Politecnico di Torino

III Facoltà di Ingegneria

DLX Project for the course Microelectronics Systems

Master degree in Computer Science Engineering

Authors: GROUP ?

Yang Yu (s213850), Bryn Lara (sxxxxxx)

October 8, 2015

Contents

CHAPTER 1

Overview and Features

Our DLX processor is a 5-stage pipelined in-order microprocessor which fully support general integer instructions and some long latency special integer instructions, like Multiplication, Division (for both Signed and Unsigned) and Unsigned Square Root.

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

In order to eliminate or reduce the delay caused by branch, the branch unit is organized in the 2nd stage (ID) which allows it to make the desition as soon as possible. Further more, a 2-bit saturating counter based Branch Prediction Unit is embeded in the branch unit to predict the result of branch in case of Branch After Load (BAL) stall.

CHAPTER 2

Structure

The whole DLX microprocessor is organized in 4 parts: the control unit, the datapath, the instruction RAM and the data RAM. As shown in figure 4.1

2.1 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 4.2

2.1.1 Control Word Generator

The Control word generator (CwGenerator) is a tiny micro-processor 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. The *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 4.3

2.1.2 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 2.4.

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 4.4

2.1.3 Branch

The Branch Unit calculates the result of branch instructions by checking the value of the invoving 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 2.4. And the detail of Branch Prediction will be covered in section 2.5

The structure of Branch unit is shown in figure 4.5

2.2 Datapath

2.2.1 IF

2.2.2 ID

2.2.3 EXE

2.2.4 MEM

2.2.5 WB

2.2.6 EXTRA

2.3 RAM

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

The structure of IRAM is shown in figure 4.14 and the structure of DRAM is shown in figure 4.15.

2.4 Hazard Handling

2.4.1 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 choosen from these 3 values based on their addresses and enable signals. If a match is labeled with ”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 4.16

2.4.2 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 talbe 2.1.

Table 2.1: 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 satuation 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 shoud be kept for 1 clock cycle. In case of RAL stall happens, the kept correct value for the other operand can be restored if needed.

2.4.3 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 2.2.

Table 2.2: 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

2.4.4 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 2.5

2.4.5 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.

2.4.6 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 2.3.

Table 2.3: 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

2.5 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 2.4: Address of branch instruction

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

Table 2.5: BHT structure

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

The branch prediction can be devided into 2 phases: prediction phase and verification phase.

During the prediction phase (ID stage), it look up the BHT by index, compare the TAG and get the VALUE. Based on VALUE, it predict whether the branch will take or not. If the TAG doesn't match, the entry will be replaced with the new TAG with VALUE="00".

During the verification phase (EXE stage), it check whether the previous prediction is correct or not. and update the VALUE area according the rule shown in figure 2.1

Table 2.6: VALUE Meaning

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

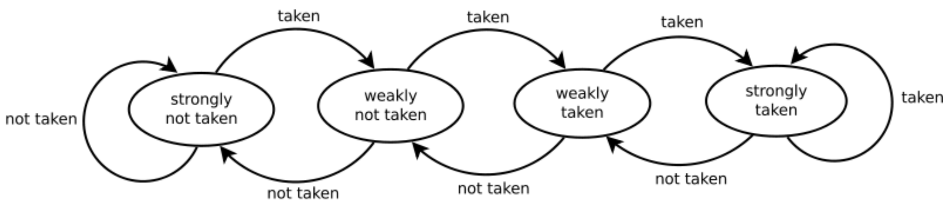

Figure 2.1: BHT VALUE update rule

2.6 Long Latency Instructions

2.6.1 Multiplication

The structure of Multiplier is shown in figure 4.12.

MULT

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

With 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 2.7.

Table 2.7: 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 in 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 convert to a signed multiplication by performing some modification 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.

2.6.2 Division

The structure of Divider is shown in figure 4.13.

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 properly adjust the inputs and output.

If we define that **the sign of the reminder 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 adjust the sign of the quotient calculated by unsigned division. Whether to perform the adjustment of output depends on the signs of both dividend and divisor. The rules are shown in table 2.8.

Table 2.8: Sign adjust rules

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

In summary, a signed division can be convert to an unsigned multiplication by performing some modification 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.

2.6.3 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 binary digits, if we assuming $SQRT(X) = [ab]$, with b equals either 1 or 0. We can get:

$$\begin{aligned}
X &>= (a \cdot 2^1 + b)^2 \\
&= (a \cdot 2^1)^2 + 2 \cdot a \cdot 2^1 \cdot b + b^2 \\
&= 4a + 4(ab) + b^2 \\
So, & \\
R_n &= (X - 4a) \\
&>= 4(ab) + b^2 \\
We\ set: & \\
R^{es}_{n+1} &= R_n - [4(ab) + b^2]
\end{aligned}$$

Therefore, let's guess b=1:

$$R^{es}_{n+1} = R_n - [4a + 1]$$

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

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

Therefore, we can use the same hardware resources of division with a little modification. The reminder will be appended 01 or 11 instead of left shift 1 bit in each clock cycle and the position of divisor will be replaced by shifted intermediate quotient.

