TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN ĐIỆN TỬ - VIỄN THÔNG

BÁO CÁO BÀI TẬP LỚN
KIẾN TRÚC MÁY TÍNH
Đề tài: Bộ xử lí MIPS 32 bits Pipeline

GVHD: TS Tạ Thị Kim Huệ

Sinh viên thực hiện:

Nguyễn Minh Hiếu (20151336) – Điện tử 03 K60

Hà Nội, 12/2019

LỜI NÓI ĐẦU

Trong học phần Kiến trúc máy tính, để phục vụ cho bài tập lớn em chọn đề tài "Bộ xử lí MIPS 32 bits pipeline" để thực hiện. Đề tài này tuy hơi khó nhưng khá hay, vận dụng được nhiều kiến thức đã học để làm. Ngoài việc thiết kế, viết code Verilog, em còn triển khai mạch trên kit FPGA DE2 của Altera.

MỤC LỤC

Contents

DANH MỤC HÌNH VẼ

DANH MỤC BẢNG BIỂU

CHƯƠNG 1. GIỚI THIỆU VÀ MỤC TIÊU THIẾT KẾ BỘ XỬ LÍ MIPS 32 BITS PIPELINE

1.1 Giới thiệu bộ xử lí MIPS 32 bits Pipeline

MIPS (Microprocessor without Interlocked Pipeline Stage) là một kiến trúc vi xử lý được phát triển bởi hãng MIPS Technologies và là kiến trúc chiếm đến 1/3 số lượng chip sản xuất trên nền kiến trúc RISC.

Bộ xử lí MIPS đầu tiên được nghiên cứu vào năm 1981 với mục đích cơ bản là nhằm tăng đột xuất hiệu năng thông qua sử dụng một đường ống lệnh (pileline instructions). Thiết kế theo pipeline làm giảm đáng kể thời gian rảnh rỗi của CPU khi thực hiện liên tiếp các câu lệnh.

Khó khăn trong quá trình tìm hiểu thiết kế: theo phương pháp đường ống lệnh nó yêu cầu một khóa đồng bộ (interlocks) được cài đặt để chắc chắn rằng các câu lệnh chiếm nhiều chu kì đồng hồ để thực hiện sẽ dừng đường ống lại để nạp nhiều dữ liệu hơn. Những khóa đồng bộ này cần một thời gian lớn để cài đặt và được cho là rào cản chính trong việc tăng tốc độ xử lí trong tương lai.

Yêu cầu đặt ra trong quá trình thiết kế: yêu cầu tất cả các câu lệnh phải được hoàn thành trong 1 chu kì xung nhịp nhờ thế lạo bỏ được sự cần thiết của khóa đồng bộ. Thiết kế này đã loại bỏ được một số câu lệnh hữu dụng, đáng kể nhất là các lệnh nhân, chia yêu cầu nhiều bước nhưng nó cho thấy hiệu suất tổng thể của hệ thống tăng lên rõ rệt vì các vi xử lý có thể chạy ở xung nhịp lớn hơn rất nhiều.

1.2 Nguyên tắc thiết kế kiến trúc MIPS Pipeline

Kiến trúc MIPS pipeline được thiết kế dựa trên các nguyên tắc sau:

- Tính đơn giản quan trọng hơn tính quy tắc (Simplicity favors regularity)
    - Chỉ thị kích thước cố định (32bit)
    - Ít định dạng chỉ thị (3 loại định dạng)
    - Mã lệnh ở vị trí cố định (6 bit đầu)
- Nhỏ hơn thì nhanh hơn
    - Số chỉ thị giới hạn
    - Số thanh ghi giới hạn
    - Số chế độ địa chỉ giới hạn
- Tăng tốc trong các trường hợp thông dụng
    - Các toán hạng số học lấy từ thanh ghi (máy tính dựa trên cơ chế load-store)
    - Các chỉ thị có thể chứa toán hạng trực tiếp
- Thiết kế đòi hỏi sự thỏa hiệp: Ba loại chỉ thị định dạng.
- Nguyên tắc hoạt động của Pipeline
    - Chia nhỏ các lệnh thành các giai đoạn đường ống
    - Bắt đầu lệnh tiếp theo trước khi lệnh hiện tại kết thúc

1.3 Các thanh ghi trong MIPS

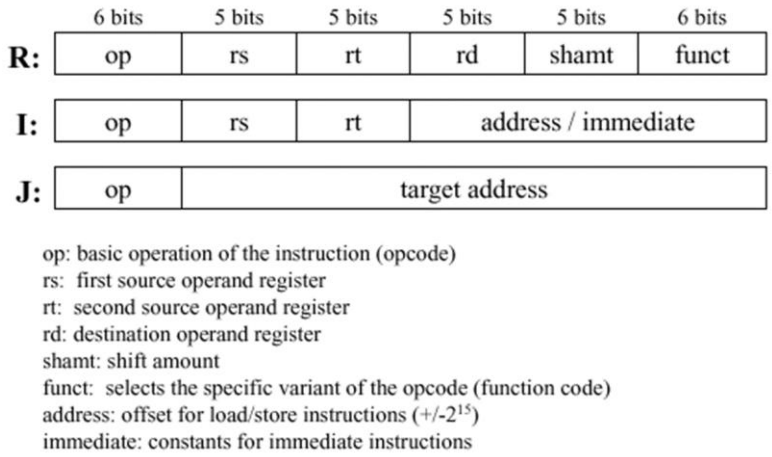
MIPS có 32 thanh ghi 32 bits. Tên gọi, số thanh ghi và chức năng của chúng được thể hiện trên bảng 1.1.

Bảng 1.1 Các thanh ghi trong MIPS

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

1.4 Kiến trúc tập lệnh MIPS

Kiến trúc tập lệnh MIPS, hay cách chuyển từ mã MIPS sang mã máy được thể hiện trên hình 1.1.



