ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HCM

BỘ MÔN ĐIỆN TỬ

EE3043: Computer Architecture

Laboratory Report

GVHD: Dr. Trần Hoàng Linh

TA: Cao Xuân Hải

Lớp: L01

Tên	MSSV
Nguyễn Đắc Tâm	2114714

Mục lục

Milestone 3

Design of Pipelined Processors

1. Introduction

RISC-V instruction set architecture (ISA) là một kiến trúc có mã nguồn mở dùng để thiết kế vi xử lý cho máy tính và đã trở nên phổ biến cho những ứng dụng trong công nghiệp và học thuật. Nó dựa trên nguyên tắc Reduced Instruction Set Computer (RISC) với mục đích đơn giản hóa tập lệnh và cải thiện tính hiệu quả cũng như hiệu năng cho vi xử lý. RV32I là những tập lệnh xử lý số nguyên nằm trong kiến trúc RISC-V và được thiết kế để triển khai số 32 bit. Trong báo cáo, chúng em sẽ trình bày thiết kế của vi xử lý RV32I pipelined 5 tầng dựa trên thiết kế single-cycle processor đã thiết kế trước đó. Pipelined processor được thiết kế với 4 models bao gồm non-forwarding, forwarding, always-taken và two-bit prediction.

Trong thiết kế vi xử lý, người ta luôn muốn tăng tốc độ của vi xử lý để đạt được những hiệu quả tốt nhất, và pipeline là một kỹ thuật giúp tăng đáng kể tốc độ lên nhiều lần. Trong thiết kế pipeline processor không thể tránh khỏi những loại hazards bao gồm structural hazard, data hazard và control hazard. Vì phiên bản single-cycle processor trước đó được thiết kế với cấu trúc Harvard nên structural hazard sẽ không phải vấn đề cần xử lý. Ở non-forwarding model, data hazard và control hazard sẽ được xử lý bằng cách stall những lệnh có dữ liệu phụ thuộc vào lệnh trước đó và flush những lệnh không đúng đã được đưa vào các tầng phía trước tầng execute. Forwarding model sẽ xử lý được data hazard bằng cách bypass những data wire cần thiết về tầng execute, control hazard vẫn được stall và flush như non-forwarding model. Always-taken model kế thừa toàn bộ forwarding model nhưng có thêm branch target buffer (BTB) để dự đoán trước địa chỉ cần nhảy khi gặp lệnh branch và jump. Two-bit prediction model kế thừa toàn bộ always-taken model và sử dụng phương pháp dự đoán 2 bit để cải thiện tính đúng đắn ở model trước.

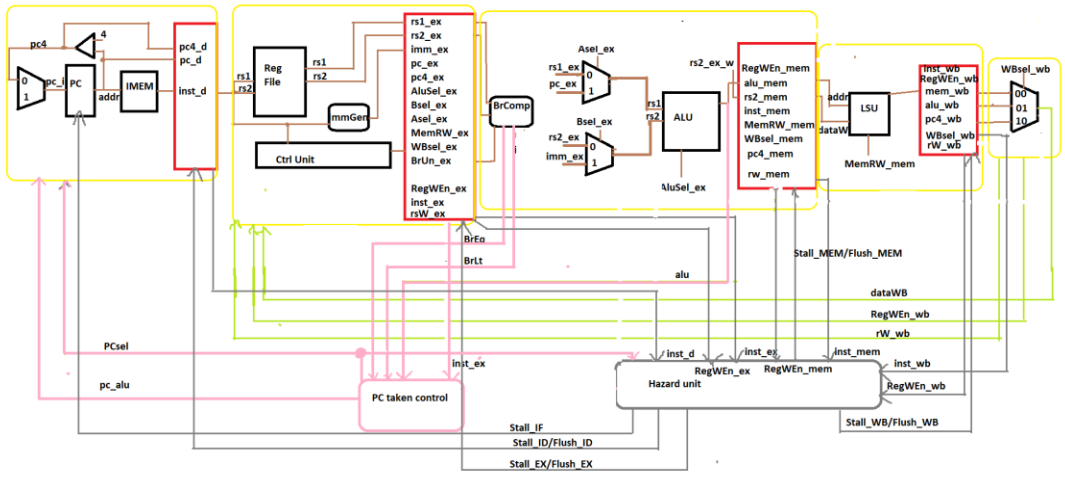
Để hoàn thành được thiết kế, chúng em xin cảm ơn thầy Trần Hoàng Linh đã dạy những kiến thức về cấu trúc máy tính và xin cảm ơn anh Cao Xuân Hải đã hướng dẫn cụ thể cách thực hiện thiết kế cũng như những kiến thức liên quan.

2. Design Strategy

Pipelined processor được thiết kế với 5 tầng instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM) và write back (WB). 4 models được thiết kế bao gồm non-forwarding, forwarding, always-taken, two-bit prediction. Các model đều được dựa trên model trước đó để xây dựng thêm các khối bổ sung để giải quyết data hazard và control hazard.

2.1. Non-forwarding

Ở model này, các instructions kế tiếp luôn được fetch vào pipeline, data hazard và control hazard chỉ được xử lý bằng cách flush những lệnh sai hoặc stall pipeline. Điều này sẽ làm giảm hiệu năng của vi xử lý. Stall và flush bằng cách sử dụng enable và reset bất đồng bộ trong flipflop. Khối hazard unit có chức năng phát hiện data hazard và control hazard để điều khiển tín hiệu stall và flush.



Hình 1. Block diagram của non-forwarding model

Data hazard bao gồm 3 trường hợp:

- Lệnh thứ 2 đọc thanh ghi được ghi ở lệnh thứ nhất. Khi này 1 trong 2 hoặc cả 2 thanh ghi đưa vào bộ ALU ở tầng execute chưa được cập nhật giá trị vì lệnh trước đó đang ở tầng memory và chưa được đưa vào tầng write back để lưu lại register file. Do đó, cần phải stall lệnh ở tầng IF, ID và flush lệnh ở tầng execute.

- Lệnh thứ 3 đọc thanh ghi được ghi ở lệnh thứ nhất. Khi này 1 trong 2 hoặc cả 2 thanh ghi đưa vào bộ ALU ở tầng execute chưa được cập nhật giá trị vì lệnh thứ nhất trước đó đang ở tầng write back và lưu lại register file nhưng lệnh thứ 3 đã lấy giá trị ở tầng ID trước đó. Do đó, cần phải stall lệnh ở tầng IF, ID và flush lệnh ở tầng EX, MEM.
- Đọc thanh ghi được ghi sau lệnh load. Khi lệnh hiện tại đang ở tầng ID để lấy dữ liệu thanh ghi thì lệnh load trước đó vẫn chưa tới tầng MEM để load giá trị và trả về register file. Do đó cần phải stall lệnh ở tầng IF, ID và flush lệnh ở tầng EX, MEM, WB.

Control hazard: có 2 trường hợp khi gặp lệnh branch là nhảy và không nhảy, trường hợp không nhảy sẽ không thành vấn đề vì những lệnh kế tiếp sau đó vẫn được thực thi. Nhưng trường hợp nhảy thì khi câu lệnh đi tới lệnh execute thì mới tính toán được có nhảy hay không, do đó cần flush đi 2 câu lệnh ở tầng IF và ID vì khi đã nhảy sẽ không thực hiện 2 lệnh đã đi vào ở 2 tầng trước đó.

Tính toán IPC:

Giả sử :

Xác suất gặp data hazard là $P_d = 10\%$

Số lượng câu lệnh cần delay trung bình là $\Delta_d = 2$

Số lượng lệnh rẽ nhánh là $P_{br} = 20\%$

Xác suất nhảy khi gặp lệnh rẽ nhánh là $P_{mís} = 70\%$

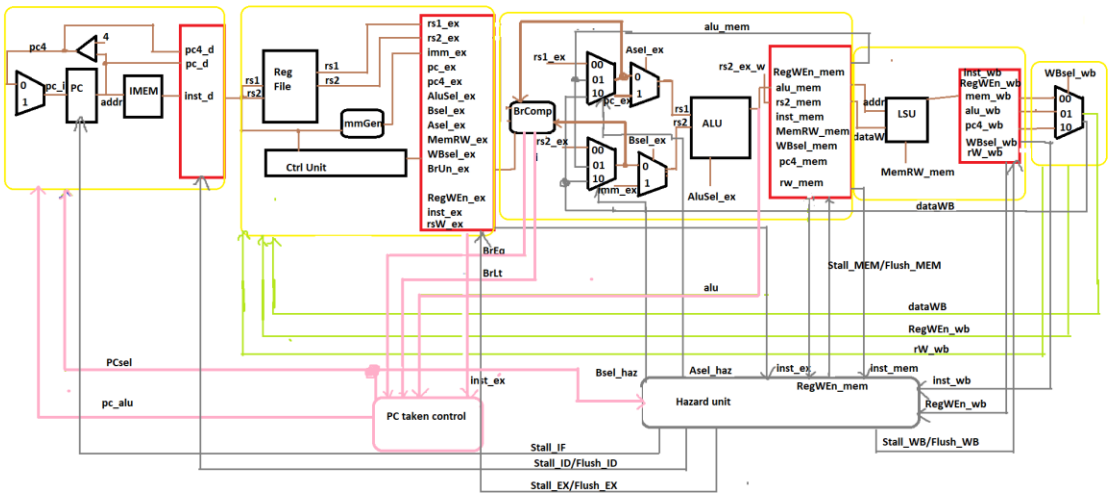
Số câu lệnh cần delay là $\Delta_{br} = 2$

Ta tính được:

$$IPC = \frac{N_{inst}}{N_{cycle}} = \frac{1}{P_d\,\Delta_d + P_{br}P_{mís}\Delta_{br} + 1} = \frac{1}{10\% \times 2 + 20\% \times 70\% \times 2 + 1} = 67.6\%$$

2.2. Forwarding

Ở model này, các loại data hazard sẽ được xử lý bằng kỹ thuật forwarding. Các dây data chứa dữ liệu cần được ghi vào register file ở tầng EX và MEM sẽ được bypass vào các thanh ghi được đọc ở khối ALU.