An example is shown in table 2.9.

Table 2.9: 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

CHAPTER 3

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 wrong prediction happens, the final jump address will be changed.

In summary, the synthesis result is listed in table 3.1.

Table 3.1: 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

Appendix

4.1 Diagram

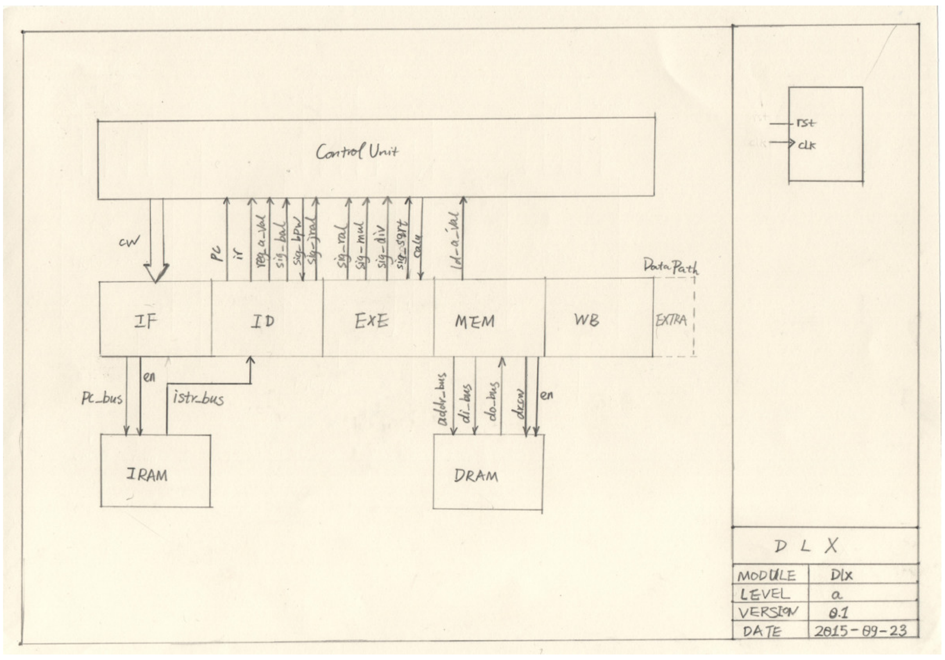

Figure 4.1: DLX overall structure

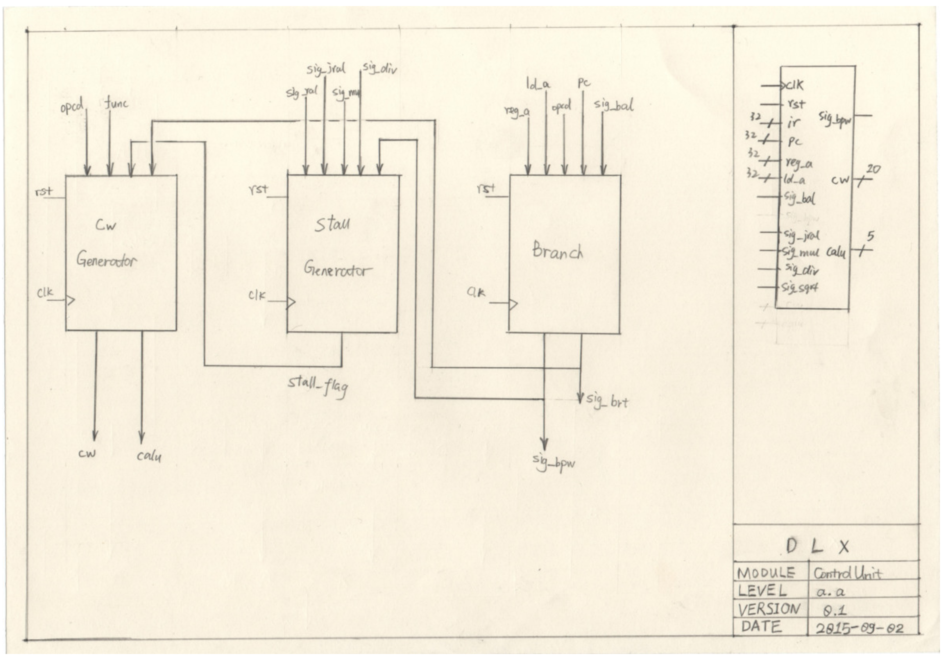

Figure 4.2: Control Unit

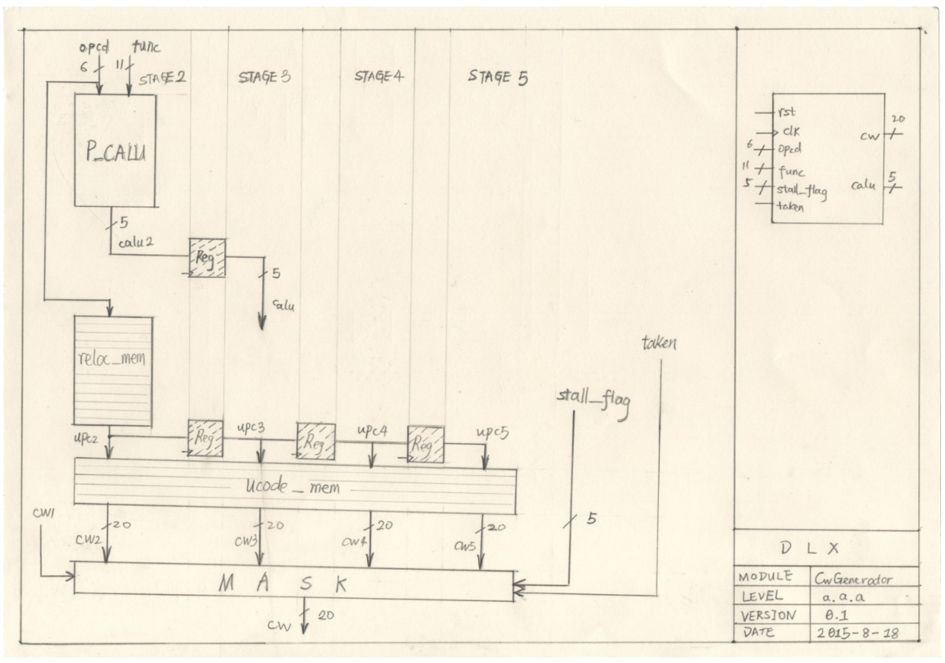

Figure 4.3: Control Word Generator