Hình 1.1 Kiến trúc tập lệnh MIPS

Trong MIPS có 3 loại lệnh:

- Lệnh loại R: Gồm các lệnh số học và logic, toán hạng là các thanh ghi
- Lệnh loại I: Gồm các lệnh số học và logic, lệnh dịch chuyển dữ liệu, lệnh rẽ nhánh có điều kiện, toán hạng đích và toán hạng nguồn 1 là thanh ghi, toán hạng nguồn 2 là địa chỉ tức thời (immediate)
- Lệnh loại J: Lệnh nhảy không điều kiện, toán hạng chỉ có địa chỉ nhảy đến 26 bits

1.5 Mục tiêu thiết kế

Mục đích của bản thiết kế, nhằm tạo ra một bộ xử lý MIPS, nhằm tăng đột xuất hiệu năng thông qua sử dụng đường ống lệnh (pipeline instructions). Thiết kế pipeline làm giảm đáng kể thời gian rảnh rỗi của CPU khi thực hiện liên tiếp các câu lệnh. Bộ xử lý này chỉ có khả năngthực hiện được một số lệnh cơ bản.

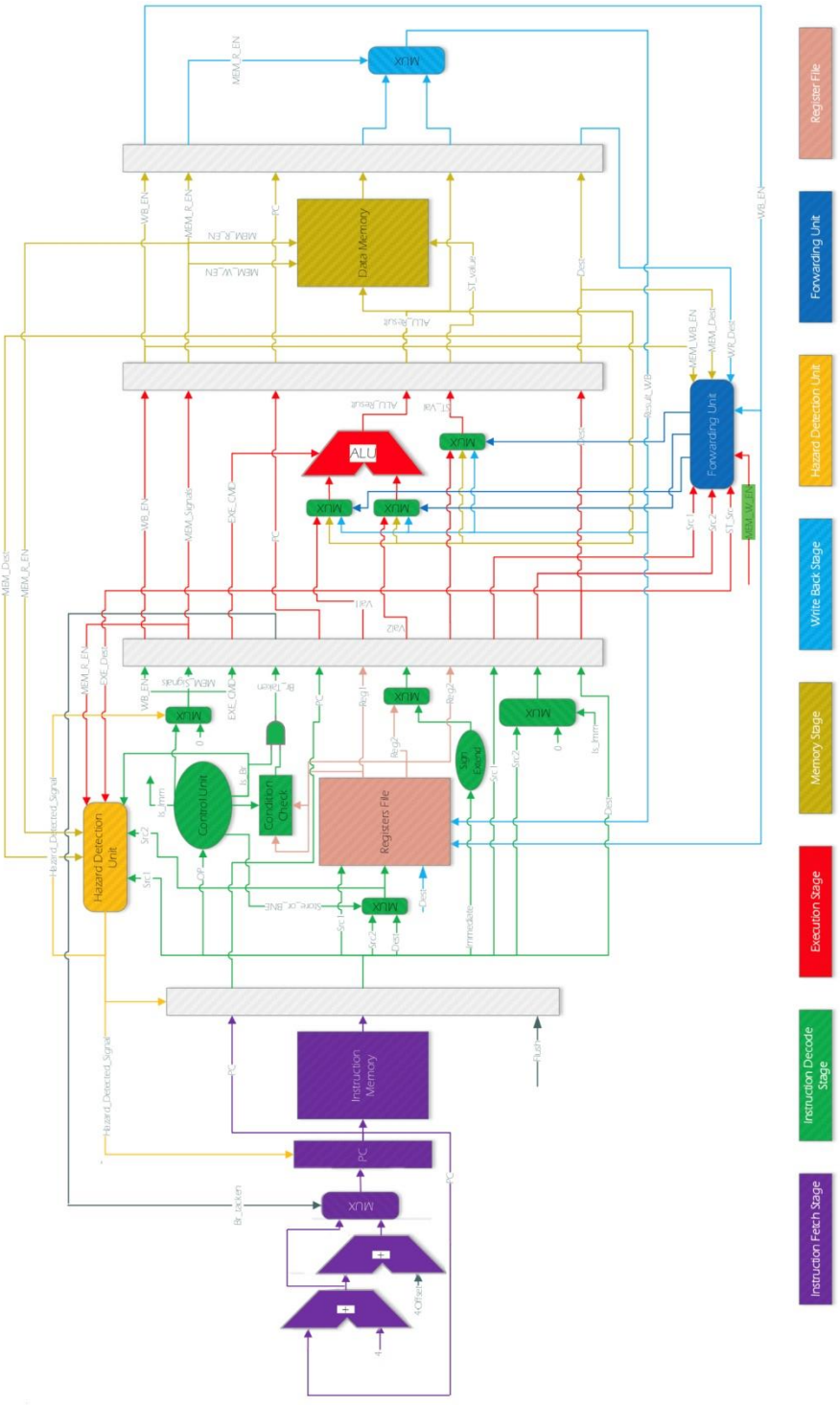
Kiến trúc tập lệnh này hỗ trợ thực hiện”

- Các phép toán số học: cộng, trừ, nhân...
- Truy cập bộ nhớ với 2 chỉ thị: lw, sw
- Lưu trữ, đọc và ghi byte dữ liệu
- Lệnh dịch, logic số học, nhảy (có điều kiện và không có điều kiện)

CHƯƠNG 2. THIẾT KẾ BỘ XỬ LÍ MIPS 32 BITS PIPELINE

2.1 Thiết kế tổng quan

Thiết kế tổng quan bộ xử lí MIPS 32 bits Pipeline được thể hiện trên hình 2.1.