Hình 2. Block diagram của forwarding model

Trong model này, tầng EX được thêm 2 bộ mux để lựa chọn dữ liệu bypass từ EX/MEM và MEM/WB khi hazard unit phát hiện data hazard. Riêng trường hợp đọc thanh ghi được ghi sau lệnh load, tầng IF, ID, EX cần được stall để lệnh load đưa dữ liệu vào thanh ghi sau đó chuyển tới tầng WB thì mới có thể tiếp tục pipeline. Control hazard vẫn được flush tương tự như non-forwarding model.

Tính toán IPC:

Giả sử :

Xác suất gặp load data hazard là $P_d = 5\%$

Số lượng câu lệnh cần delay là $\Delta_d = 1$

Số lượng lệnh rẽ nhánh là $P_{br} = 20\%$

Xác suất nhảy khi gặp lệnh rẽ nhánh là $P_{mis} = 70\%$

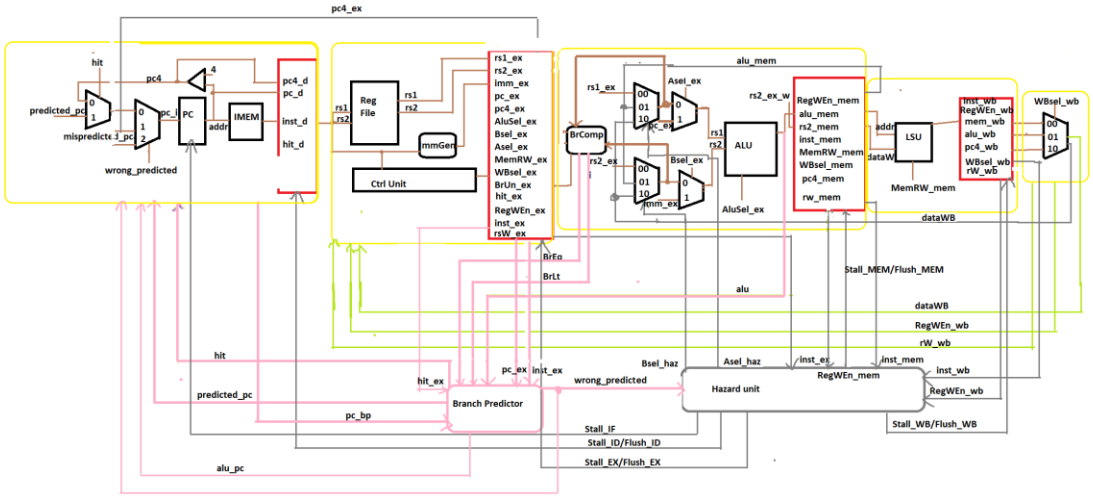
Số câu lệnh cần delay là $\Delta_{br} = 2$

Ta tính được:

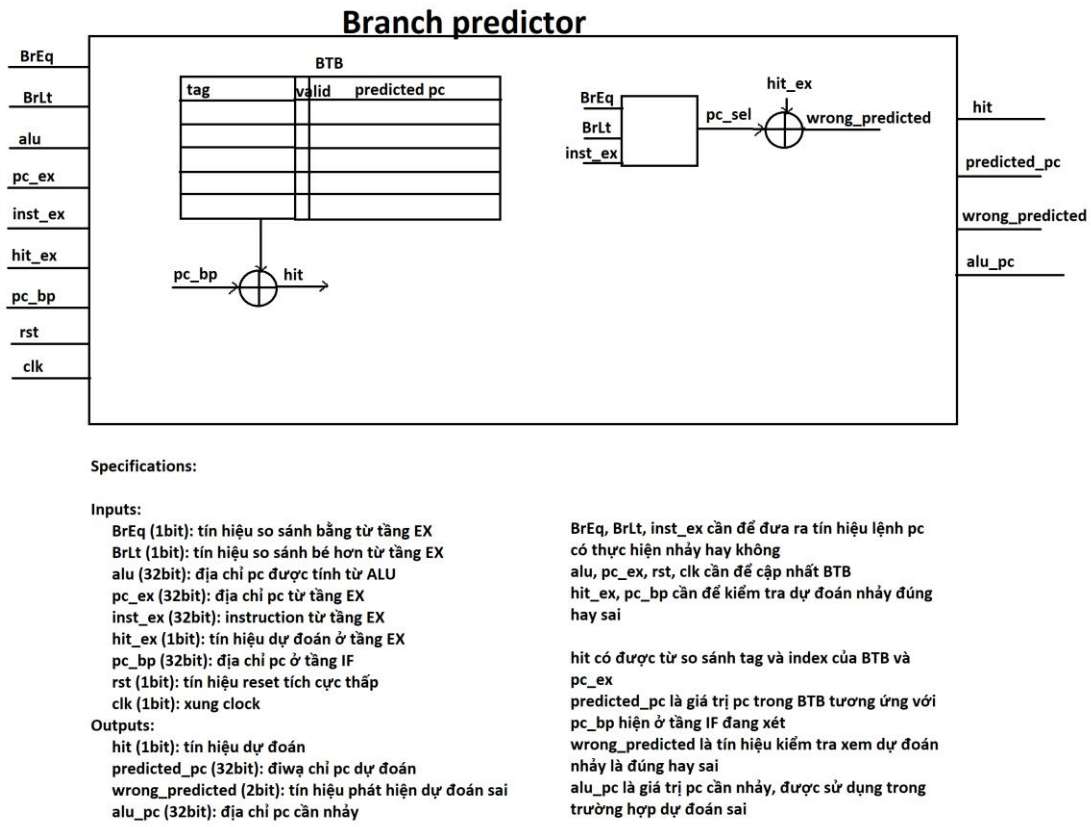
$$IPC = \frac{N_{inst}}{N_{cycle}} = \frac{1}{P_d\,\Delta_d + P_{br}P_{mis}\Delta_{br} + 1} = \frac{1}{5\% \times 1 + 20\% \times 70\% \times 2 + 1} = 75.2\%$$

2.3. Always – taken

Ở model này, kỹ thuật static branch prediction được sử dụng để dự đoán địa chỉ pc cần nhảy tới khi gặp lệnh branch và jump. Branch target buffer (BTB) được thiết kế để lưu lại predicted pc và giá trị địa chỉ của lệnh branch hoặc jump để nhận biết. Vì địa chỉ pc chứa 32 bit nên sẽ cần dung lượng lớn để lưu trữ, ở đây BTB được thiết kế với chiều dài là 32. Ở mỗi phần tử BTB, 20 bit đầu tiên là tag sẽ chứa 20 bit cao của địa chỉ pc ở lệnh branch hoặc jump, 1 bit valid để nhận biết phần tử BTB đã được cập nhật hay chưa, 32 bit cuối sẽ chứa địa chỉ pc mà lệnh branch hoặc jump nhảy tới. Giá trị index của BTB sẽ là 5 bit đầu của địa chỉ pc ở lệnh branch hoặc jump vì depth của BTB được thiết kế bằng 32.