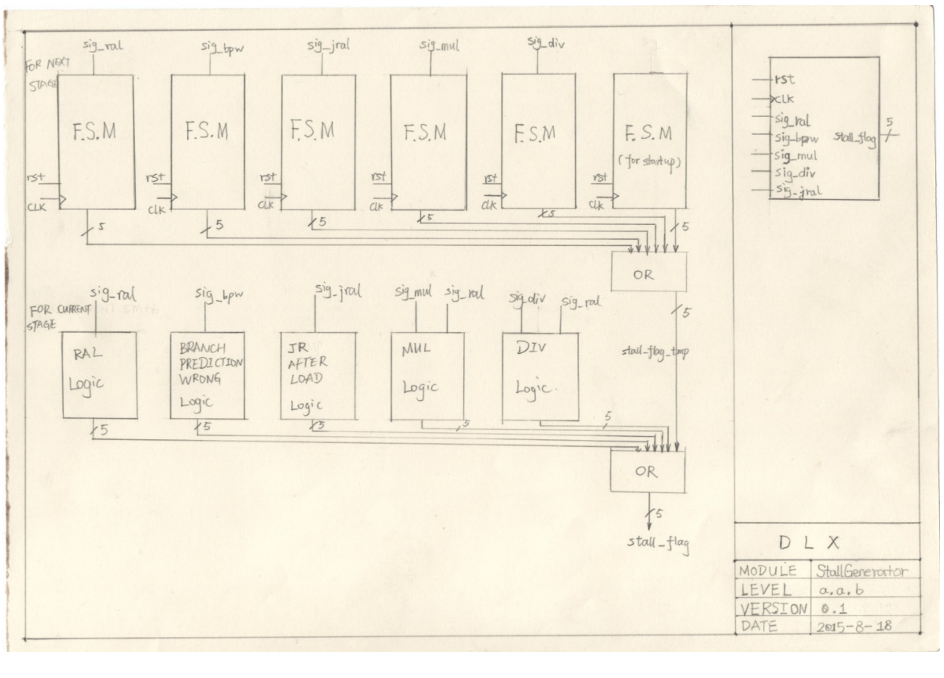

Figure 4.4: Stall Generator

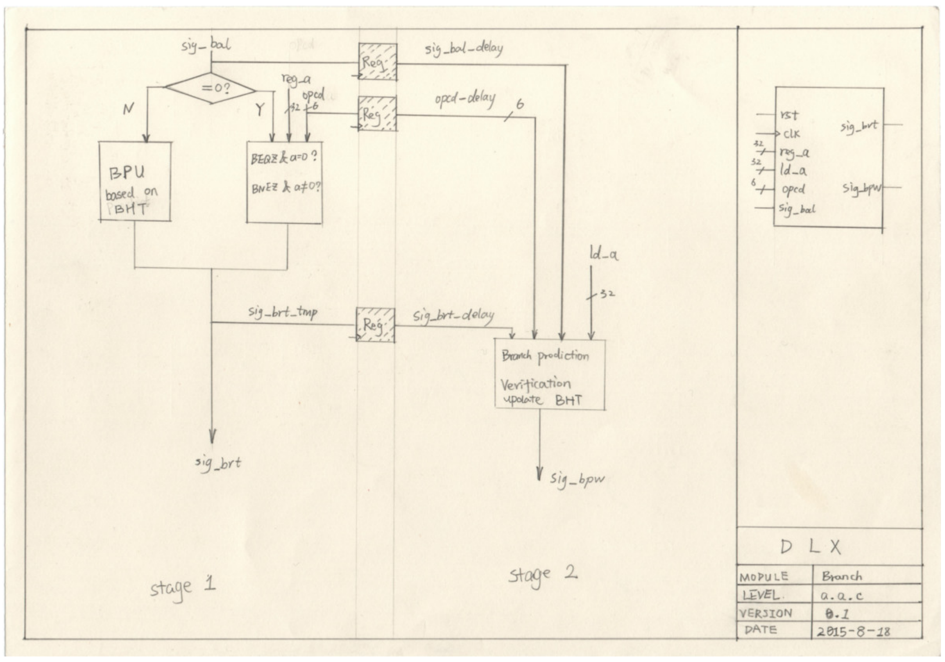

Figure 4.5: Branch Unit

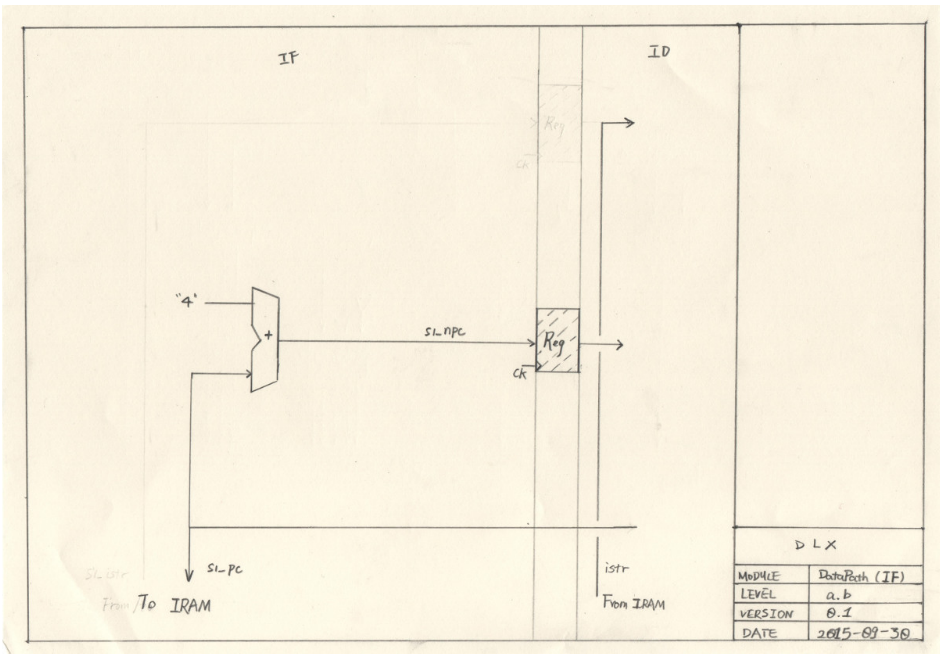

Figure 4.6: Datapath (IF)

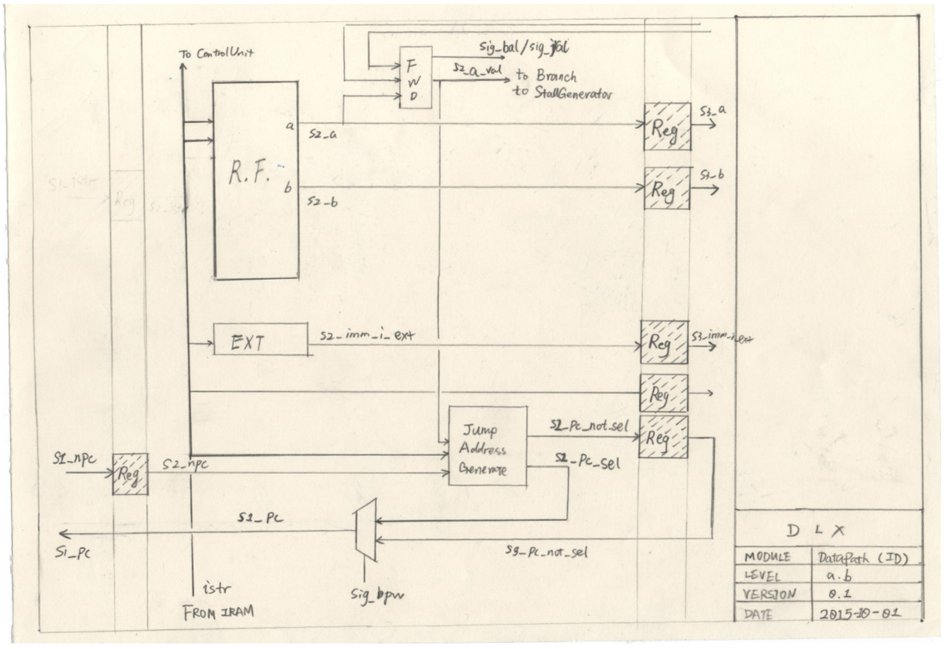

Figure 4.7: Datapath (ID)

