CA final report -- Pipeline MIPS

B02901011 趙祐毅
B02901120 羅志軒
B03901026 許凱傑
2017.6.21

Table of Contents

Introduction **2**
- **A. Target** **2**
- **B. Division of work** **2**

Baseline **3**
- **A. Cache** **3**
  1. Design Architecture 3
- **B. MIPS** **3**
  1. Design Architecture 3
  2. Special Design 6
  3. Critical path 7
- **C. Synthesis Result and Analysis** **8**

Extension **9**
- **A. Branch Prediction** **9**
  1. Target 9
  2. Design Architecture 9
  3. Synthesis Result and Analysis 9
- **B. Multiplication / Division** **12**
  1. Target 12
  2. Iterative approach 12
  3. Simulation 13
  4. Synthesis 14
- **C. Two-Level Cache** **15**
  1. Description 15
  2. Experiment 15

References **16**

Introduction

A. Target

Design a pipelined MIPS processor with instruction cache and data cache, and support the following instructions..

Table 1. Required Instruction Set

Name	Description
ADD	Addition, overflow detection for signed operand is not required*
ADDI	Addition immediate with sign-extension, without overflow detection*
SUB	Subtract, overflow detection for signed operand is not required*
AND	Boolean logic operation
ANDI	Boolean logic operation, zero-extension for upper 16bit of immediate
OR	Boolean logic operation
ORI	Boolean logic operation, zero-extension for upper 16bit of immediate
XOR	Boolean logic operation
XORI	Boolean logic operation, zero-extension for upper 16bit of immediate
NOR	Boolean logic operation
SLL	Shift left logical (zero padding)
SRA	Shift right arithmetic (sign-digit padding)
SRL	Shift right logical (zero padding)
SLT	Set less than, comparison instruction
SLTI	Set less than variable, comparison instruction
BEQ	Branch on equal, conditional branch instruction
J	Unconditionally jump
JAL	Unconditionally jump and link (Save next PC in $r31)
JR	Unconditionally jump to the instruction whose address is in $rs
JALR	Jump and link register
LW	Load word from data memory (assign word-aligned)
SW	Store word to data memory (assign word-aligned)
NOP	No operation

B. Division of work

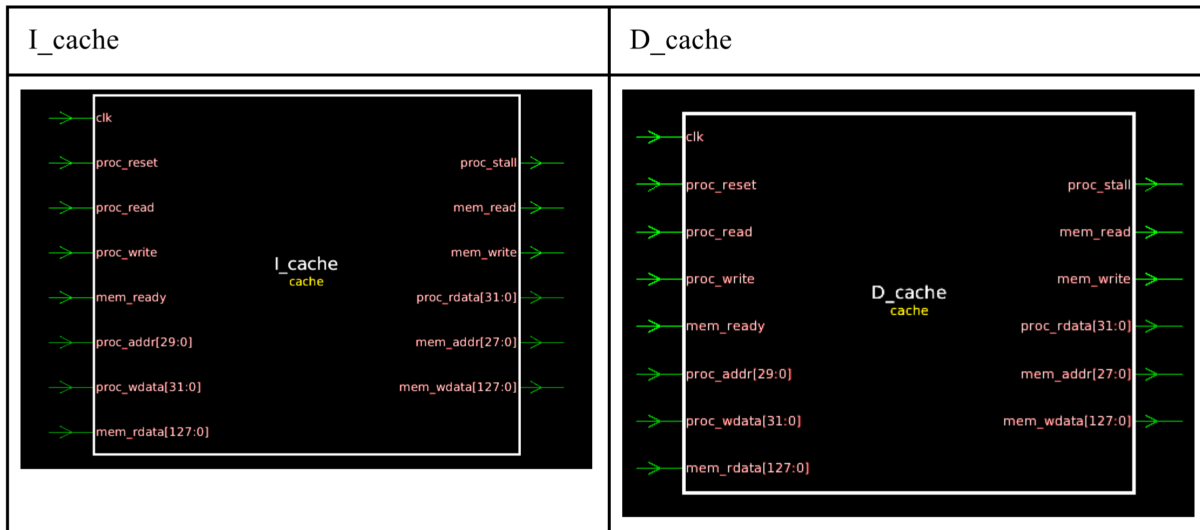
- 趙祐毅
  Baseline: PCsrcLogic, nextPCcalculator, Branch prediction, hazard detection
  Extension: Branch prediction
- 羅志軒
  Baseline: forwarding unit, ALU, precontrol unit, main control unit
  Extension: Two-level cache
- 許凱傑
  Baseline: Initial architecture design, registers, other wires.
  Extension: Multiplication/ division

Baseline

A. Cache

1. Design Architecture

We use Direct-mapped cache here.