Hình 3: Block diagram của always-taken model



Hình 4: Block diagram của khối branch predictor trong always-taken model

Tính toán IPC:

Giả sử :

Xác suất gặp load data hazard là $P_d = 5\%$

Số lượng câu lệnh cần delay là $\Delta_d = 1$

Số lượng lệnh rẽ nhánh là $P_{br} = 20\%$

Xác suất nhảy khi gặp lệnh rẽ nhánh là $P_{mis} = 30\%$

Số câu lệnh cần delay là $\Delta_{br} = 2$

Ta tính được:

$$IPC = \frac{N_{inst}}{N_{cycle}} = \frac{1}{P_d\ \Delta_d + P_{br}P_{mís}\Delta_{br} + 1} = \frac{1}{5\% \times 1 + 20\% \times 30\% \times 2 + 1} = 85.5\%$$

3. Verification Strategy

Kiểm tra các model qua những trường hợp hazard như sau:

Case 1

```
i0: add r5, r3, r2
i1: xor r6, r5, r1
i2: sub r9, r3, r5
i3: or  r2, r7, r5
i4: sll r4, r5, r5
```

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

Case 3

```
i0: add r4, r3, r2
i1: beq r5, r6, _L0
i2: sub r9, r5, r1
...
i8: _L0
    sll r4, r5, r1
i9: xor r6, r8, r2
```

Các lệnh assembly được sử dụng trong từng trường hợp như sau:

- Case 1:
  addi x1, x0, 1
  addi x2, x0, 2
  addi x3, x0, 3
  addi x4, x0, 4
  addi x5, x0, 5
  addi x6, x0, 6
  addi x7, x0, 7
  addi x8, x0, 8
  addi x9, x0, 9

  add x5, x3, x2
  xor x6, x5, x1
  sub x9, x3, x5
  or x2, x7, x5
  sll x4, x5, x5

- Case 2:
  addi x1, x0, 1
  addi x2, x0, 2
  addi x3, x0, 3
  addi x4, x0, 4
  addi x5, x0, 5
  addi x6, x0, 6
  addi x7, x0, 7

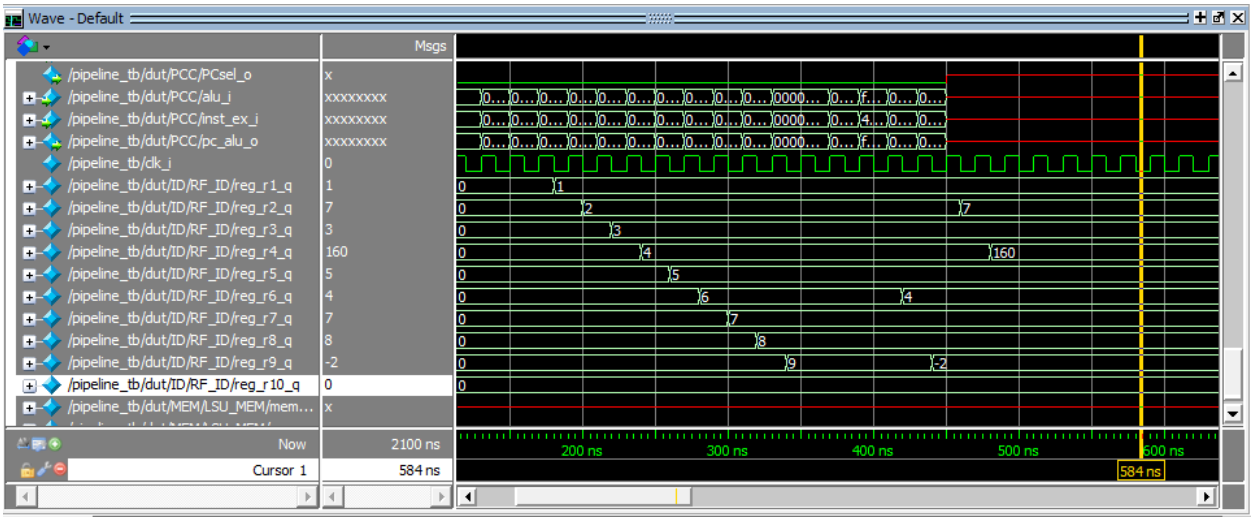
```
addi x8, x0, 8
addi x9, x0, 9
sw x6, 0(x1)

add x4, x3, x2
lw x5, 0(x1)
sub x9, x5, x1
or x2, x7, x5
sll x4, x5, x1
```

- Case 3:

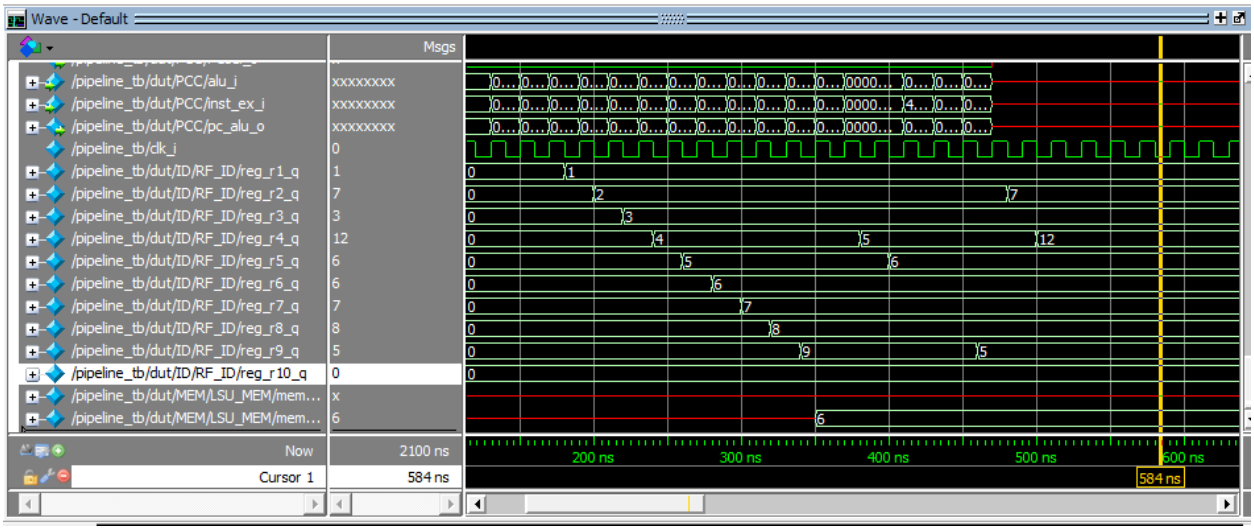
```
addi x1, x0, 1
addi x2, x0, 2
addi x3, x0, 3
addi x4, x0, 4
addi x5, x0, 5
addi x6, x0, 6
addi x7, x0, 7
addi x8, x0, 8
addi x9, x0, 9

add x4, x3, x2
beq x5, x4, _L0
sub x9, x5, x1

addi x5, x0, 50
addi x6, x0, 60
addi x7, x0, 70
addi x8, x0, 80
addi x9, x0, 90

_L0:
sll x4, x5, x1
xor x6, x8, x2
```

3.1. Non-forwarding

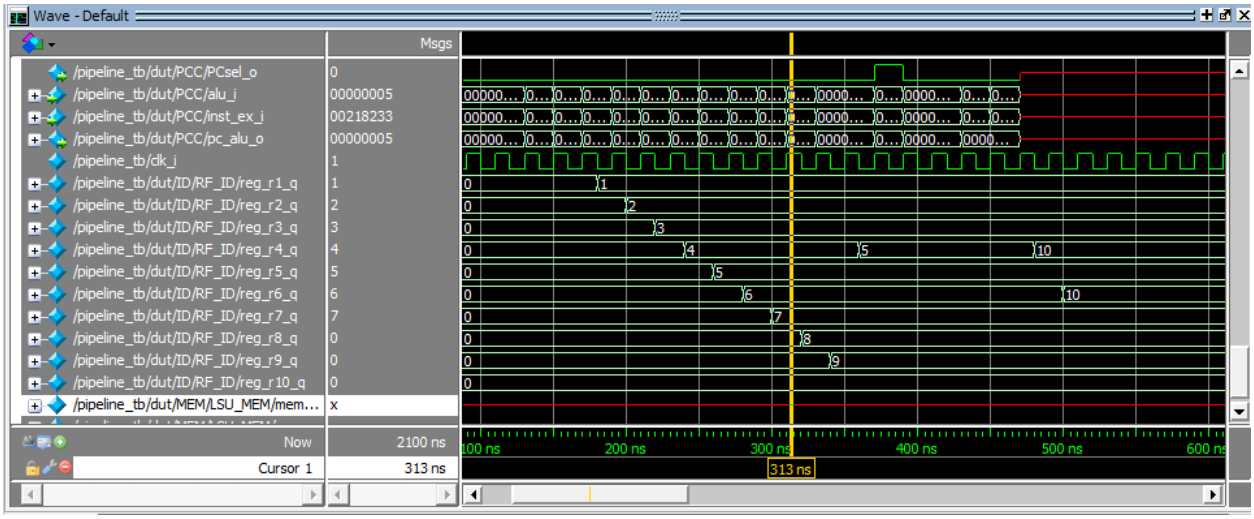
Case 1:


Wave - Default
Msgs
/pipeline_tb/dut/PCC/PCsel_o
/pipeline_tb/dut/PCC/alu_i
/pipeline_tb/dut/PCC/inst_ex_i
/pipeline_tb/dut/PCC/pc_alu_o
/pipeline_tb/clk_i
/pipeline_tb/dut/ID/RF_ID/reg_r1_q
/pipeline_tb/dut/ID/RF_ID/reg_r2_q
/pipeline_tb/dut/ID/RF_ID/reg_r3_q
/pipeline_tb/dut/ID/RF_ID/reg_r4_q
/pipeline_tb/dut/ID/RF_ID/reg_r5_q
/pipeline_tb/dut/ID/RF_ID/reg_r6_q
/pipeline_tb/dut/ID/RF_ID/reg_r7_q
/pipeline_tb/dut/ID/RF_ID/reg_r8_q
/pipeline_tb/dut/ID/RF_ID/reg_r9_q
/pipeline_tb/dut/ID/RF_ID/reg_r10_q
/pipeline_tb/dut/MEM/LSU_MEM/mem...
Now
2100 ns
Cursor 1
584 ns
200 ns
300 ns
400 ns
500 ns
600 ns

Case 2:


Wave - Default
Msgs
/pipeline_tb/dut/PCC/alu_i
/pipeline_tb/dut/PCC/inst_ex_i
/pipeline_tb/dut/PCC/pc_alu_o
/pipeline_tb/clk_i
/pipeline_tb/dut/ID/RF_ID/reg_r1_q
/pipeline_tb/dut/ID/RF_ID/reg_r2_q
/pipeline_tb/dut/ID/RF_ID/reg_r3_q
/pipeline_tb/dut/ID/RF_ID/reg_r4_q
/pipeline_tb/dut/ID/RF_ID/reg_r5_q
/pipeline_tb/dut/ID/RF_ID/reg_r6_q
/pipeline_tb/dut/ID/RF_ID/reg_r7_q
/pipeline_tb/dut/ID/RF_ID/reg_r8_q
/pipeline_tb/dut/ID/RF_ID/reg_r9_q
/pipeline_tb/dut/ID/RF_ID/reg_r10_q
/pipeline_tb/dut/MEM/LSU_MEM/mem...
/pipeline_tb/dut/MEM/LSU_MEM/mem...
Now
2100 ns
Cursor 1
584 ns
200 ns
300 ns
400 ns
500 ns
600 ns

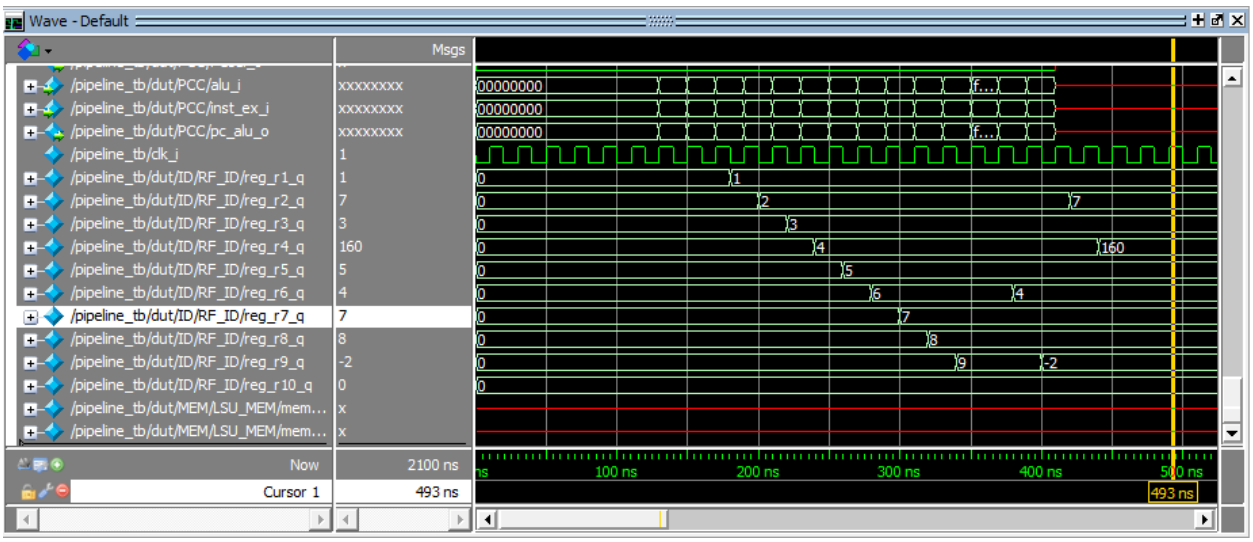
Case 3:



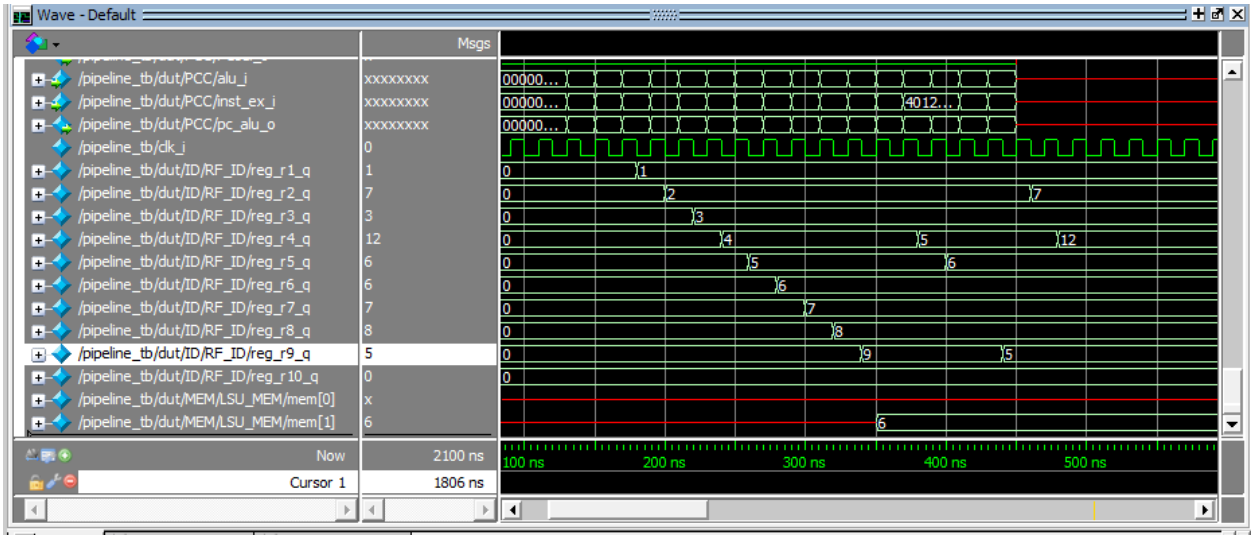
Các trường hợp gặp data hazard thì chương trình phải stall và flush một số lệnh do đó làm giảm hiệu suất của processor.

3.2. Forwarding

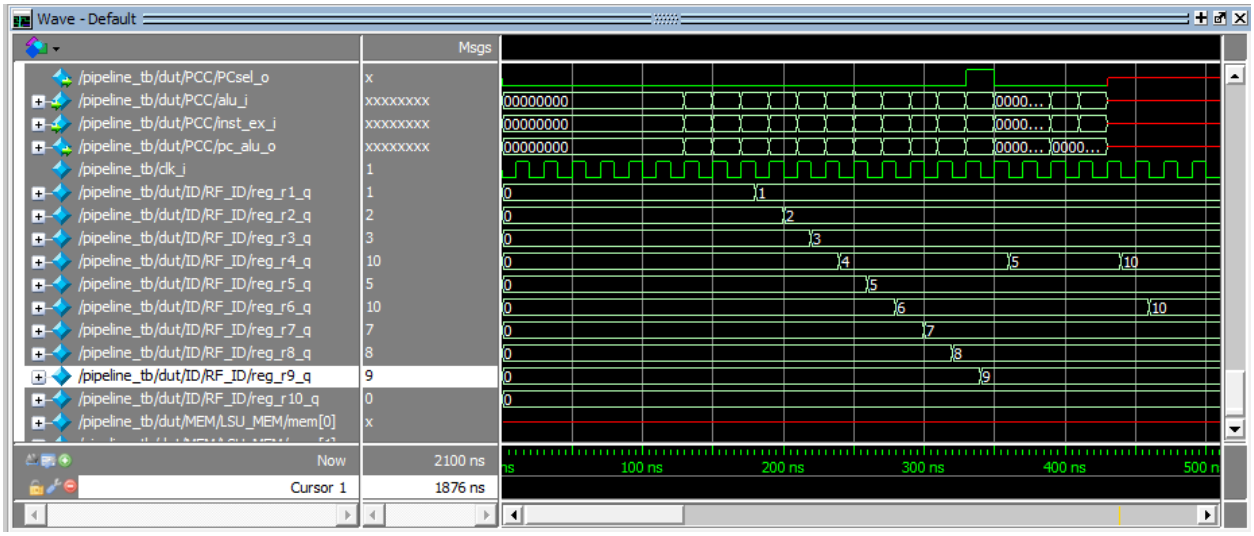
Case 1:



Case 2:



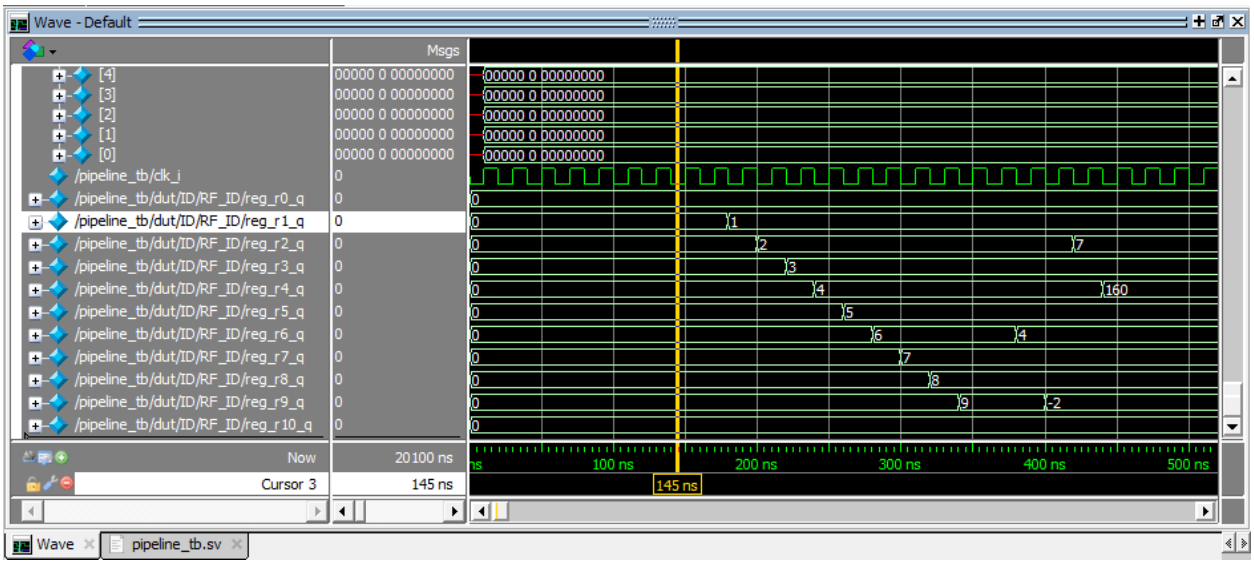
Case 3:



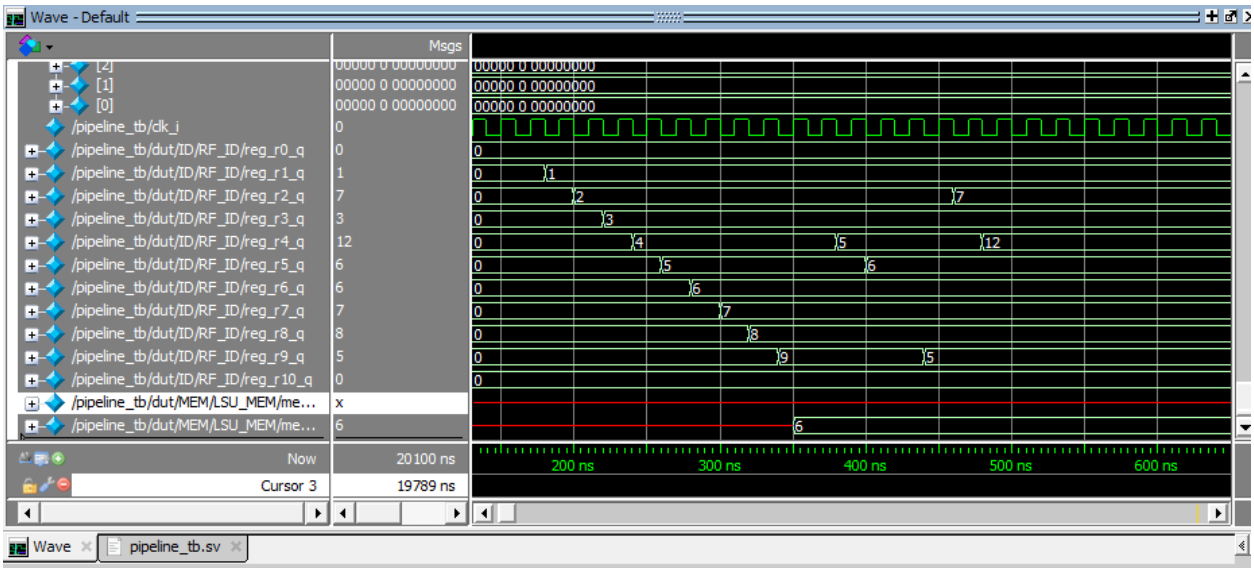
Data hazard ở forwarding model được giải quyết và riêng trường hợp đọc thanh ghi được ghi sau lệnh load bị stall 1 chu kỳ. Control hazard vẫn được flush như model trước.

3.3. Always-taken

Case 1:



Case 2:



Case 3:

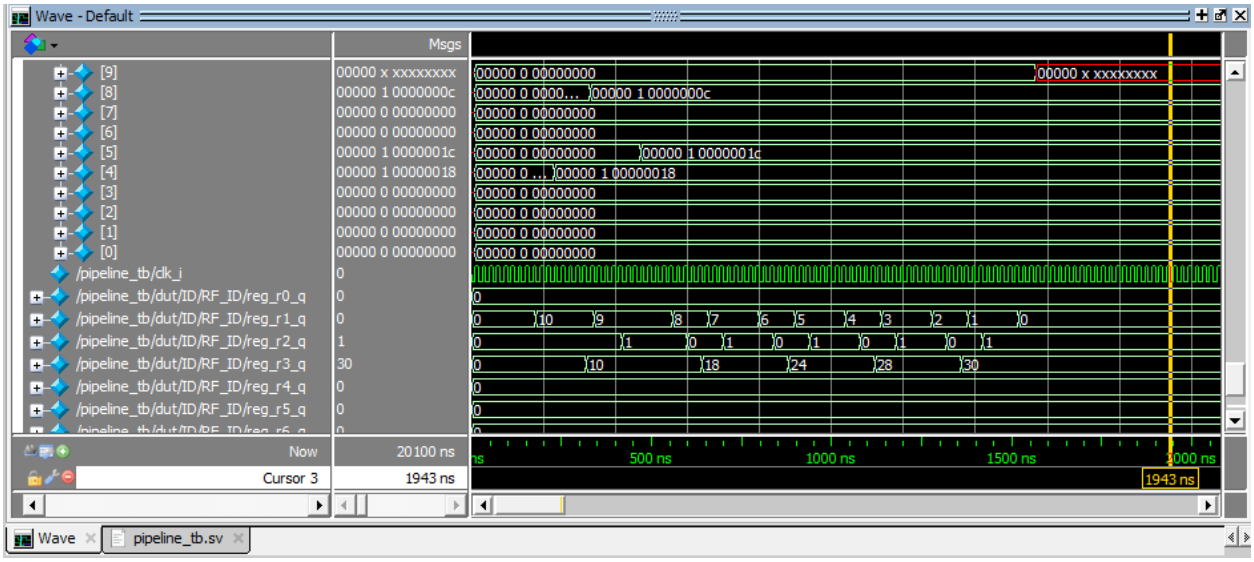
Ở trường hợp này, assembly sẽ thay đổi để có thể thực hiện nhiều lệnh nhảy giúp dễ kiểm chứng hơn.

Assembly:

```
addi x1, x0, 10

addi x8, x0, 1
```

```
add x3, x0, x0

_COMPARE:

and x2, x1, x8

beq x2, x0, _ADD_EVEN

jal x10, _DECREASE

_ADD_EVEN:

add x3, x1, x3

_DECREASE:

sub x1, x1, x8

bne x1, x0, _COMPARE

_EXIT:
```