Hình 2.1 Thiết kế tổng quan bộ xử lí MIPS 32 bits pipeline

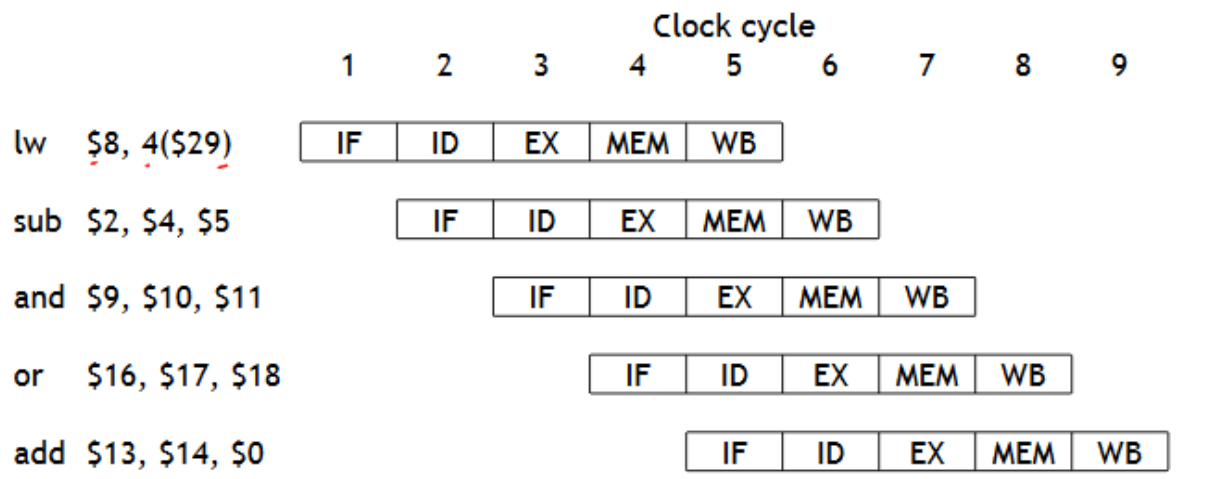
Khác với bộ xử lí đơn xung nhịp khi các lệnh đều được thực hiện xong trong một chu kì máy, bộ xử lí pipeline thực hiện 1 lệnh trong 5 stages, mỗi stage thực hiện trong 1 chu kì máy:

- Instruction fetch (IF): Nạp lệnh và cập nhật giá trị thanh ghi PC xác định địa chỉ lệnh tiếp theo
- Instruction decode (ID): Giải mã lệnh, xác định toán tử thực thi lệnh, xác định toán hạng bằng cách đọc tệp thanh ghi từ địa chỉ cho trong lệnh
- Excution (EX): Thực thi phép toán bằng ALU
- Memory (MEM): Đọc hoặc ghi dữ liệu trên bộ nhớ
- Write back (WB): Ghi dữ liệu vào tệp thanh ghi

Bộ xử lí pipeline cải thiện hiệu năng theo thông lượng. Câu lệnh sau không cần đợi câu lệnh trước hoàn tất mới bắt đầu thực hiện mà mỗi stage sẽ được thực hiện liên tiếp bởi các thanh ghi pipeline. Có tất cả 4 thanh ghi pipeline giữa 5 stages:

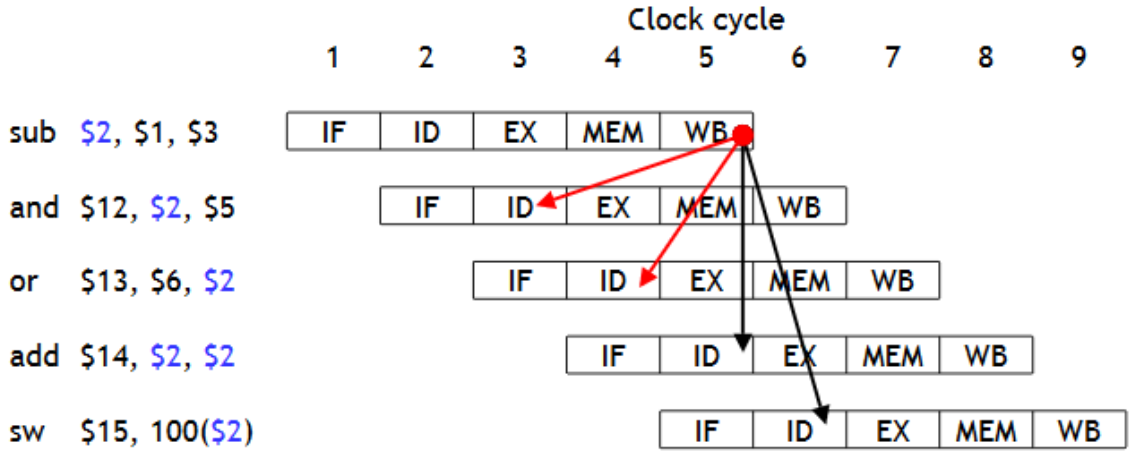
- Thanh ghi IF-ID: Ở giữa IF stage và ID stage
- Thanh ghi ID-EX: Ở giữa ID stage và EX stage
- Thanh ghi EX-MEM: Ở giữa EX stage và MEM stage
- Thanh ghi MEM-WB: Ở giữa MEM stage và WB stage

Nguyên lí thực thi của 1 chương trình thực thi bằng bộ xử lí pipeline được thể hiện trên hình 2.2 với 1 ví dụ chương trình MIPS



Hình 2.2 Nguyên lí thực thi chương trình bằng bộ xử lí pipeline

Đây là 1 ví dụ đơn giản (lí tưởng) thực thi chương trình bằng bộ xử lí pipeline. Mỗi lệnh cần 5 clock cycles để thực hiện. Trong 1 clock cycle, bộ xử lí thực hiện đồng thời các stages IF, ID, EX, MEM, WB của các lệnh liên tiếp. Đoạn code này là lí tưởng, không có sự phụ thuộc giữa các lệnh với nhau, do đó không xảy ra xung đột. Xét 1 chương trình có xung đột (hình 2.3).