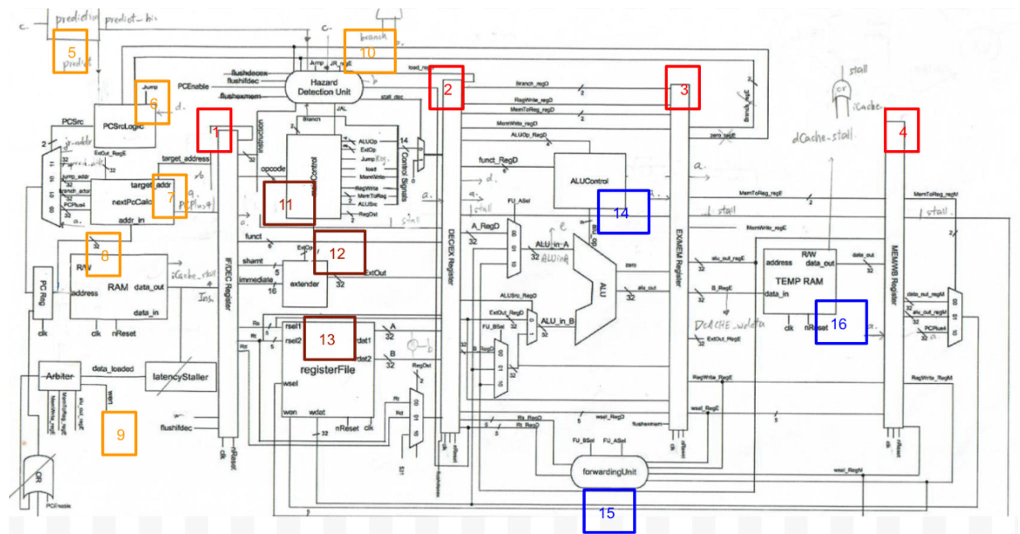


B. MIPS

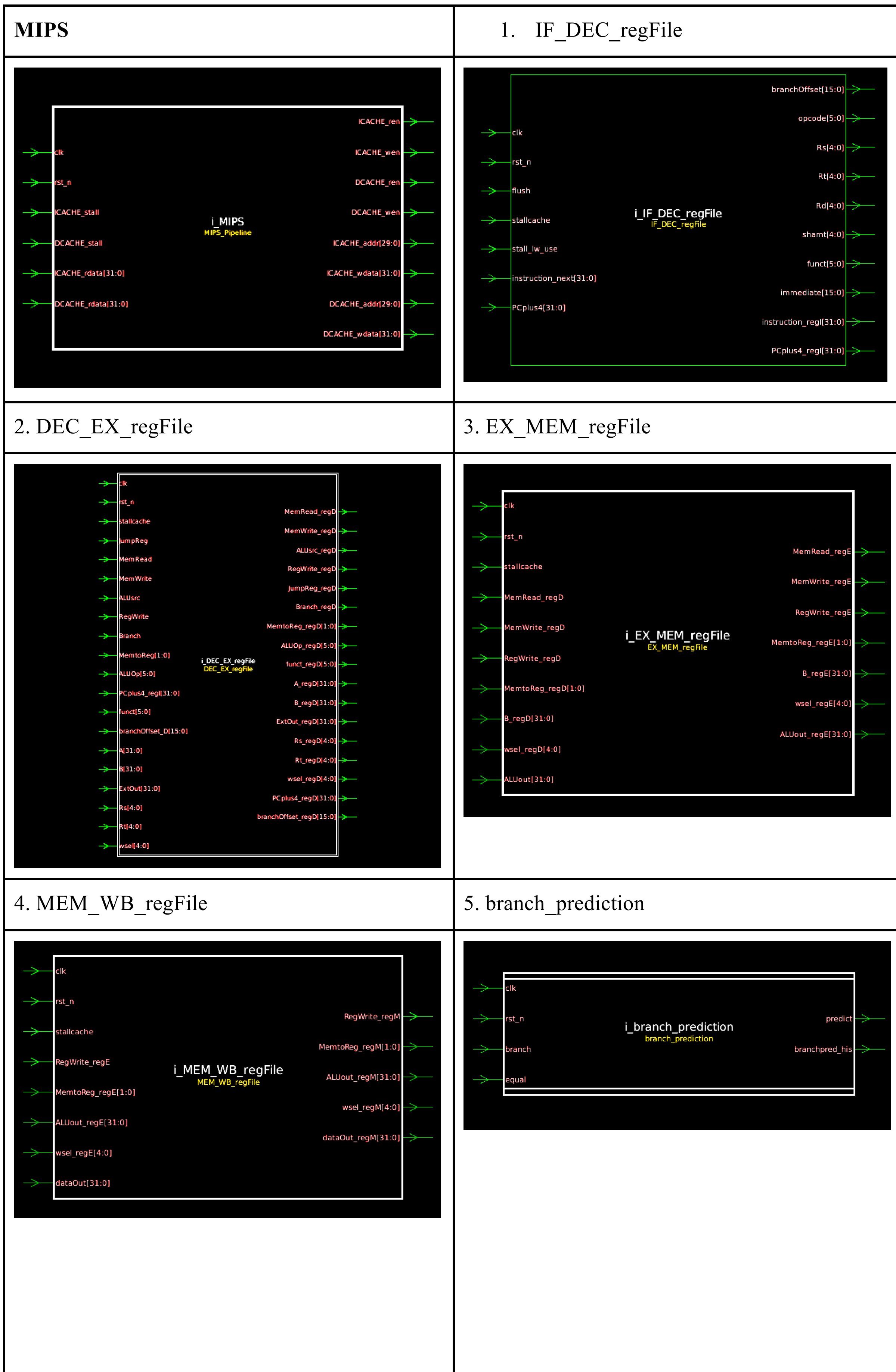
1. Design Architecture

- <u>Modules</u>

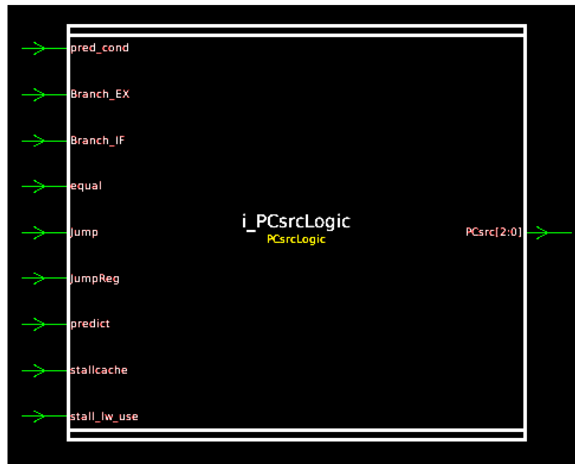
1) IF_DEC_regFile
2) DEC_EX_regFile
3) EX_MEM_regFile
4) MEM_WB_regFile
5) branch_prediction
6) PCSrcLogic
7) nextPCcalculator
8) *I_cache*
9) precontrolDec
10) Hazard_detection
11) mainControl
12) Extender
13) registerFile
14) ALU
15) forwarding
16) *D_cache*