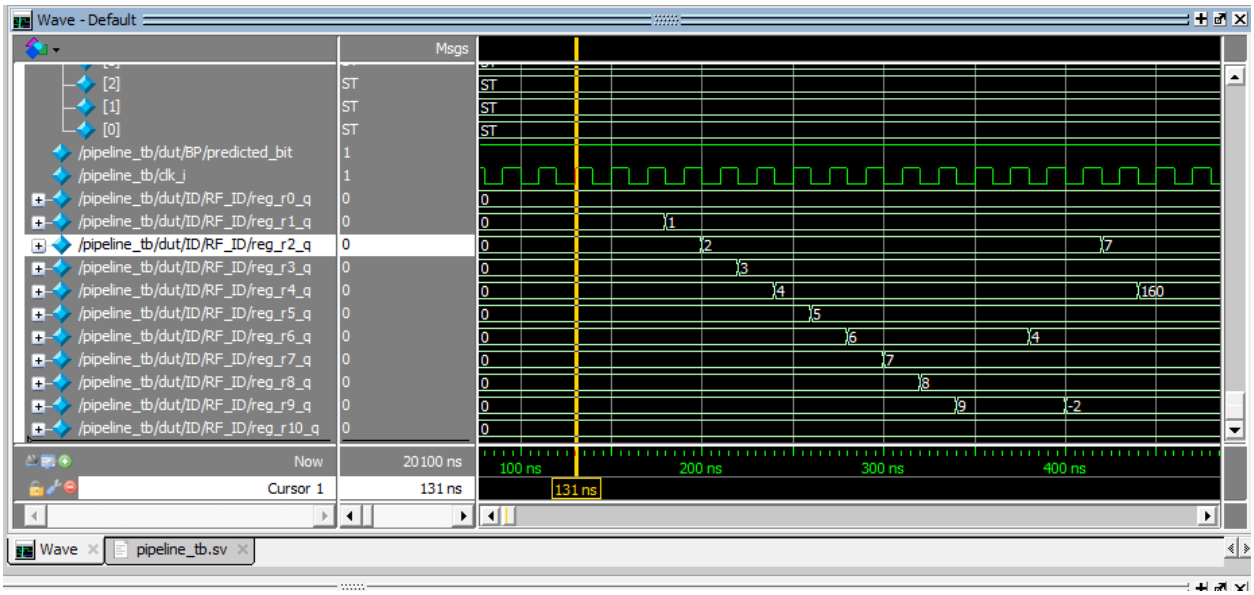
Chương trình assembly trên sẽ tính tổng các số chẵn từ 1 đến 10 và lưu vào thanh ghi x3.



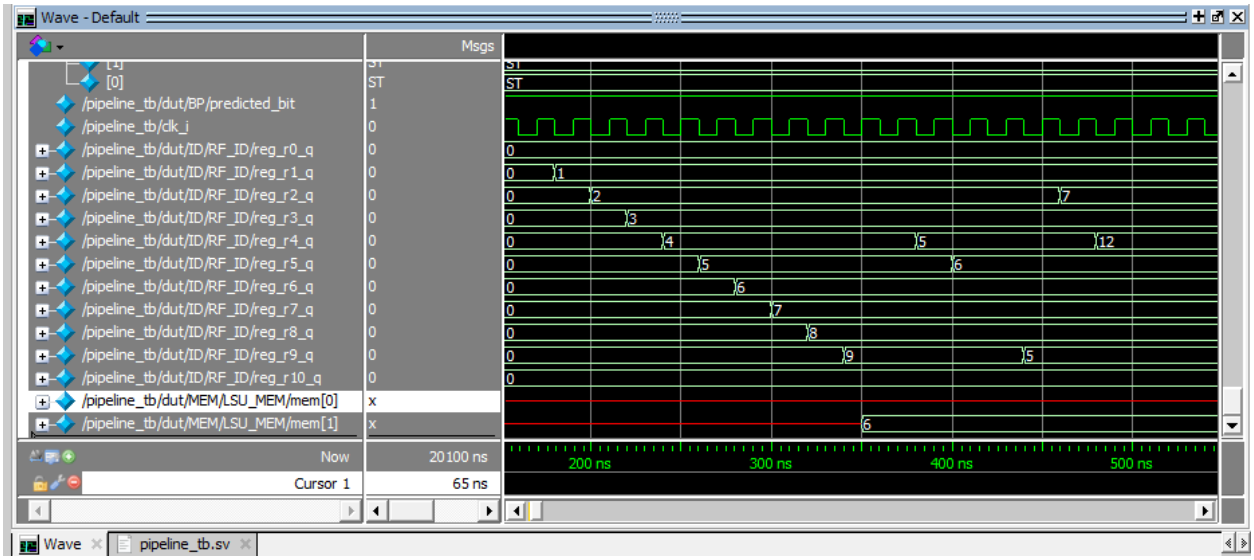
Như waveform trên, BTB[4], BTB[5], BTB[8] đã lưu giá trị tag và predicted pc. Khi gặp các lệnh nhảy thì ở model này bit dự đoán sẽ luôn là 1, nghĩa là luôn chọn nhảy đến địa chỉ pc được dự đoán.

3.4. Two-bit prediction

Case 1:



Case 2:



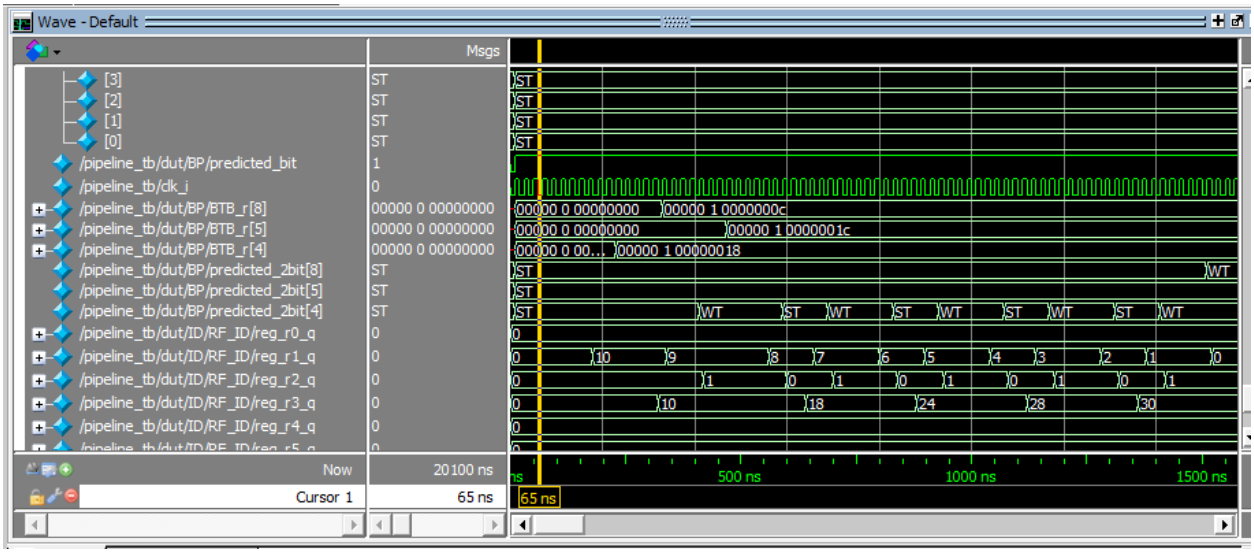
Case 3:

Ở trường hợp này, assembly sẽ thay đổi để có thể thực hiện nhiều lệnh nhảy giúp dễ kiểm chứng hơn.

Assembly:

```
addi x1, x0, 10

addi x8, x0, 1

add x3, x0, x0

_COMPARE:

and x2, x1, x8

beq x2, x0, _ADD_EVEN

jal x10, _DECREASE

_ADD_EVEN:

add x3, x1, x3

_DECREASE:

sub x1, x1, x8

bne x1, x0, _COMPARE

_EXIT:
```

Chương trình assembly trên sẽ tính tổng các số chẵn từ 1 đến 10 và lưu vào thanh ghi x3.



Trong waveform, BTB đã được cập nhật giá trị tag và predicted pc ở index cụ thể, tương tự như BTB, trạng thái của 2 bit prediction cũng được cập nhật khi nhảy sai với dự đoán.

2 bit prediction có 4 trạng thái:

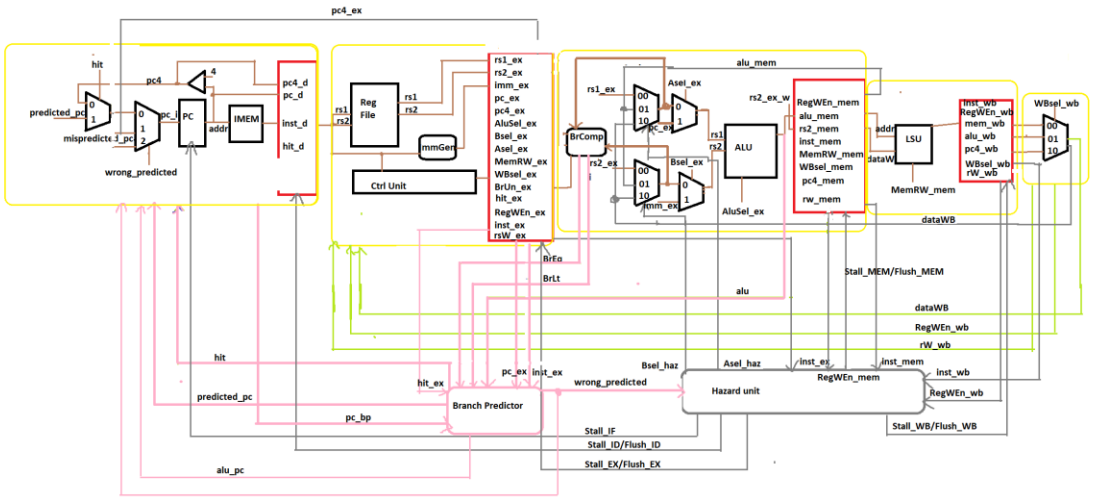
- ST (strongly taken)
- WT (weakly taken)
- SNT (strongly not taken)
- WNT (weakly not taken)

Chỉ khi lệnh nhảy được dự đoán sai 2 lần liên tiếp thì 2 bit prediction mới thay đổi từ taken sang not taken hoặc ngược lại. Trong waveform cho thấy lệnh nhảy beq x2, x0, _ADD_EVEN sẽ taken và not taken liên tiếp nhau nên 2 bit prediction chỉ thay đổi từ ST sang WT và ngược lại.