Hình 2.3 Chương trình gây xung đột trong bộ xử lí pipeline

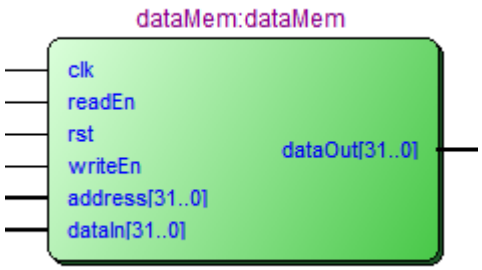
Các mũi tên màu đỏ chỉ ra xung đột dữ liệu, khi mà lệnh đầu tiên ở WB stage mới cập nhật giá trị thanh ghi $2, trong khi 2 lệnh sau cần truy cập tệp thanh ghi để lấy giá trị $2 ở ID stage. Đây chỉ là một loại xung đột pipeline, có tất cả 3 loại xung đột:

- Xung đột cấu trúc: Tài nguyên (bộ nhớ, tệp thanh ghi) được truy cập cùng lúc ở nhiều stage
- Xung đột dữ liệu: Phụ thuộc dữ liệu giữa các lệnh, lệnh sau cần đợi lệnh trước hoàn thành việc ghi dữ liệu
- Xung đột điều khiển: Xác định lệnh tiếp theo phụ thuộc vào lệnh trước

  Để giải quyết xung đột, có thể dùng 2 phương pháp:
- Phương pháp dừng và chờ: Chèn các chu kì đợi vào giữa các lệnh có xung đột
- Phương pháp chuyển tiếp dữ liệu: Chuyển dữ liệu đến stage cần sử dụng của lệnh sau ngay khi có kết quả từ lệnh trước

2.2 Bộ nhớ dữ liệu

Bộ nhớ dữ liệu sử dụng trong bài tập lớn này là bộ nhớ dung lượng 1 KB, gồm các ngăn nhớ 8 bits. Vì bộ xử lí là 32 bits nên mỗi lần truy cập bộ nhớ sẽ đọc hoặc ghi trên 4 ngăn nhớ liên tiếp với địa chỉ cơ sở là 0, 4, 8, …1000. Thiết kế bộ nhớ dữ liệu được thể hiện trên hình 2.4.



Hình 2.4 Bộ nhớ dữ liệu

Các đầu vào gồm:

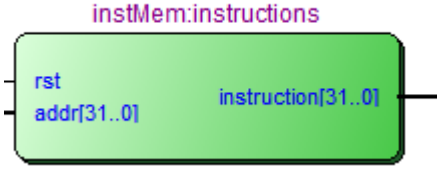
- clk, rst: Xung clock và tín hiệu reset các ô nhớ về 0
- readEn, writeEn: Tín hiệu cho phép đọc hoặc ghi bộ nhớ
- address (32 bits): Địa chỉ truy cập bộ nhớ
- dataIn (32 bits): Dữ liệu ghi vào bộ nhớ

Đầu ra chỉ có:

- dataOut (32 bits): Dữ liệu đọc ra từ bộ nhớ

2.3 Bộ nhớ lệnh

Bộ nhớ lệnh cũng tương tự bộ nhớ dữ liệu, có dung lượng 1 KB, gồm các ngăn nhớ 8 bits. Mỗi lần truy cập bộ nhớ lệnh để đọc lệnh 32 bits ra sẽ đọc trên 4 ngăn nhớ liên tiếp. Thiết kế bộ nhớ lệnh được thể hiện trên hình 2.5.



Hình 2.5 Bộ nhớ lệnh

Các đầu vào gồm:

- rst: Khởi tạo bộ nhớ lệnh
- addr (32 bits): Địa chỉ truy cập bộ nhớ lệnh

Đầu ra chỉ có:

- instruction (32 bits): Lệnh đọc ra

Các lệnh đã được viết sẵn trong code vào bộ nhớ lệnh, theo quy ước chuyển mã MIPS sang mã máy cho các lệnh loại R, I, J. Khi triển khai, bộ xử lí sẽ thực hiện lần lượt các lệnh bắt đầu từ địa chỉ 0.

2.4 Tệp thanh ghi

Tệp thanh ghi cũng là 1 kiểu bộ nhớ, với dung lượng 32 (32 thanh ghi trong MIPS), mỗi ngăn nhớ 32 bits (độ lớn 1 thanh ghi). Vì vậy, thiết kế tệp thanh ghi cũng tương tự bộ nhớ dữ liệu và bộ nhớ lệnh, được thể hiện trên hình 2.6.

Hình 2.6 Tệp thanh ghi

Các đầu vào gồm:

- clk, rst: Xung clock và tín hiệu reset giá trị các thanh ghi về 0
- writeEn: Tín hiệu cho phép ghi vào tệp thanh ghi
- src1, src2, dest (5 bits): Địa chỉ truy cập tệp thanh ghi (rs, rt, rd)
- writeVal (32 bits): Giá trị ghi vào tệp thanh ghi

Tệp thanh ghi gồm 2 đầu ra:

- reg1, reg2 (32 bits): Giá trị 2 thanh ghi đọc ra để làm 2 toán hạng đưa vào ALU

2.5 Thanh ghi PC

Thanh ghi PC 32 bits lưu địa chỉ lệnh cần truy cập vào bộ nhớ lệnh. Thiết kế thanh ghi PC được thể hiện trên hình 2.7.