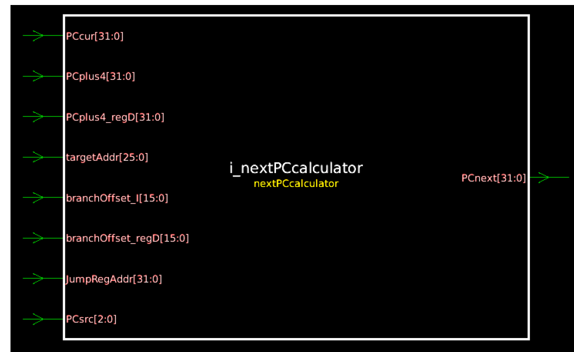
- Module inputs/ outputs



6. PCSrcLogic



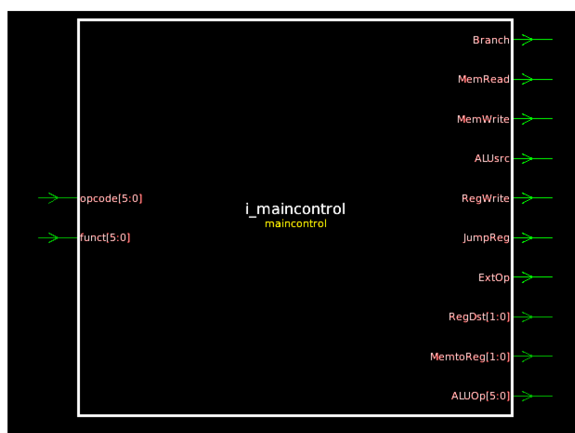
7. nextPCcalculator



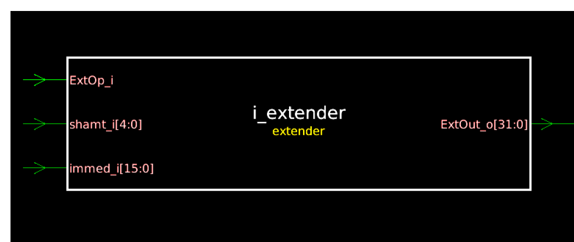
9. precontrolDec

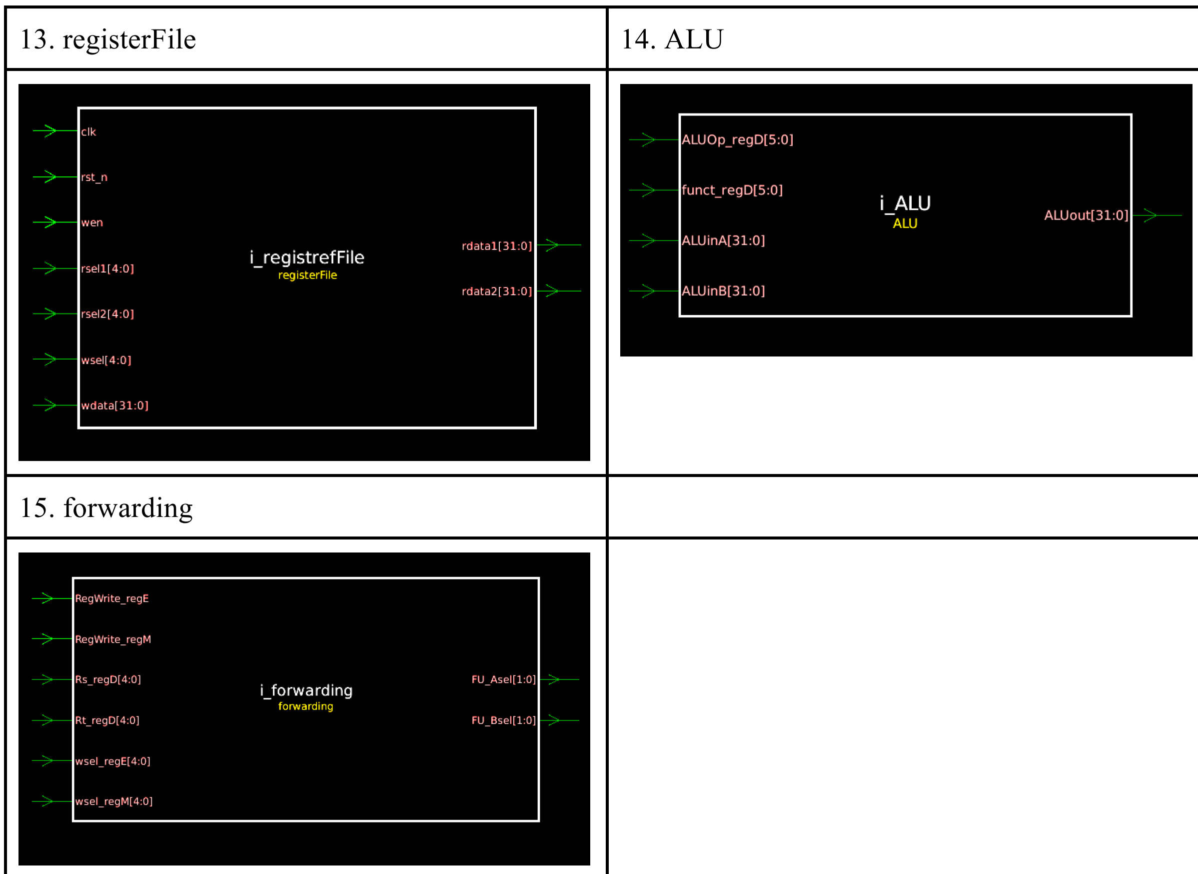
10. Hazard_detection

11. mainControl



12. Extender





2. Special Design

branch in EX

We do Branch prediction in IF stage, and get Branch result in EX stage. The reasons are:

a) If we get branch result in DEC stage, we need to do several steps. We should read data from registerFile, compare the data, if the result is different from prediction, we should pass the signal to PCsrcLogic to set PC value. Therefore, it could yield too long critical path.

b) If we do branch in DEC stage, we have to redesign the forwarding unit to ensure that the value read from registerFile is correct. It would cost some effort.

J/ JAL

We decide whether the instruction if J/JAL in IF stage. If so, nextPCcalculator can get the address immediately, so that we can save a NOP.

Pre-control Unit

We use a ‘Pre-control Unit’ to decode instruction in IF stage, so that branch/ J/ JAL can be processed immediately. Also, PCscrLogic, nextPCcalculator and hazard detection unit work to do successive steps.

JALR/ JR

We do JALR/ JR in EX stage because we need to read from registerFile as the reason we do branch in EX stage.

JAL/ JALR

When we get JAL/ JALR, we need to pass PCplus4 signal and save it. Rather than add one more bus only for PCplus4, we take advantage of original buses. We add mux to the output of registerFile, one has input PCplus4, another has input 0. Then, we use adding process to save the value. We can decrease two 32-bit registers and

two 32-bit wires.

Flush mechanism

When branch prediction is wrong or JR/ JALR happens, we need to flush IF and DEC signals. Here is our design:

a) We add flush signal input to IF/ DEC register, which do the same thing as reset.
b) For DEC stage, we add mux to the maincontrol signal and set MemRead, MemWrite, RegWrite, Branch_DEC, JumpReg to '0'. We don't need to reset all signals, so as to save 'mux'.

Hazard detection, Branch prediction, PCsrcLogic, nextPCcalculator relationship

Hazard detection has some main tasks:

a) Catch 'stall' signal from D_cache and I_cache. Then, output stall signal.
b) Tell is there's lw_use hazard.
c) Input branch result and the result of branch prediction, and output 'pred_cond' signal. (If 1 means prediction wrong, we need to flush.)

Branch prediction:

According to the prediction and result, change the state of FSM.

