure

....	
TAG (25 bits)	VALUE (2 bits)
....	

Table 3.6: VALUE Meaning

00	Strongly not taken
01	Weakly not taken
10	Weakly taken
11	Strongly taken

The branch prediction can be divided into 2 phases, prediction phase and verification phase.

During the prediction phase (ID stage), it looks up the BHT by index, compares the TAG and get the VALUE. Based on VALUE, it predicts whether the branch will be taken or not. If the TAG does not match, the entry will be replaced with the new TAG with VALUE="00".

During the verification phase (EXE stage), it checks whether the previous prediction is correct or not and refreshes the VALUE area according the rule shown in figure 3.1

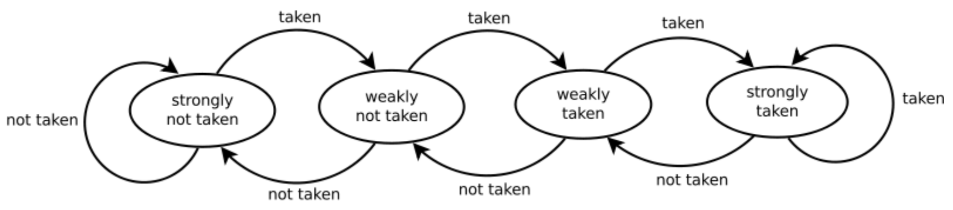

Figure 3.1: BHT VALUE update rule

3.6 Long Latency Instructions

3.6.1 Multiplication

The structure of Multiplier is shown in figure 2.12.

MULT

Signed multiplication is performed with booth's algorithm, it is completed within 8 clock cycles.

By properly encoding the output of Booth Encoder and introducing of Adder/Substractor, we reduced the complexity of multiplexer and removed the substractors needed by calculating -a and -2a. The encoding rules are shown in table 3.7.

Table 3.7: Booth Encoder Output

BIT2	ZERO bit	"0" if we need to choose "0", otherwise "1"
BIT1	SIGN bit	"0" if we need to choose +a or +2a, otherwise "1"
BIT0	DOUBLE bit	"0" if we need to choose +a or -a, otherwise "1"

MULTU

Unsigned multiplication is a standard unsigned multiplication with some modification on inputs and output.

Let's take a 4-bit unsigned multiplication as an example:

$$\begin{aligned}
&[A_3|A_2|A_1|A_0] \times [B_3|B_2|B_1|B_0]\\
=&[A_3B_3|A_3B_2 + A_2B_3|A_3B_1 + A_2B_2 + A_1B_3|A_3B_0 + A_2B_1 + A_1B_2 + A_0B_3|\\
&A_2B_0 + A_1B_1 + A_0B_2|A_1B_0 + A_0B_1|A_0B_0]
\end{aligned}$$

The result can be further divided into several parts.

$$\begin{aligned}
&[A_3|A_2|A_1|A_0] \times [B_3|B_2|B_1|B_0]\\
=&[A_3B_3|A_3B_2 + A_2B_3|A_3B_1 + A_2B_2 + A_1B_3|A_3B_0 + A_2B_1 + A_1B_2 + A_0B_3|\\
&A_2B_0 + A_1B_1 + A_0B_2|A_1B_0 + A_0B_1|A_0B_0]\\
=&[A_2B_2|A_1B_2 + A_1B_2|A_2B_0 + A_1B_1 + A_0B_2|A_1B_0 + A_0B_1|A_0B_0]+\\
&([A_3|A_2|A_1|A_0]B_3 + [B_3|B_2|B_1|B_0]A_3 - [A_3B_3|0|0|0]) << 3\\
=&[0|A_2|A_1|A_0] \times [0|B_2|B_1|B_0]+\\
&([A_3|A_2|A_1|A_0]B_3 + [B_3|B_2|B_1|B_0]A_3 - [A_3B_3|0|0|0]) << 3
\end{aligned}$$

With this equation, we can transfer a 4-bit unsigned multiplication to 4-bit signed multiplication of 2 positive operands. The result should be adjusted by adding $([A_3|A_2|A_1|A_0]B_3 + [B_3|B_2|B_1|B_0]A_3 - [A_3B_3|0|0|0]) << 3$.

In summary, an unsigned multiplication can be converted to a signed multiplication by performing some modifications on inputs and output. So the total latency for a multiplication will be 10 clock cycles, which include 1 clock cycle for input adjustment, 8 clock cycles for signed multiplication and 1 clock cycle for output adjustment.

3.6.2 Division

The structure of Divider is shown in figure 2.13.

DIVU

The core part which performs unsigned division is based on the Non-Restoring Division algorithm, completed within 32 clock cycles.

The control part of the divider is a F.S.M.

DIV

The signed division is based on the unsigned division with some adjustments on the inputs and output.