Hình 2.7 Thanh ghi PC

Các đầu vào gồm:

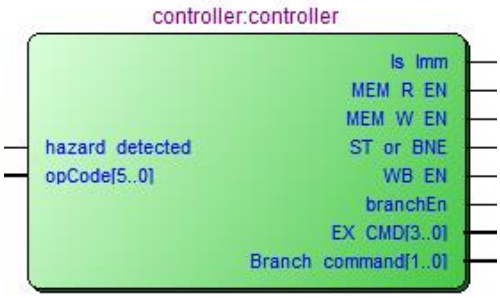
- clk, rst: Xung clock và tín hiệu reset địa chỉ lệnh tiếp theo là 0
- writeEn: Tín hiệu cho phép xác định địa chỉ lệnh tiếp theo
- regIn (32 bits): Xác định địa chỉ lệnh tiếp theo

Đầu ra chỉ có:

- regOut (32 bits): Địa chỉ lệnh cần truy cập vào bộ nhớ lệnh

2.6 Bộ điều khiển

Bộ điều khiển nhận 6 bits opCode từ mã máy và phát ra các tín hiệu điều khiển. Thiết kế bộ điều khiển được thể hiện trên hình 2.8



Hình 2.8 Bộ điều khiển

Các đầu vào gồm:

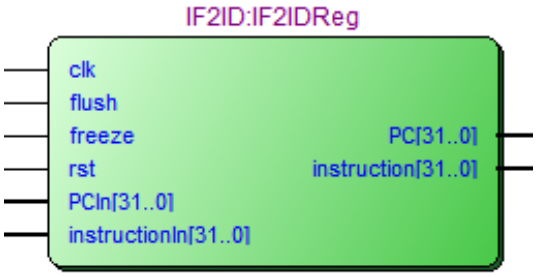
- Hazard_detected: Xác định có xung đột không
- opCode (6 bits): Mã máy xác định tín hiệu điều khiển]

Các đầu ra gồm:

- Is_Imm: Xác định có phải lệnh loại I không
- MEM_R_EN: Tín hiệu cho phép đọc memory
- MEM_W_EN: Tín hiệu cho phép ghi memory
- ST_or_BNE: Xác định lệnh là store hoặc BNE
- WB_EN: Tín hiệu cho phép ghi vào tệp thanh ghi
- branchEn: Tín hiệu xác định lệnh có phải rẽ nhánh không
- EX_CMD (4 bits): Xác định phép toán thực hiện trong ALU
- Branch_command (2 bits): Xác định loại lệnh rẽ nhánh

2.7 Thanh ghi IF-ID

Thanh ghi IF-ID 64 bits nằm ở giữa IF stage và ID stage, chứa 32 bits PC và 32 bits lệnh. Thiết kế thanh ghi IF-ID được thể hiện trên hình 2.9.



Hình 2.9 Thanh ghi IF-ID

Các đầu vào gồm:

- clk, rst: Xung clock và tín hiệu reset PC và lệnh đầu ra về 0
- flush: Tín hiệu xác đinh "vỡ đường ống" (khi thực hiện lệnh rẽ nhánh)
- freeze: Tín hiệu xác định có xung đột hay không
- PCIn, InstructionIn (32 bits): Giá trị PC và lệnh đưa vào từ IF Stage

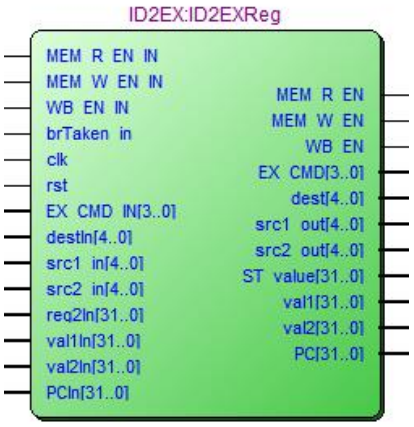
Các đầu ra gồm:

- PC (32 bits): Giá trị PC đưa vào thanh ghi EX-MEM
- instruction (32 bits): Địa chỉ lệnh đưa vào ID Stage

Nếu không có xung đột và lệnh không phải rẽ nhánh, các giá trị PC và instruction được lấy ra từ PCIn và instructionIn, ngược lại thì bằng 0.

2.8 Thanh ghi ID-EX

Thanh ghi ID-EX 151 bits nằm ở giữa ID Stage và EX Stage, chứa 8 bits điều khiển, 10 bits rt và rd, 5 bits (rd hoặc 0) địa chỉ thanh ghi dùng trong chuyển tiếp dữ liệu, 64 bits dữ liệu từ 2 thanh ghi đọc ra từ tệp thanh ghi, 32 bits mở rộng dấu và 32 bits PC. Thiết kế thanh ghi ID-EX được thể hiện trên hình 2.10.



Hình 2.10 Thanh ghi ID-EX

Các đầu vào gồm:

- MEM_R_EN_IN, MEM_W_EN_IN, WB_EN_IN, brTaken_in: Các tín hiệu điều khiển từ ID Stage
- EX_CMD_IN (4 bits): Từ ID Stage, dùng để điều khiển chọn toán hạng ALU
- destIN, src1_in, src2_in (5 bits): rt, rd và địa chỉ thanh ghi dùng trong chuyển tiếp dữ liệu, được đưa sang từ thanh ghi IF-ID
- reg2In, val1In, val2In, PCIn (32 bits): 64 bits dữ liệu từ 2 thanh ghi đọc ra từ tệp thanh ghi, 32 bits mở rộng dấu và 32 bits PC

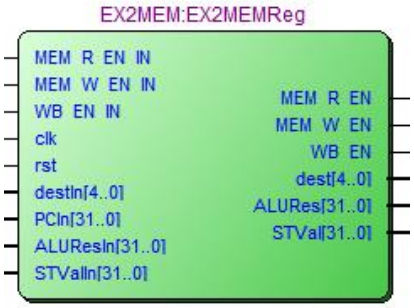
Các đầu ra gồm:

- MEM_R_EN, MEM_W_EN, WB_EN: Các tín hiệu điều khiển đưa đến thanh ghi EX-MEM
- brTaken: Tín hiệu điều khiển đưa vào bộ MUX trong IF Stage để chọn địa chỉ lệnh tiếp theo
- EX_CMD (4 bits): Đưa vào ALU trong EX Stage
- dest (5 bits): Đưa vào thanh ghi EX-MEM
- src1_out, src2_out: Đưa vào khối chuyển tiếp dữ liệu
- ST_value: Giá trị cần lưu vào bộ nhớ dữ liệu trong MEM Stage

- val1, val2: 2 toán hạng đưa vào ALU trong EX Stage
- PC (32 bits): Đưa vào thanh ghi EX-MEM

2.9 Thanh ghi EX-MEM

Thanh ghi EX-MEM 104 bits nằm ở giữa EX Stage và MEM Stage, chứa 3 bits điều khiển, 5 bits rt hoặc rd, 32 bits PC, 32 bits kết quả từ ALU, 32 bits giá trị cần lưu vào bộ nhớ dữ liệu. Thiết kế thanh ghi EX-MEM được thể hiện trên hình 2.11.