PCsrcLogic

According to stall, Jump, Branch signals, output right PCsrc signal.

nextPCcalculator

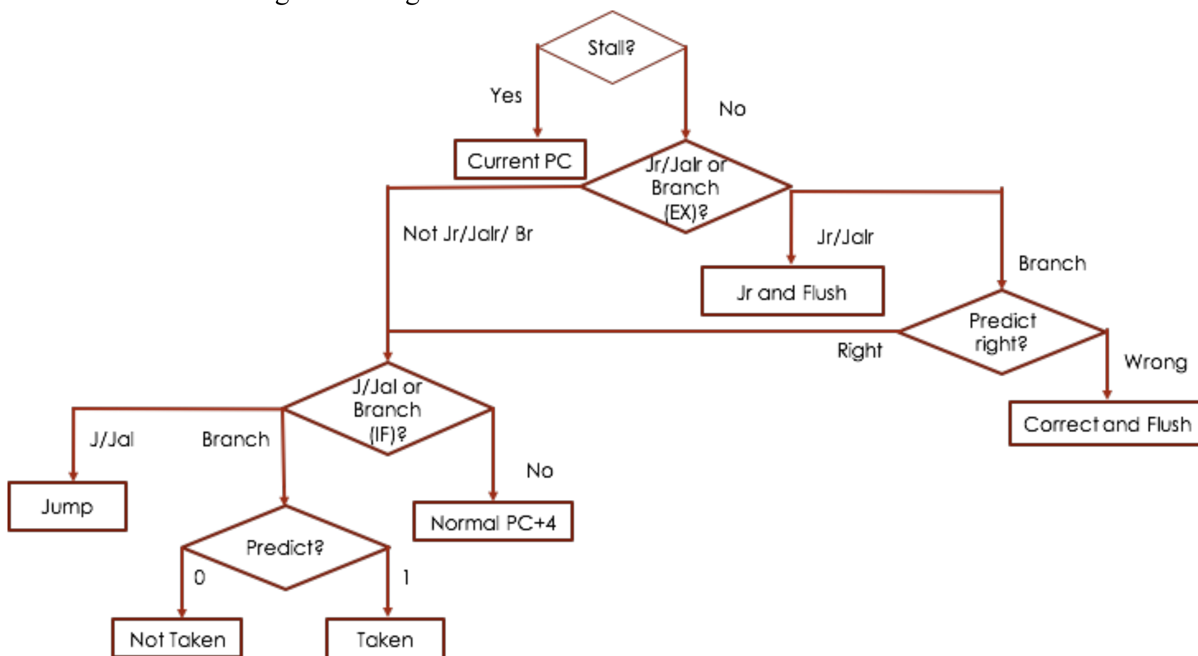
Output 'nextPC' according to PCsrc signal.

PCsrcLogic Priority

When working on PCsrcLogic module, we need to consider the priority of the signals. As following:

a) Stall is #1
b) Signals from EX stages (JR/ JALR/ Branch) is #2
c) If branch prediction right or nothing happens, check the signal of IF stage. (Branch/ J/ JAL)

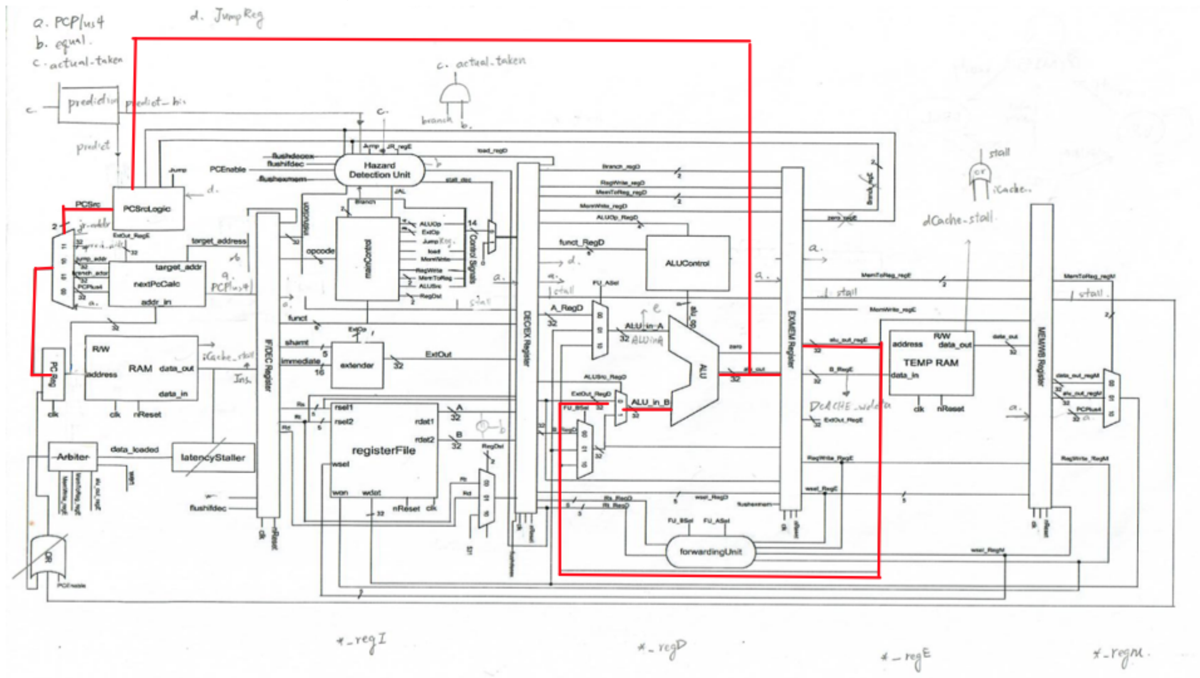
* J/ branch should not appear at the same time.
* Refer to the following block diagram.



3. Critical path

Our critical path happens when:

a) Branch happens.
b) Branch takes data from MEM stage, so forwarding unit sends data back.
c) ALU calculation tells branch result.
d) PCsrc tells branch prediction was wrong.
e) Send correct PC to PCreg.



C. Synthesis Result and Analysis

- Direct-mapped I-cache/ D-cache
- Cell Area: 291905 (μm^2)
- Clock cycle: 4 (ns)
- Timing of hasHazard TB: 8266 (ns)
- Area(Cell)*T: ***2.41 * 10^(9) (μm^2 * ns)***

```
Number of references:                         201

Combinational area:          154847.012229
Noncombinational area:       137058.261446
Net Interconnect area:       1959636.806183

Total cell area:             291905.273675
Total area:                  2251542.079858
1
design_vision>
```

Extension

A. Branch Prediction

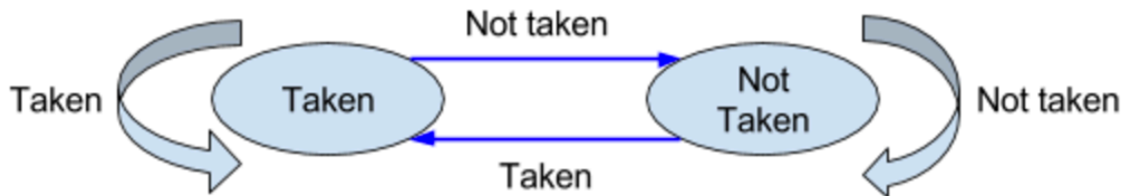
1. Target

We want to know the relationship between prediction policies and different testbenches.