If we define that **the sign of the remainder should be the same as the dividend**, we can conclude that **the absolute value of quotient is the same for both signed and unsigned division**. Therefore, we can get the result of signed division by only adjusting the sign of the quotient calculated by unsigned division. Performing this adjustment on output depends on the signs of both dividend and divisor. The rules are shown in table 3.8.

Table 3.8: Sign adjust rules

Dividend	Divisor	Output Adjust
+	+	No
+	-	Yes
-	+	Yes
-	-	No

In summary, a signed division can be converted to an unsigned multiplication by performing some modifications on inputs and output. So the total latency for a division will be 34 clock cycles, which include 1 clock cycle for input adjustment, 32 clock cycles for unsigned division and 1 clock cycle for output adjustment.

3.6.3 Square Root

Only unsigned square root exists in real domain.

A Restoring Square root algorithm is based on equation $(a+b)^2 = a^2 + 2ab + b^2$. For base-2, if we assume $R = (a_1 + a_2 + ... + a_m + ... + a_n)$, with a_i equals either 1 or 0. We can get:

$$R = a_1a_1 + [2a_1 + a_2]a_2 + [2(a_1 + a_2) + a_3]a_3 + ... + [2(\Sigma_{i=1}^{m-1} a_i) + a_m]a_m + ... + [2(\Sigma_{i=1}^{m-1} a_i) + a_n]a_n$$

When we try to get the m-th bit. We get:

$$\begin{aligned}R_m &= [2(\Sigma_{i=1}^{m-1} a_i) + a_m]a_m + ... + [2(\Sigma_{i=1}^{m-1} a_i) + a_n]a_n \\ &= [2Q + a_m]a_m + R_{m+1} \\ R_{m+1} &= R_m - [2Q + a_m]a_m\end{aligned}$$

Q is the result by setting m+1 to n bits to zero. In each iteration, Q can be obtained by left shifting 1 bit of current Q result.

Now we guess $a_m = 1$ and set:

$$R_{m+1}^{es} = R_m - [2Q + 1]$$

If $R_{n-1}^{es} >= 0$, the guess is correct, otherwise, the guess is wrong; we need to restore $R_{n+1} = R_n$.

The Non-Restoring method is similar. Instead of restoring the remainder, we compensate the wrong guess in the next addition.

Therefore, we can use the same hardware resources of division with a little modification. The divisor will be replaced by quotient appended 01 or 11. And the remainder will be left shift 2 bits instead of 1.

An example is shown in table 3.9.

Table 3.9: SQRT example

	1	0	0	1	
)	01	01	11	01	Positive
-	01				Since positive, append 01 and perform SUB
	00	01			Positive, append Q with 1, Now Q=1
-	1	01			Since positive, append 01 and perform SUB
	11	00	11		Negative, append Q with 0, Now Q=10
+		10	11		Since negative, append 11 and perform ADD
	11	11	10	01	Negative, append Q with 0, Now Q=100
+		1	00	11	Since negative, append 11 and perform ADD
			11	00	Positive, append Q with 1, Now Q=1001

3.7 Synthesis and Optimization

For the synthesis part, we only optimized the delay with time constrain *clock* = 2.5. The critical path is in the stage 2 which is used to generate the jump address. This is because:

The relative jump address will ADD to the next pc counter, which will take time.

The register file works at the FALLING edge of clock.

The branch unit, especially the branch prediction unit, works after it gets the register value from the register file.

In case of a wrong prediction happens, the final jump address will be changed.

In summary, the synthesis result is listed in table 3.10.

Table 3.10: Synthesis Result

Timing(Arrive Time)	Before Optimization	3.27
Timing(Arrive Time)	After Optimization	2.03
Area	Before Optimization	30520.839844
Area	After Optimization	31421.783203

CHAPTER 4

Discussion and Conclusions

Our processor has been tested with all the provided assembly code files except for the ones inside the directory "advanced_asm_example". Some scripts cannot be executed correctly due to wrong instruction format (e.g. BNEZ r1, #-3).

We have modified the perl script of assembler to support SQRT instruction. The processor has been tested with SQRT instruction and it works correctly.

Our DLX processor is based on the pipelined datapath, which only allows 1 instruction in each stage. When the amount of long latency instructon increases, the processor will wast a lot of time on stall. If we want to solve this issue, it is nessasary to introduce instruction-level parallism. However, since we are mainly dealing with integer instructions, thinking of implementing superscalar structure or register renaming technology to our DLX project would be too complex.

We assume that there is no cache miss in both IRAM and DRAM, also that, every Load and Store operation completes in 1 clock cycle. But these situations will not happen in real life. Therefore, we need to improve our interface to RAM and deal with stall caused by cache miss. A possible new feature could deal with these situations by adding a real L1-cache between the processor and RAM.

We optimized the delay of our processor with clock constrain $T_{ck} = 2.5$. At the end we got a positive slack which means the clock frequency is fine.

CHAPTER 5

References

- T. Sutikno, ”An Optimized Square Root Algorithm for Implementation in FPGA Hardware”, pp.2-3, April 2010