Hình 2.11 Thanh ghi EX-MEM

Các đầu vào gồm

- MEM_R_EN_IN, MEM_W_EN_IN, WB_EN_IN: Các tín hiệu điều khiển từ thanh ghi ID-EX
- destIn (5 bits): rt hoặc rd, từ thanh ghi ID-EX
- PCIn (32 bits): Từ thanh ghi ID-EX
- ALURes, STVal (32 bits): Kết quả ALU và giá trị cần lưu vào bộ nhớ dữ liệu từ EX Stage

Các đầu ra gồm:

- MEM_R_EN: Đưa vào memory trong MEM Stage và thanh ghi MEM-WB
- MEM_W_EN: Đưa vào bộ nhớ dữ liệu trong MEM Stage
- WB_EN: Đưa vào thanh ghi MEM-WB
- dest (5 bits): Đưa vào thanh ghi MEM-WB
- PC (32 bits): Đưa vào thanh ghi MEM-WB
- ALURes (32 bits): Đưa vào memory trong MEM Stage và thanh ghi MEM-WB
- STVal (32 bits): Đưa vào memory trong MEM Stage

2.10 Thanh ghi MEM-WB

Thanh ghi MEM-WB 81 bits, nằm ở giữa MEM Stage và WB Stage, chứa 2 bits điều khiển, 5 bits rt hoặc rd, 32 bits kết quả từ ALU, 32 bits dữ liệu đọc từ memory. Thiết kế thanh ghi MEM-WB được thể hiện trên hình 2.12.

Hình 2.12 Thanh ghi MEM-WB

Các đầu vào gồm:

- WB_EN_IN, MEM_R_EN_IN: Các tín hiệu điều khiển từ thanh ghi EX-MEM
- destIn (5 bits): rt hoặc rd, từthanh ghi EX-MEM
- ALUResIn (32 bits): Kết quả thực hiện ALU từ thanh ghi EX-MEM
- memReadValIn (32 bits): Giá trị đọc memory từ MEM Stage

Các đầu ra gồm:

- MEM_R_EN: Lựa chọn ALURes haoc memReadVal lam gia tri write back
- WB_EN: Đưa vào tệp thanh ghi và khối chuyển tiếp dữ liệu
- dest (5 bits): Địa chỉ write back, đưa vào tệp thanh ghi
- ALURes, memReadVal (32 bits): Đưa vào tệp thanh ghi

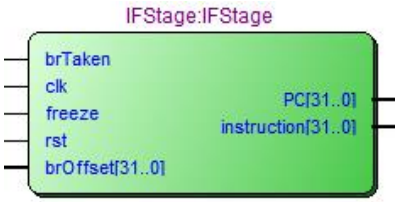
2.11 IF Stage

Trong IF Stage, bộ xử lí lấy ra lệnh cần thực hiện trong bộ nhớ lệnh và xác định địa chỉ lệnh tiếp theo. Địa chỉ lệnh tiếp theo được xác định như sau:

$$\mathrm{PC_{next}} = \mathrm{PC_{current}} + 4, \text{ nếu lệnh tuần tự}$$

$$\mathrm{PC_{next}} = \mathrm{PC_{current}} + 4 + 4 * \mathrm{offset}, \text{ nếu lệnh rẽ nhánh}$$

Thiết kế IF Stage được thể hiện trên hình 2.13.



Hình 2.13 IF Stage

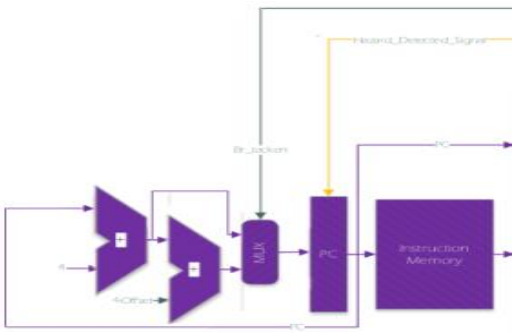
Các đầu vào gồm:

- brTaken: Xác định lệnh có phải lệnh rẽ nhánh không
- freeze: Xác định có xung đột không
- brOffset (32 bits): Xác định địa chỉ lệnh tiếp theo trong lệnh rẽ nhánh

Các đầu ra gồm:

- PC (32 bits): Địa chỉ lệnh cần truy cập vào bộ nhớ lệnh
- instruction (32 bits): Lệnh đọc ra từ bộ nhớ lệnh

Thiết kế chi tiết IF Stage được thể hiện trên hình 2.14.

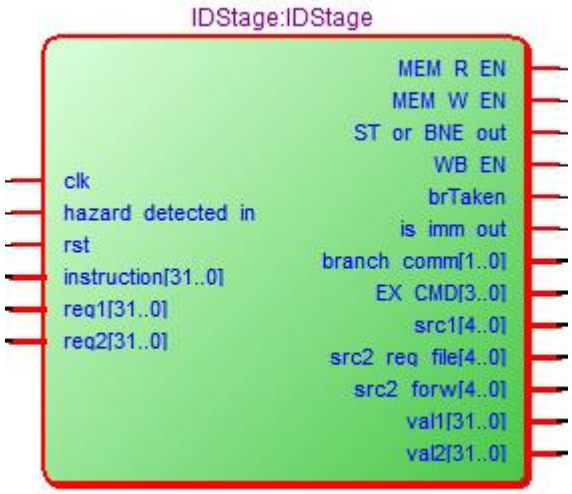


Hình 2.14 Thiết kế chi tiết IF Stage

2 bộ cộng 32 bits dùng để xác định $\mathrm{PC_{current}} + 4$ và $\mathrm{PC_{current}} + 4 + 4 * \mathrm{offset}$, bộ MUX 2 to 1 dùng để xác định địa chỉ lệnh tiếp theo với tín hiệu chọn brTaken.