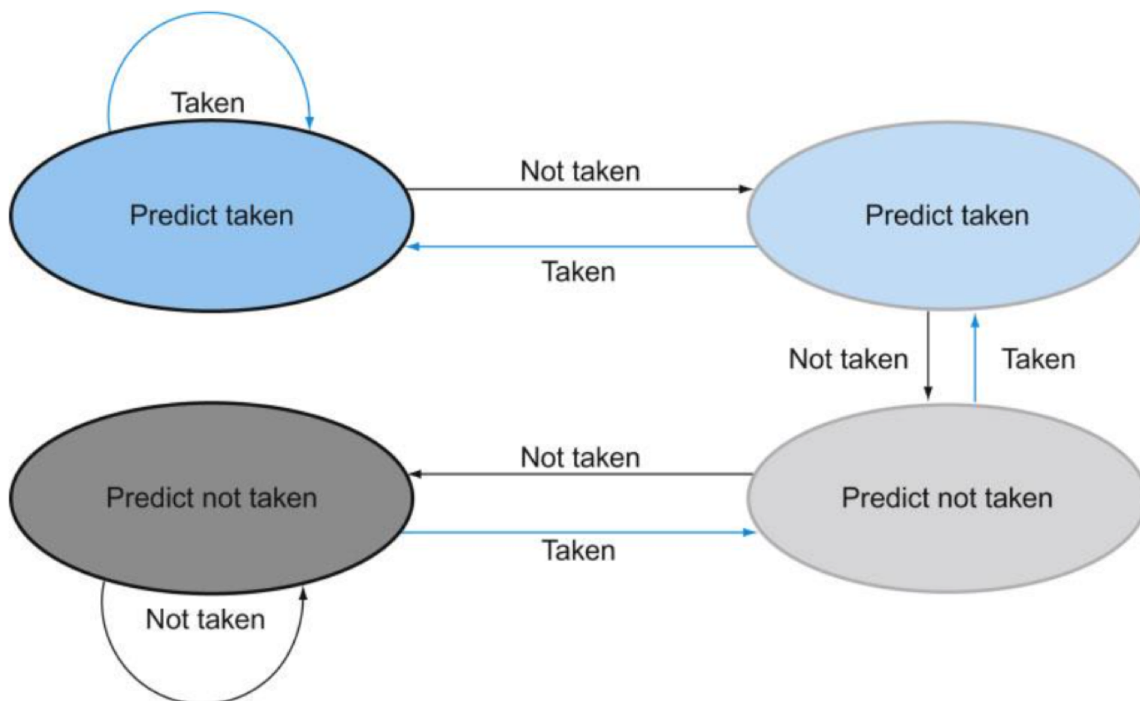
2. Design Architecture

4 designs

a) No BPU(taken): Always do taken
b) No BPU(not taken): Always do untaken
c) 1-bit BPU



d) 2-bit BPU



3. Synthesis Result and Analysis

Experiment design (A, B, C)

A: number of branch not taken

B: number of branch not taken, taken interleaved

C: number of branch taken

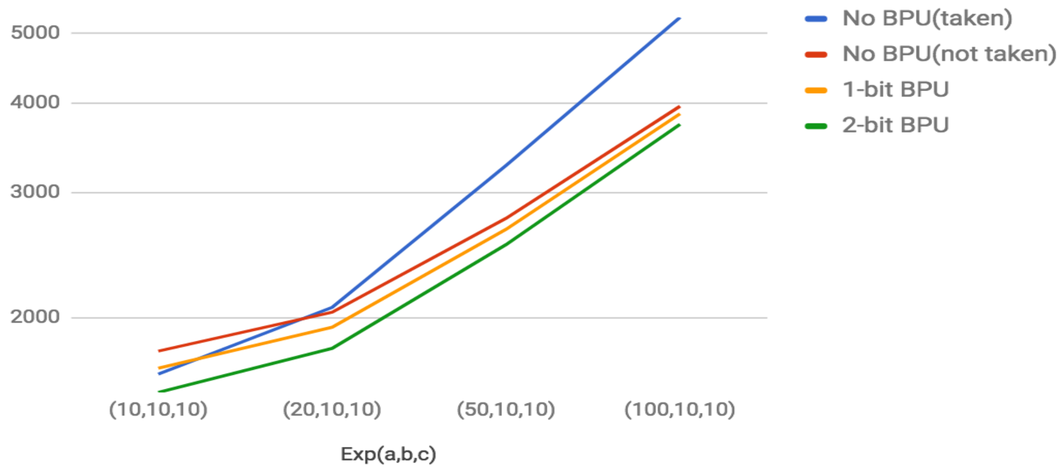
Clock cycle = 6 ns

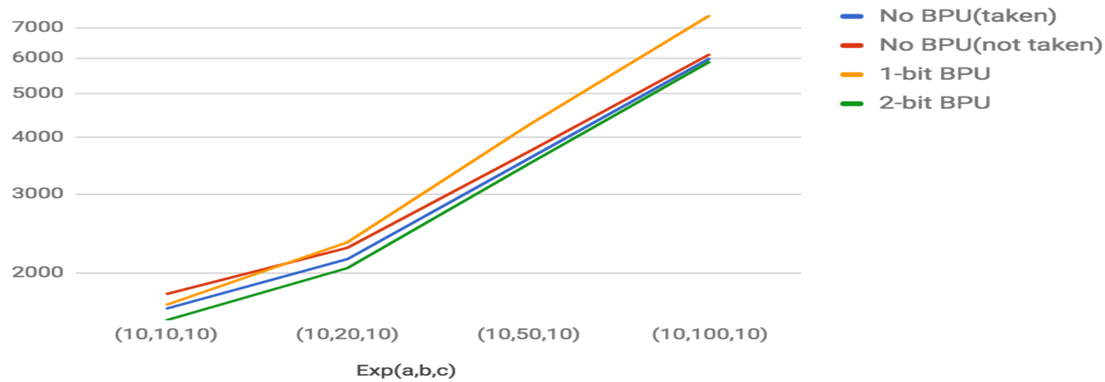
Exp (A, B, C)	No BPU(taken)	No BPU(not taken)	1-bit BPU	2-bit BPU
(10,10,10)	1395	1431	1377	1311
(20,10,10)	1695	1611	1557	1491
(50,10,10)	2595	2151	2097	2031
(100,10,10)	4095	3051	2997	2931
(10,20,10)	1755	1791	1857	1671
(10,50,10)	2835	2871	3297	2751
(10,100,10)	4635	4671	5697	4551
(10,10,20)	1839	X	1821	1761
(10,10,50)	3105	X	3081	3027
(10,10,100)	5265	X	5235	5193