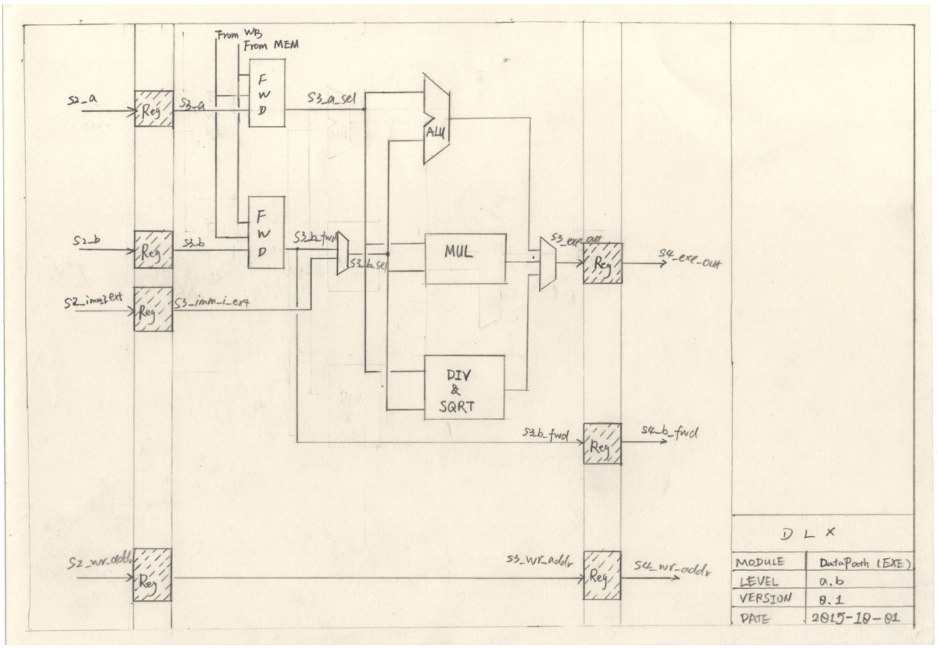

Figure 4.8: Datapath (EXE)

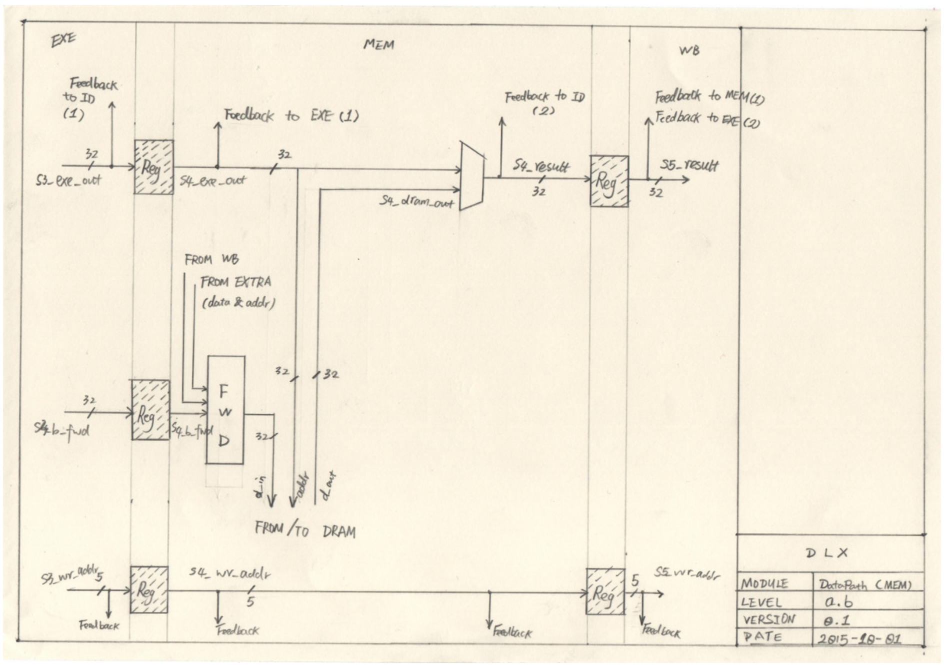

Figure 4.9: Datapath (MEM)