2.12 ID Stage

Trong ID Stage, bộ điều khiển phát ra các tín hiệu điều khiển, bộ xử lí truy cập tệp thanh ghi để lấy ra giá trị 2 thanh ghi làm toán hạng ALU. Thiết kế ID Stage được thể hiện trên hình 2.15.



Hình 2.15 ID Stage

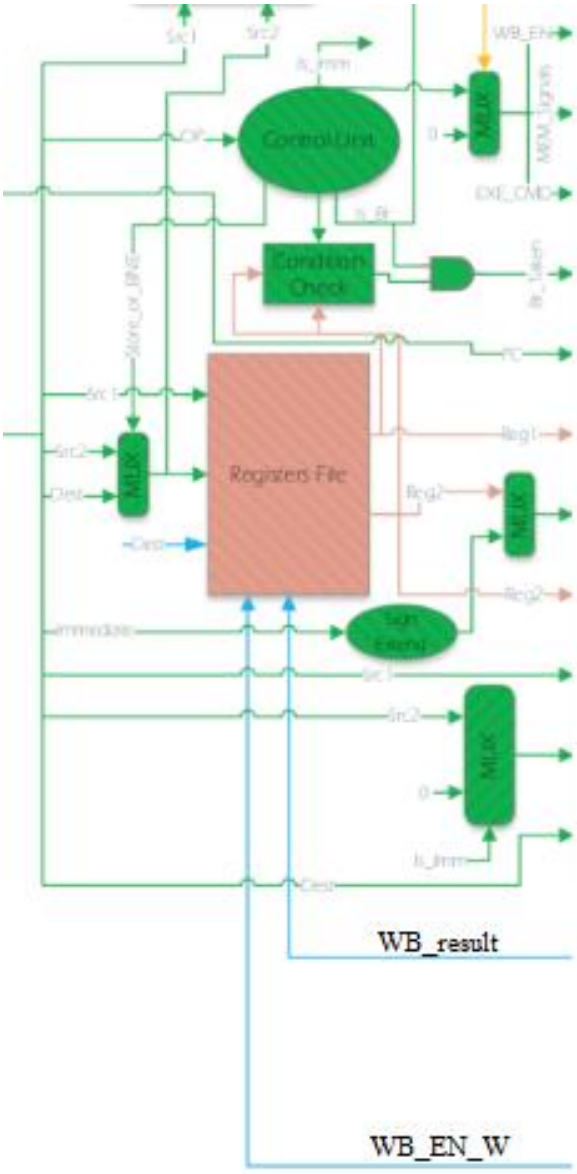
Các đầu vào gồm:

- hazard_detected_in: Từ khối phát hiện xung đột, đưa vào bộ điều khiển
- instruction (32 bits): Từ thanh ghi IF-ID
- reg1, reg2 (32 bits): 2 giá trị đọc ra từ tệp thanh ghi, dùng để kiểm tra điều kiện trong lệnh rẽ nhánh

Các đầu ra gồm:

- MEM_R_EN, MEM_W_EN, ST_or_BNE_out, WB_EN, brTaken, is_imm_out, branch_comm, EX_CMD: Các tín hiệu điều khiển phát ra từ bộ điều khiển
- src1 (5 bits): rt, đưa vào tệp thanh ghi và khối phát hiện xung đột
- src2_reg_file (5 bits): rs hoặc rd, đưa vào tệp thanh ghi
- val1, val2 (32 bits): 2 giá trị đọc ra từ tệp thanh ghi hoặc có 1 giá trị là từ bộ mở rộng dấu, đưa vào thanh ghi ID-EX dùng làm toán hạng đưa vào ALU

  Thiết kế chi tiết ID Stage được thể hiện trên hình 2.16.

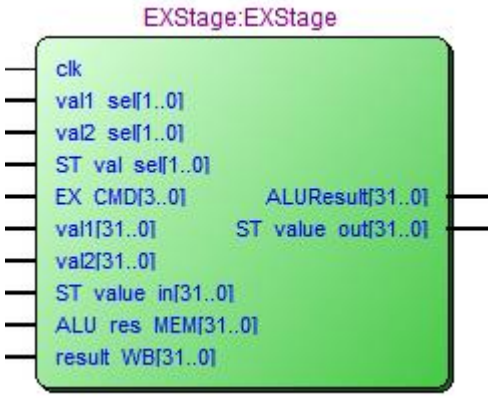


Hình 2.16 Thiết kế chi tiết ID Stage

Khối kiểm tra điều kiện sử dụng reg1 và reg2 để kiểm tra có thỏa mãn điều kiện trong lệnh rẽ nhánh không. Bộ mở rộng dấu mở rộng số 16 bits thành 32 bits dùng trong lệnh loại I. Các bộ MUX lựa chọn tín hiệu phù hợp tùy theo trường hợp lệnh loại I, lệnh rẽ nhánh, phát hiện xung đột. Các đường tín hiệu màu xanh da trời thuộc về WB Stage (sẽ trình bày ở phần WB Stage).

2.13 EX Stage

Trong EX Stage, ALU thực hiện phép toán, kết quả có thể dùng để ghi lại vào tệp thanh ghi hoặc xác định địa chỉ truy cập memory. Thiết kế EX Stage được thể hiện trên hình 2.17.