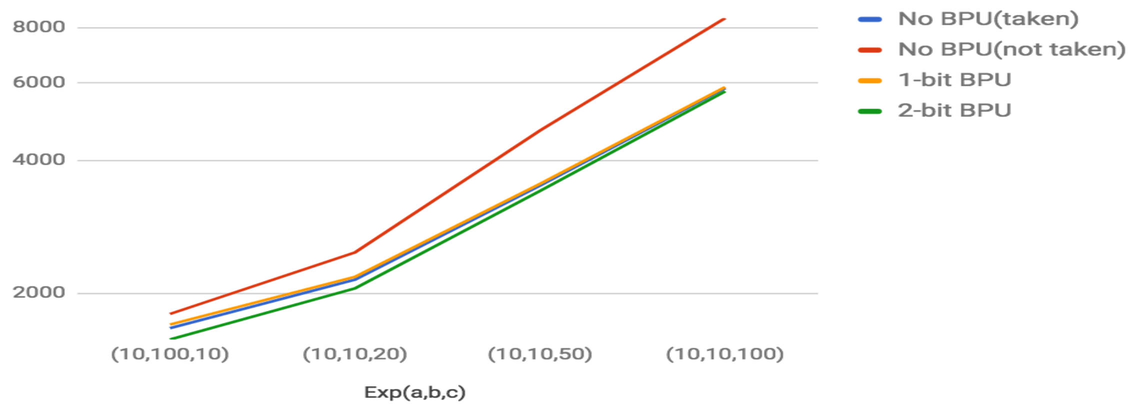
Because the clock cycle is not long enough for 'NO BPU', we can't finish the experiment in some situation. We prolonged the clock cycle to 8ns.

Clock cycle = 8 ns

Exp (A, B, C)	No BPU(taken)	No BPU(not taken)	1-bit BPU	2-bit BPU
(10,10,10)	1668	1796	1700	1572
(20,10,10)	2068	2036	1940	1812
(50,10,10)	3268	2756	2660	2532
(100,10,10)	5268	3956	3860	3732
(10,20,10)	2148	2276	2340	2052
(10,50,10)	3588	3716	4260	3492
(10,100,10)	5988	6116	7460	5892
(10,10,20)	2148	2476	2180	2052
(10,10,50)	3508	4676	3540	3412
(10,10,100)	5828	8396	5860	5732







Result analysis

1. 2-bit predictor yeilds the best result among the three graph.
2. No BPU(taken) did the worst when changing 'A' (number of not taken)
3. No BPU(not taken) did the worst when changing 'C' (number of taken)
4. 2-bit BPU does only a bit better than others because if can't handle the situation of taken/ not taken interleaved.
5. We can try to use the taken history to predict. However, it could enlarge the size of BPU. Also, we can't tackle the condition of continuous Branch (previous has not been proven before the next branch comes.)

B. Multiplication / Division

1. Target

a) support 4 kinds of instruction, namely,

mult \$rt, \$rs ({\$HI, \$LO} = \$rt ×\$rs)
div \$rt, \$rs (\$HI = \$rt / \$rs, \$LO = \$rt % \$rs)
mfhi \$rd (\$rd = \$HI)
mflo \$rd (\$rd = \$LO)

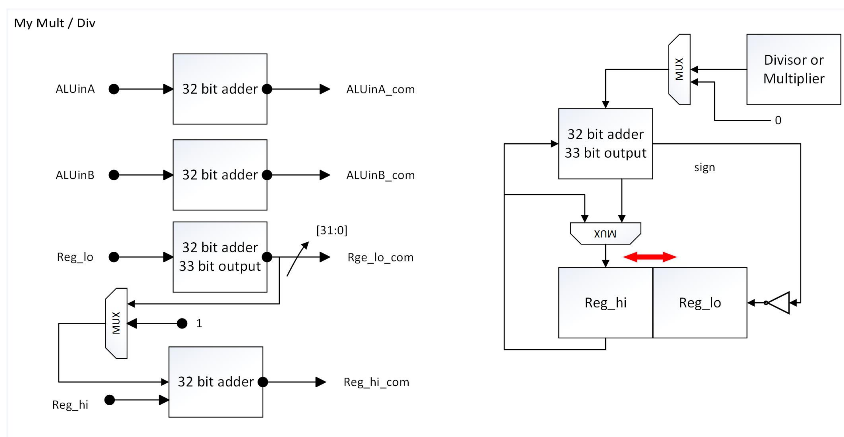
b) support signed multiplication and division

2. Iterative approach

a) Reason

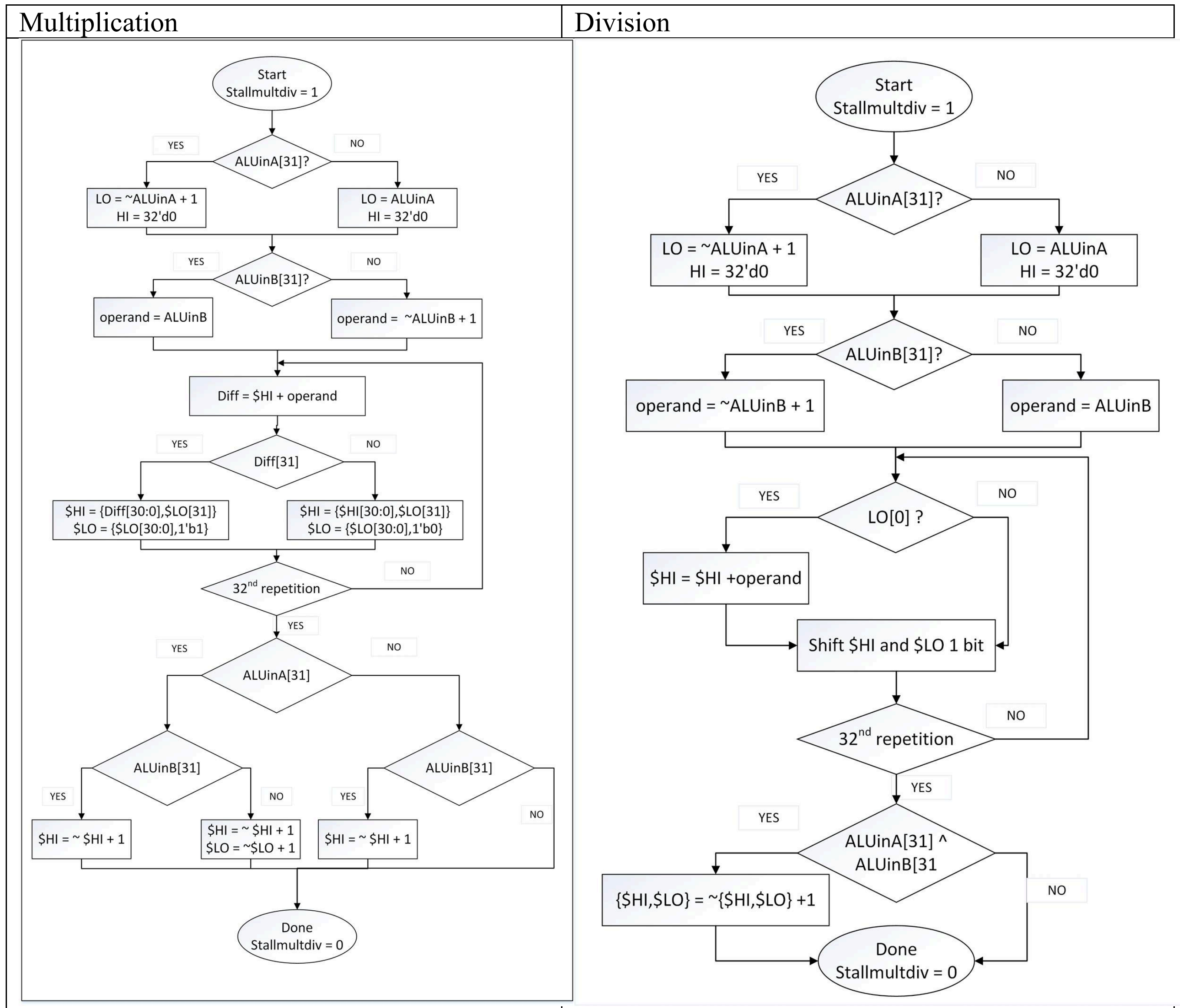
1) less area: Pipeline approach will need 2 or more mult/div unit, many registers to store results
2) maybe total less execution cycle:
   The basic implementation of pipeline approach takes 20 cycles to finish mult/div and iterative approach needs 36 cycles. However, pipelined approach will need 1 more cycle for other instruction.
   Therefore, if the percentage of mult/div instructions doesn't exceed 1/16, iterative approach takes less number of execution cycles to finish.