4. Advanced Design

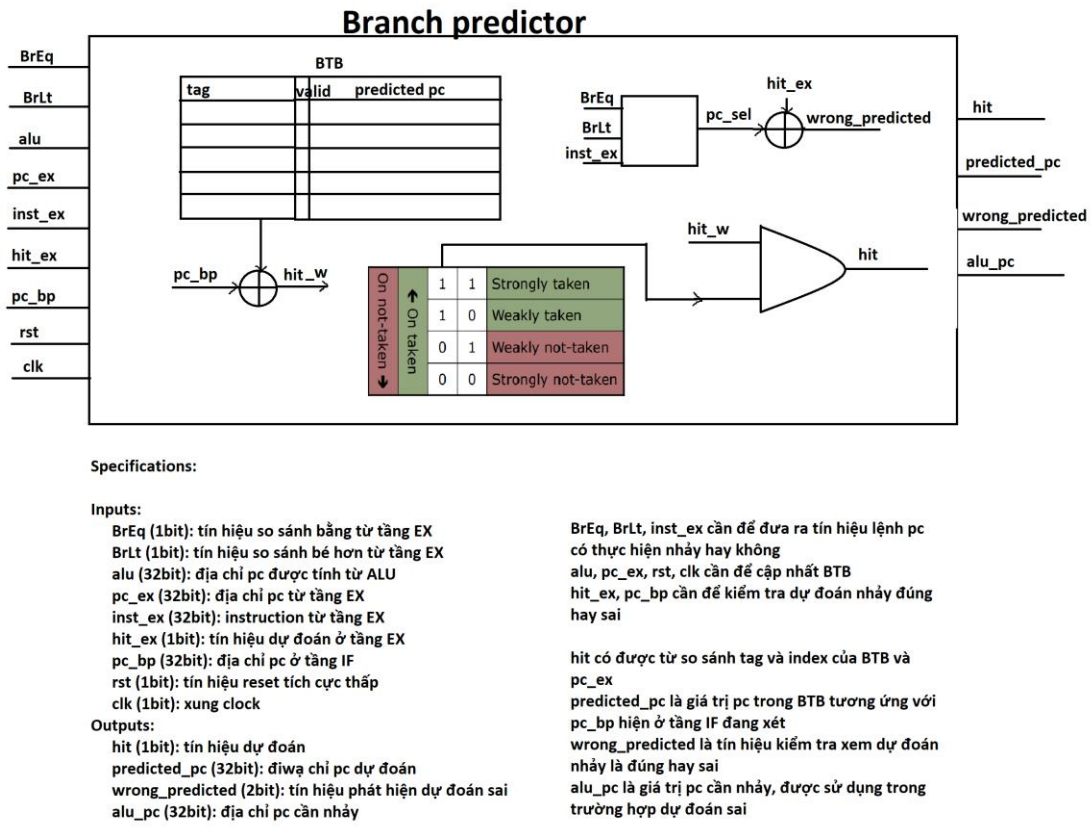
4.1. Two-bit prediction

Ở model này được kế thừa hoàn toàn từ always-taken model, riêng phần dự đoán bit được cải thiện bằng kỹ thuật two-bit prediction. Kỹ thuật chính ở model này là two-bit dynamic branch prediction so với static branch prediction ở always-taken model.



Hình 3: Block diagram của always-taken model

Block diagram ở model này hoàn toàn giống always-taken model, riêng branch predictor có thay đổi.



Specifications:

Inputs:

- **BrEq (1bit): tín hiệu so sánh bằng từ tầng EX**
- **BrLt (1bit): tín hiệu so sánh bé hơn từ tầng EX**
- **alu (32bit): địa chỉ pc được tính từ ALU**
- **pc_ex (32bit): địa chỉ pc từ tầng EX**
- **inst_ex (32bit): instruction từ tầng EX**
- **hit_ex (1bit): tín hiệu dự đoán ở tầng EX**
- **pc_bp (32bit): địa chỉ pc ở tầng IF**
- **rst (1bit): tín hiệu reset tích cực thấp**
- **clk (1bit): xung clock**

Outputs:

- **hit (1bit): tín hiệu dự đoán**
- **predicted_pc (32bit): điwạ chỉ pc dự đoán**
- **wrong_predicted (2bit): tín hiệu phát hiện dự đoán sai**
- **alu_pc (32bit): địa chỉ pc cần nhảy**

BrEq, BrLt, inst_ex cần để đưa ra tín hiệu lệnh pc có thực hiện nhảy hay không
alu, pc_ex, rst, clk cần để cập nhất BTB
hit_ex, pc_bp cần để kiểm tra dự đoán nhảy đúng hay sai

hit có được từ so sánh tag và index của BTB và pc_ex
predicted_pc là giá trị pc trong BTB tương ứng với pc_bp hiện ở tầng IF đang xét
wrong_predicted là tín hiệu kiểm tra xem dự đoán nhảy là đúng hay sai
alu_pc là giá trị pc cần nhảy, được sử dụng trong trường hợp dự đoán sai

Hình 5: Block diagram của branch predictor trong two-bit prediction model

Tính toán IPC:

Giả sử :

Xác suất gặp load data hazard là $P_d = 5\%$

Số lượng câu lệnh cần delay là $\Delta_d = 1$

Số lượng lệnh rẽ nhánh là $P_{br} = 20\%$

Xác suất nhảy khi gặp lệnh rẽ nhánh là $P_{mís} = 10\%$

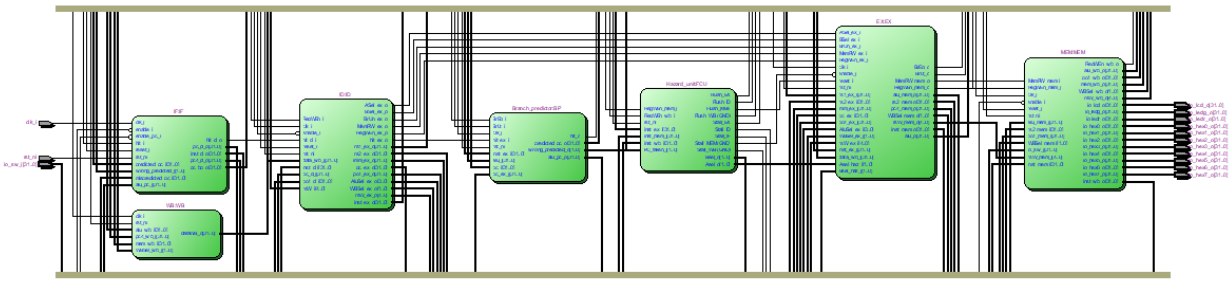
Số câu lệnh cần delay là $\Delta_{br} = 2$

Ta tính được:

$$IPC = \frac{N_{inst}}{N_{cycle}} = \frac{1}{P_d\ \Delta_d + P_{br}P_{mis}\Delta_{br} + 1} = \frac{1}{5\% \times 1 + 20\% \times 10\% \times 2 + 1} = 91.7\%$$

5. Evaluation

Sau khi thiết kế hoàn chỉnh, mô phỏng được RTL của mạch như hình:



❖ Ứng dụng dùng để đánh giá hiệu năng của từng model (sử dụng IPC):

Mô tả:

Input: 16 bit SW[15:0]

Output: 8 HEX 7 segment led.

Hiện 16 bit input đã nhập ra 8 led 7 đoạn dưới dạng số thập phân.