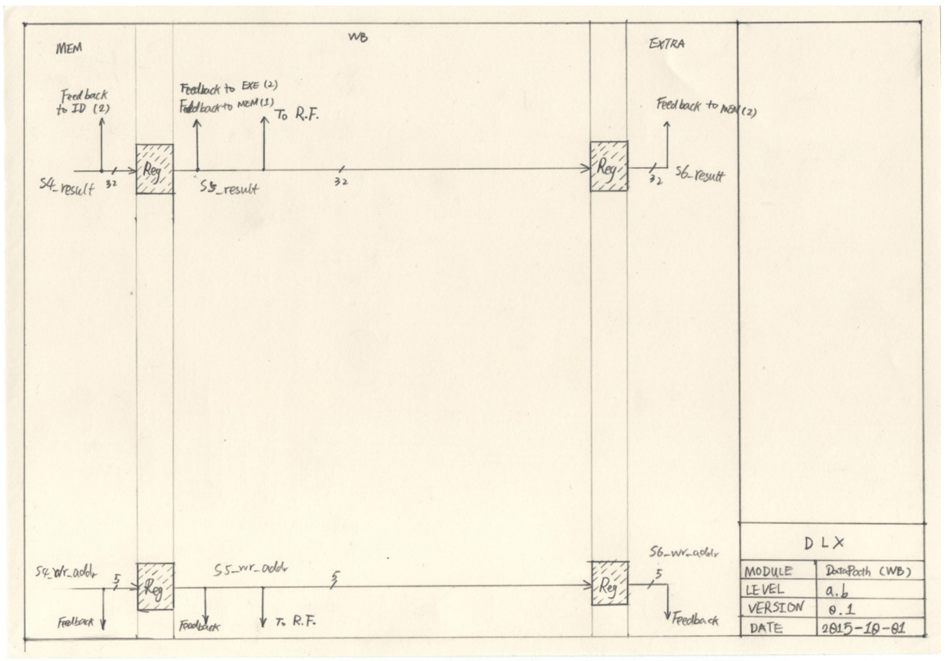

Figure 4.10: Datapath (WB)