b) Design Architecture



1) 4 32-bit adder to implement 2's complement
   Convert Reg_hi, Reg_lo, multiplicand(dividend) and multiplier(divisor) into their 2's complement if needed.
   Instead of using 64-bit adder to do 2's complement of {Reg_HI, Reg_LO}. We implement it by using a MUX to do both signed multiplication and division result.
2) 1 32-bit adder to do subtraction and addition
   We don't need 32-bit subtractor by store divisor as negative number.
3) stallmultdiv: stall other instructions until mult/div finished

c) Flow Chart



3. Simulation

a) testbench

unsigned

```
int a = 1;
int i = 2;
int b = 0;
for(; i<9; i++){
    b = a * i;
    cout << b << "\n";
    a = b;
}
cout << "quotient: " << a / 2
     << ",remainder: " << a % 2;
cout << "\nquotient: " << a / 5
     << ",remainder: " << a % 5;
cout << "\nquotient: " << a / 11
     << ",remainder: " << a % 11;
cout << endl;
```

signed

```
int a = -1;
int i = 2;
int b = 0;
for(; i<9; i++){
    b = a * i;
    cout << b << "\n";
    a = b;
}
cout << "quotient: " << a / 2
     << ",remainder: " << a % 2;
cout << "\nquotient: " << a / 5
     << ",remainder: " << a % 5;
cout << "\nquotient: " << a / -11
     << ",remainder: " << a % -11;
cout << endl;
```

b) Result

unsigned	signed
b03026@cad34:src	b03026@cad34:src
--	--
START!!! Simulation Start	START!!! Simulation Start .....
--	--
Correct: addr= 0, data= 2, expected= 2.	Correct: addr= 0, data= -2, expected= -2.
Correct: addr= 1, data= 6, expected= 6.	Correct: addr= 1, data= -6, expected= -6.
Correct: addr= 2, data= 24, expected= 24.	Correct: addr= 2, data= -24, expected= -24.
Correct: addr= 3, data= 120, expected= 120.	Correct: addr= 3, data= -120, expected= -120.
Correct: addr= 4, data= 720, expected= 720.	Correct: addr= 4, data= -720, expected= -720.
Correct: addr= 5, data= 5040, expected= 5040.	Correct: addr= 5, data= -5040, expected= -5040.
Correct: addr= 6, data= 40320, expected= 40320.	Correct: addr= 6, data= -40320, expected= -40320.
Correct: addr= 7, data= 20160, expected= 20160.	Correct: addr= 7, data= -20160, expected= -20160.
Correct: addr= 8, data= 0, expected= 0.	Correct: addr= 8, data= 0, expected= 0.
Correct: addr= 9, data= 8064, expected= 8064.	Correct: addr= 9, data= -8064, expected= -8064.
Correct: addr=10, data= 0, expected= 0.	Correct: addr=10, data= 0, expected= 0.
Correct: addr=11, data= 3665, expected= 3665.	Correct: addr=11, data= 3665, expected= 3665.
Correct: addr=12, data= 5, expected= 5.	Correct: addr=12, data= -5, expected= -5.
==	==
\(^o^)/ CONGRATULATIONS!! The simulation result is PASS!!!	\(^o^)/ CONGRATULATIONS!! The simulation result is PASS!!!
==	==
Simulation complete via $finish(1) at time 3015 NS + 0	Simulation complete via $finish(1) at time 3075 NS + 0

1) 603 cycles for unsigned and 615 cycles for signed

4. Synthesis

a) Origin MIPS_Pipeline.v

1) Area: 31658 um^2
2) Cycle: 4.0nS (for other instruction except MULT/DIV)