Code assembly:

```
# Application: input 16 bit from switches, convert input to decimal and display on 7 segment led

# 7segment common anode code
# 0 : 0x40
# 1 : 0x79
# 2 : 0x24
# 3 : 0x30
# 4 : 0x19
# 5 : 0x12
# 6 : 0x02
# 7 : 0x78
# 8 : 0x00
# 9 : 0x10


addi x1, x0, 1
addi x2, x0, 2
addi x3, x0, 3
```

```
addi x4, x0, 4
addi x5, x0, 5
addi x6, x0, 6
addi x7, x0, 7
addi x8, x0, 8
addi x9, x0, 9
addi x10, x0, 10

start:
lw x13, 320(x0) # read value from switches
#addi x13, x0, 765
sw x13, 264(x0)



# convert bin to bcd, x13 store value of bin number
#hex0
addi x19, x13, 0 #hex0
jal x11, mod10
jal x11, convert_code_7seg
sw x21, 256(x0)
addi x16, x13, 0
jal x11, div10
#hex1
addi x19, x13, 0    #hex1
jal x11, mod10
jal x11, convert_code_7seg
sw x21, 257(x0)
addi x16, x13, 0
jal x11, div10
#hex2
addi x19, x13, 0 #hex2
jal x11, mod10
jal x11, convert_code_7seg
sw x21, 258(x0)
addi x16, x13, 0
jal x11, div10
#hex3
addi x19, x13, 0 #hex3
jal x11, mod10
jal x11, convert_code_7seg
```

```
sw x21, 259(x0)
addi x16, x13, 0
jal x11, div10
addi x16, x13, 0
#hex4
addi x19, x13, 0    #hex3
jal x11, mod10
jal x11, convert_code_7seg
sw x21, 260(x0)

addi x30, x0, 0x40
sw x30, 261(x0)
sw x30, 262(x0)
sw x30, 263(x0)

jal x12, start
jal x12, exit

########################

# divide 10, x13=x13/10
div10:
addi x14, x0, 0 # divide 10
srli x15, x13, 1
add x14, x14, x15
srli x15, x13, 2
add x14, x14, x15
srli x15, x14, 4
add x14, x14, x15
srli x15, x14, 8
add x14, x14, x15
srli x15, x14, 16
add x14, x14, x15
srli x14, x14, 3
slli x15, x14, 2
add x15, x15, x14
slli x15, x15, 1
sub x13, x13, x15
blt x13, x10, temp_1
add x13, x14, x1
```

```
jal x12, out_div10
temp_1:
add x13, x14, x0
out_div10:
jalr x12, x11, 0
##########################3

# modulus 10, x18 = x19 % 10
mod10:
addi x18, x0, 0 # modulus 10

andi x20, x19, 0x0F # the first 4 bit
add x18, x18, x20

srli x19, x19, 4
andi x20, x19, 0x0F # the second 4 bit
add x18, x18, x20
andi x21, x20, 1
bne x21, x1, skip1
addi x18, x18, 5

skip1:
srli x19, x19, 4
andi x20, x19, 0x0F # the third 4 bit
add x18, x18, x20
andi x21, x20, 1
bne x21, x1, skip2
addi x18, x18, 5

skip2:
srli x19, x19, 4
andi x20, x19, 0x0F # the forth 4 bit
add x18, x18, x20
andi x21, x20, 1
bne x21, x1, skip3
addi x18, x18, 5

skip3:
blt x18, x10, out_mod10
addi x18, x18, -10
```

```
jal x12, skip3

out_mod10:
jalr x12, x11, 0
####################33


# x18 store value that need to convert, x21 return code
convert_code_7seg:
beq x18, x0, no0 #convert_code_7seg:
beq x18, x1, no1
beq x18, x2, no2
beq x18, x3, no3
beq x18, x4, no4
beq x18, x5, no5
beq x18, x6, no6
beq x18, x7, no7
beq x18, x8, no8
beq x18, x9, no9
jal x12, skip

no0:
addi x21, x0, 0x40
jal x12, skip

no1:
addi x21, x0, 0x79
jal x12, skip

no2:
addi x21, x0, 0x24
jal x12, skip

no3:
addi x21, x0, 0x30
jal x12, skip

no4:
addi x21, x0, 0x19
jal x12, skip
```

```
no5:
addi x21, x0, 0x12
jal x12, skip

no6:
addi x21, x0, 0x02
jal x12, skip

no7:
addi x21, x0, 0x78
jal x12, skip

no8:
addi x21, x0, 0x00
jal x12, skip

no9:
addi x21, x0, 0x10
jal x12, skip

skip:
jalr x12, x11, 0
################################
nop
nop
exit:
```

Từ ứng dụng trên, thống kế được ở mỗi input nhập vào có:

Hazard trong chương trình chính:

- Data hazard: 2, trong đó có 1 load data hazard.
- Control hazard: 15, trong đó 5 lần gọi chương trình mod10, 5 lần gọi chương trình convert_code_7seg, 4 lần gọi chương trình div10, 1 lần jump.

Hazard trong chương trình mod10:

- Data hazard (11)
- Forward conditional branch (4)

- Backward conditional branch (1)
- Jump (1)

Hazard trong chương trình convert_code_7seg:

- Forward conditional branch (10)
- Jump (2)

Hazard trong chương trình div10:

- data hazard (19)
- Forward conditional branch (1)
- Jump (1)

Số câu lệnh thực hiện chưa tính các lệnh trong chương trình con: $N = 46$

Số câu lệnh thực hiện trong chương trình con div10: $N_{div10} = 20$

Số câu lệnh thực hiện trong chương trình con mod10: $N_{mod10} = 25$

Số câu lệnh thực hiện trong chương trình con convert_code_7seg: $N_{conv} = 14$

Xác suất nhảy của câu lệnh forward conditional branch là 50%

Xác suất nhảy của câu lệnh backward conditional branch là 90%

Xác suất nhảy của câu lệnh jumb là 100%

Ở always-taken model, xác suất nhảy của lệnh branch là 30%

Ở 2-bit prediction branch, xác suất dự đoán sai của lệnh branch là 10%

Từ thống kê trên, tính IPC cho từng model trong ứng dụng trên, các giá trị Ninst và Ncycle được lấy tương đối vì tùy vào giá trị input mà Ninst và Ncycle có thể thay đổi:

- Non-forwarding model (đối với mỗi loại hazard có 2 cycle penalty):

$$IPC = \frac{N_{inst}}{N_{cycle}} = \frac{N+4*N_{div10}+5*N_{mod10}+5*N_{conv}}{(N+2*2+15*2)+4*(N_{div10}+20*2+1*2*50\%)+5*(N_{mod10}+12*2+4*2*50\%+1*1*90\%)+5*(N_{conv}+2*2+10*2*50\%)} = 43.8\%$$

- Forwarding model (1 cycle penalty cho load data hazard, 2 cycle penalty cho control hazard):

$$IPC = \frac{N_{inst}}{N_{cycle}} = \frac{N+4*N_{div10}+5*N_{mod10}+5*N_{conv}}{(N+2*1+15*2)+4*(N_{div10}+1*2+1*2*50\%)+5*(N_{mod10}+1*2+4*2*50\%+1*1*90\%)+5*(N_{conv}+2*2+10*2*50\%)} = 68.4\%$$

- Always-taken model (1 cycle penalty cho load data hazard, 2 cycle penalty cho control hazard):

$$IPC = \frac{N_{inst}}{N_{cycle}}$$
$$= \frac{N + 4 * N_{div10} + 5 * N_{mod10} + 5 * N_{conv}}{(N + 2 * 1) + 4 * (N_{div10} + 2 * 1 * 50\%) + 5 * (N_{mod10} + 4 * 2 * 50\%) + 5 * (N_{conv} + 10 * 2 * 50\%)}$$
$$= \frac{46 + 4 * 20 + 5 * 25 + 5 * 14}{(46 + 2 * 1) + 4 * (20 + 2 * 1 * 50\%) + 5 * (25 + 4 * 2 * 50\%) + 5 * (14 + 10 * 2 * 50\%)}$$
$$= 80.9\%$$

- Two-bit prediction model (1 cycle penalty cho load data hazard, 2 cycle penalty cho control hazard):

$$IPC = \frac{N_{inst}}{N_{cycle}}$$
$$= \frac{N + 4 * N_{div10} + 5 * N_{mod10} + 5 * N_{conv}}{(N + 2 * 1) + 4 * (N_{div10} + 2 * 1 * 10\%) + 5 * (N_{mod10} + 4 * 2 * 10\%) + 5 * (N_{conv} + 10 * 2 * 10\%)}$$
$$= \frac{46 + 4 * 20 + 5 * 25 + 5 * 14}{(46 + 2 * 1) + 4 * (20 + 2 * 1 * 10\%) + 5 * (25 + 4 * 2 * 10\%) + 5 * (14 + 10 * 2 * 10\%)}$$
$$= 95\%$$

Từ kết quả IPC trên và những lý thuyết về thiết kế ở trên, 4 models được thiết kế ở trên đạt được hiệu năng theo thứ tự như sau: 2-bit prediction > always-taken > forwarding > non-forwarding.

Hiệu năng được đánh giá qua IPC và có thể cải thiện bằng cách thiết kế khối branch predictor với những kỹ thuật phức tạp hơn như G-share hoặc kết hợp giữa global và local history predictor.

6. Conclusion

Piped line processor được thiết kế hoàn chỉnh với 4 model, mỗi model phía sau là phiên bản cải thiện của phiên bản trước. Một số ứng dụng cơ bản được viết bằng hợp ngữ và biên dịch ra mã hex, sau đó cho cpu chạy thử và thực hiện đúng chức năng. Chi tiết source code thiết kế, testbench và ứng dụng được đăng trên github:

https://github.com/Stork1323/Risc-V-Pipelined-RV32I-.git