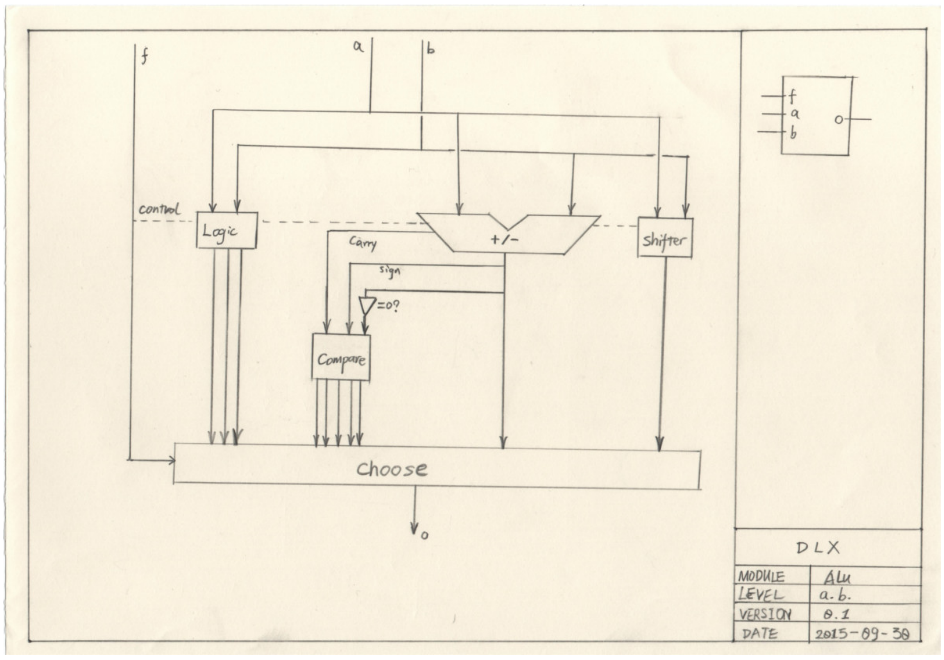

Figure 4.11: ALU

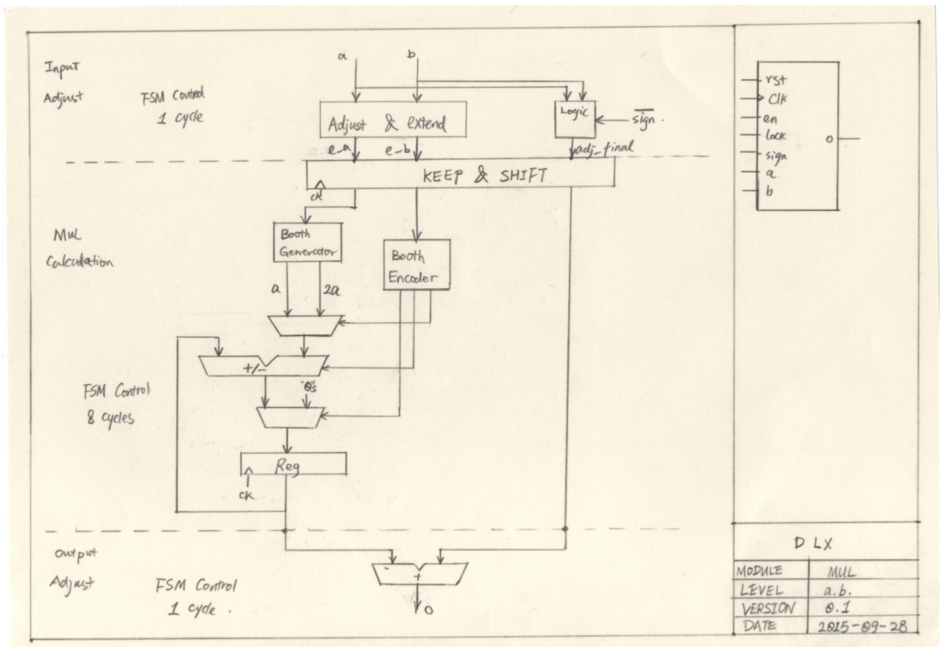

Figure 4.12: Multiplier

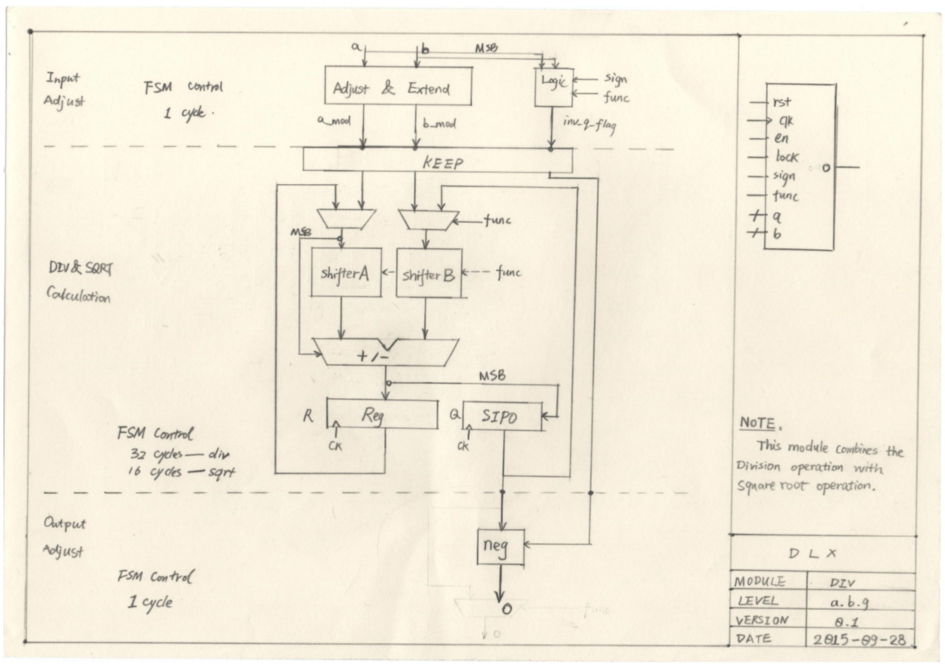

Figure 4.13: Divider and SQRT unit