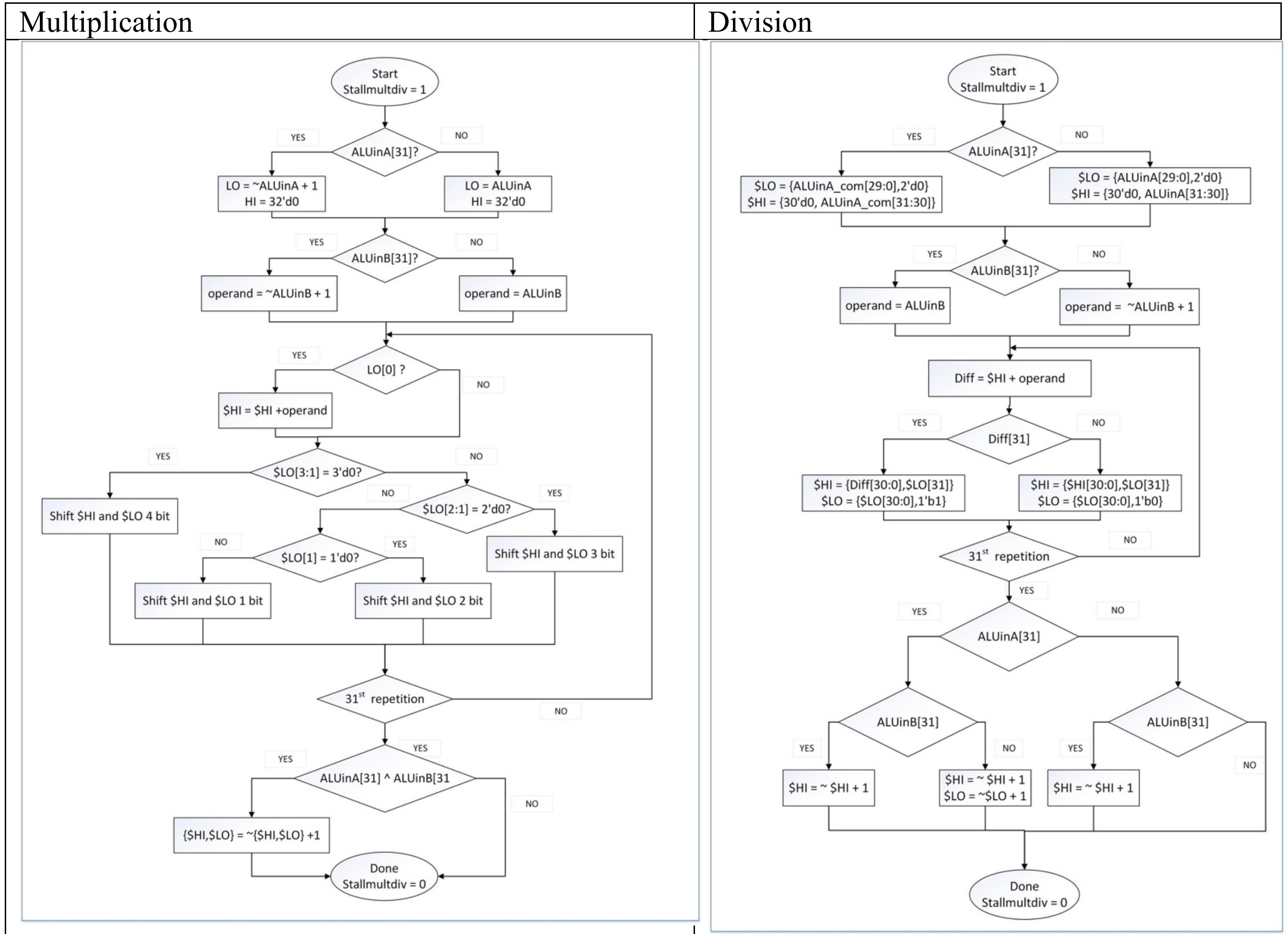
b) This Design

1) Area: 49171 um^2
2) Cycle: 4.8nS
3) Execution Time: 2508 / 2556 nS

c) Improvement

1) use 3 comparators to omit iterations only with shift. See whether Reg_lo[1], Reg_lo[2:1] and Reg_lo[3:1] are all zero(s) and shift multiple bits correspondingly.(This improvement is inspired by the presentation of 黃安、吳宇、蔣采容)
   i. Area: 48218 um^2
   ii. Cycle: 4.4 nS
   iii. Execution Time: 1621.4 / 1665.4 nS
2) omit the 32nd iteration(i.e. there is only 31 iterations for mult/div)
   below is the result of both a) and b)
   i. Area: 49171 um^2
   ii. Cycle: 4.2 nS
   iii. Execution Time: 1518.3 / 1560.3 nS

3) Flow chart



C. Two-Level Cache

1. Description

Cache is a part of memory hierarchy between CPU and memory used to reduce the average cost to access data from the main memory. Most modern computer have at least three independent caches: an instruction cache, a data cache, and a unified (or separate) L2 cache for both D-cache and I-cache. In this part, we try to implement a separate L2 cache with NC-verilog code. The total size of our L2 caches is 256 words (128 words for each). In each L2 cache, there are 32 blocks with 4 words in each block.
To demonstrate the effectiveness of L2 cache, we compare the execution time and average memory access time between caches with different architecture below.

2. Experiment

a) <u>Direct-mapped L1 cache vs 2-way set associative L1 cache (hasHazard)</u>

	Direct-mapped	**Direct-mapped**	**2-way set associative**
Synthesization cycle	2.0	2.0	2.0
Test bench cycle	4.0	4.8	4.8

Execution time	8266000 PS	9943200 PS	8944800 PS
Execution cycle	2066	2071	1863.5
Cell area (um^2)	290605.063220	290605.063220	341116.289725
Total area (um^2)	2197741.587116	2197741.587116	2629474.717551

From the table above, we can see that 2-way set associative cache makes progress on execution cycle, yet in the same time expanding cell area from 290605.063220 to 341116.289725. The extra area of 2-way set associative cache is mainly from different replacement algorithm (LRU), in which release bit must be used.

(In this part, for some reason, we can't run 2-way set associative cache under same smallest test bench cycle time (4.0 ns) as Direct-mapped cache, so we didn't compare execution time between them.)

b) L1 cache vs L1+L2 cache (simulation time in TestBed_L2Cache.v)

	L1	L1+L2
L1 L2 Associative	Direct	Two-way
Original L2 tb	55495 NS	54465 NS
Nb = 3	4905 NS	4855 NS
Nb = 5	14985 NS	14695 NS
Nb = 8	42135 NS	41485 NS
Nb = 10	70595 NS	67585 NS

The table above demonstrates the improvement in execution time with L2 cache. As the number of Fibonacci Series increases, we can compare the difference of effectiveness between L1 cache and L1+L2 cache.

c) L2 cache avg. memory access time

(Hit time: 10 NS, Miss penalty: 58 NS)

	L1 Miss rate	L2 Miss rate	Avg. MAT without L2	Avg. MAT with L2
Nb = 3	0.08	0.035	14.64	10.1624
Nb = 5	0.022	0.013	11.276	10.0166
Nb = 8	0.009	0.006	10.522	10.003132
Nb = 10	0.006	0.009	10.348	10.003132

This table we can see how L2 cache benefit to L1 cache. With L2 cache, average memory access time reduces almost close to hit time, especially when miss rate is high.

d) L1 cache vs L1+L2 cache (synthesis in TestBed_hasHazard.v)

	L1	L1+L2	L1+L2
L1 L2 Associative	Direct	Two-way	Two-way
Synthesization cycle	2.0	5.0	2.0
Test bench cycle	4.0	10.0	9.5
Execution time	8266000 PS	18305000 PS	17693750 PS
Total area (um^2)	2197741.587116	7511647.332870	7592994.770748

This table shows some problem of our L2 cache. Logically, L1+L2 cache will beat L1 cache in Execution time. However, for some reason, although we set cycle to 5.0 while synthesization, our L2 cache only passes TestBed_hasHazard.v when setting cycle to larger than 10.0 in test bench. We guess the cause probably is we didn't add flip-flop between MIPS and cache, which makes delay affect our signal and leads to some error.

References

- MIPS design architecture
- Branch prediction -- wiki