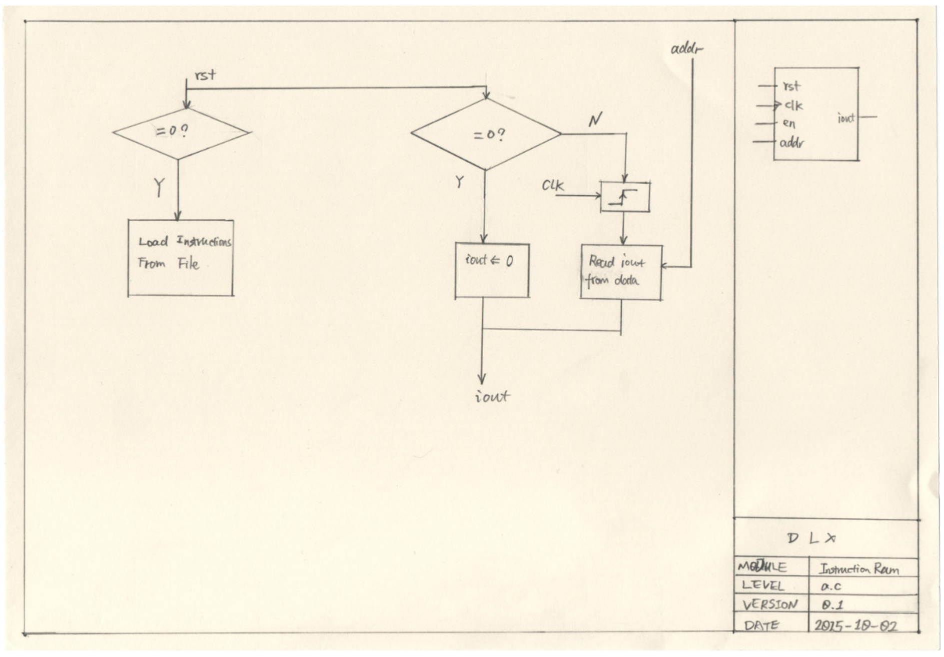

Figure 4.14: Instruction RAM

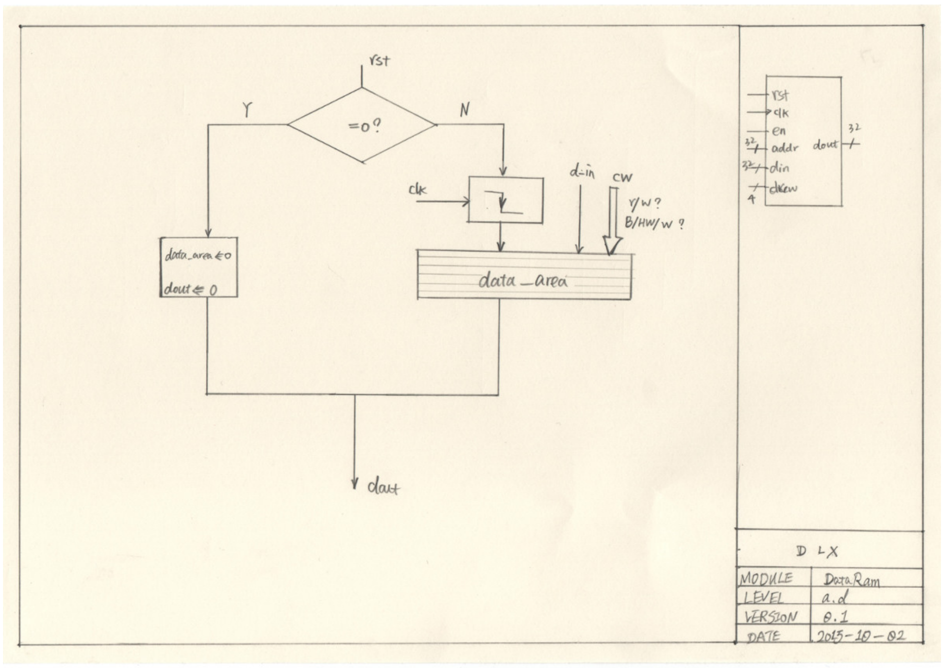

Figure 4.15: Data RAM

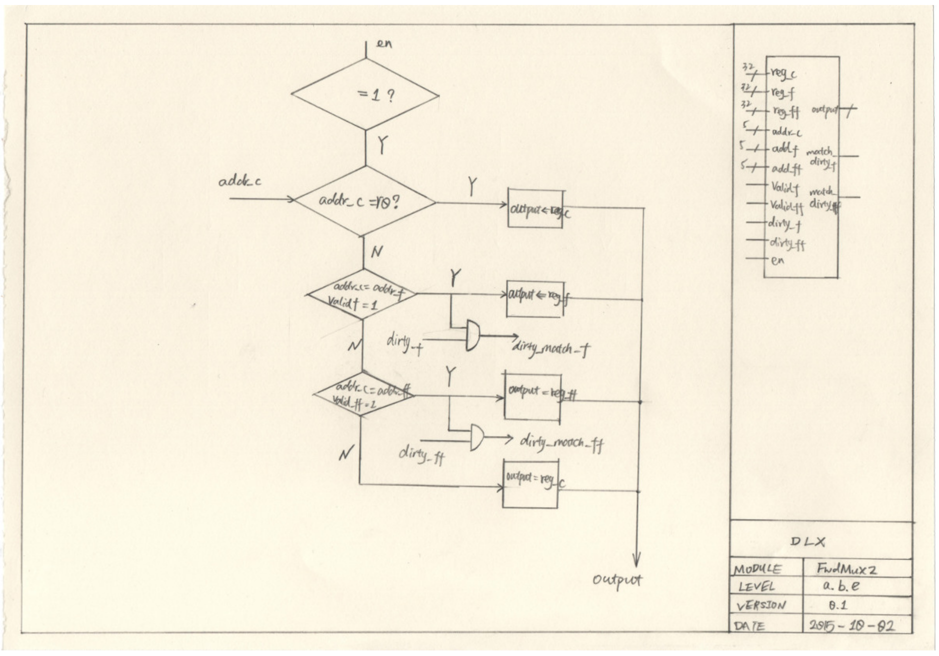

Figure 4.16: Forwarding Unit