Hình 2.17 EX Stage

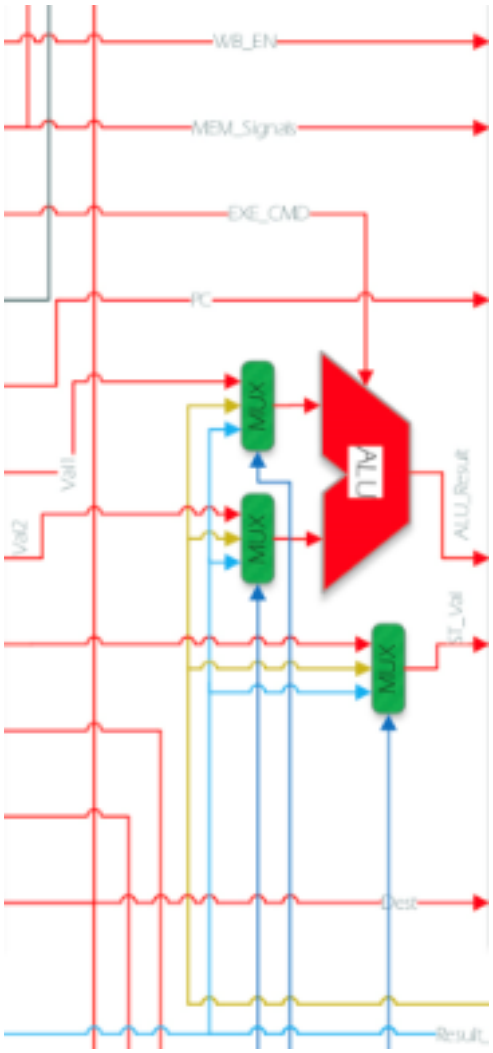
Các đầu vào gồm:

- val1_sel, val2_sel (2 bits): Tín hiệu chọn dùng cho 2 bộ MUX lựa chọn 2 toán hạng vào ALU khi có chuyển tiếp dữ liệu
- ST_val_sel (2 bits): Tín hiệu chọn cho bộ MUX lựa chọn giá trị lưu ghi vào memory
- val1, val2 (32 bits): 2 giá trị từ thanh ghi ID-EX, được đưa vào 2 bộ MUX để làm 2 toán hạng ALU
- ST_value_in (32 bits): Đưa vào bộ MUX lựa chọn giá trị ghi vào memory
- ALU_res_MEM, result_WB(32 bits): Kết quả tính toán ALU và giá trị write back được đưa ngược lại 3 bộ MUX kể trên dùng trong chuyển tiếp dữ liệu

Các đầu ra gồm:

- ALUResult, ST_value_out (32 bits): Đưa vào thanh ghi EX-MEM

Thiết kế chi tiết EX Stage được thể hiện trên hình 2.18.

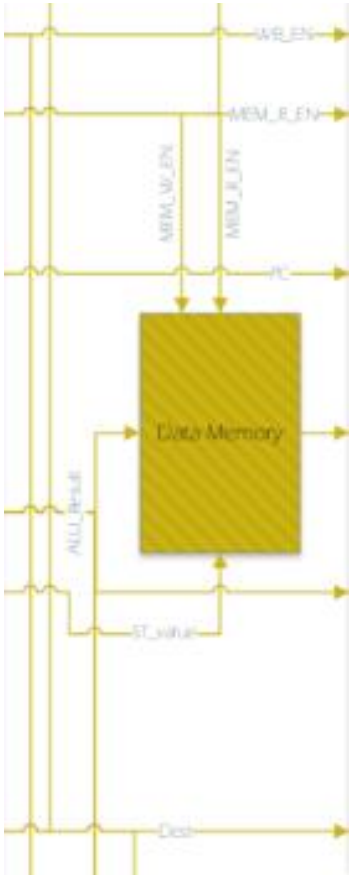


Hình 2.18 Thiết kế chi tiết EX Stage

Đường tín hiệu màu vàng là kết quả tính toán ALU, được đưa vào từ thanh ghi EX-MEM, đường tín hiệu màu xanh da trời là giá trị write back, được đưa vào từ WB Stage. Các tín hiệu chọn cho 3 bộ MUX được đưa vào từ khối chuyển tiếp dữ liệu, việc xác định các tín hiệu này sẽ được trình bày ở khối chuyển tiếp dữ liệu.

2.14 MEM Stage

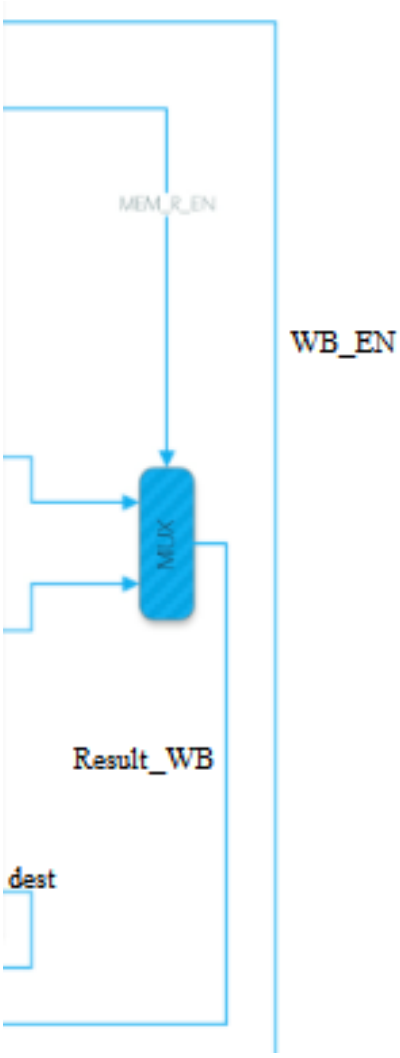
Trong Stage này, bộ xử lí truy cập vào bộ nhớ dữ liệu để tiến hàng đọc hoặc ghi, kết quả đọc được đưa vào thanh ghi MEM-WB. Thiết kế MEM Stage chỉ gồm có memory (mục 2.2), thiết kế chi tiết MEM Stage được thể hiện trên hình 2.19.



Hình 2.19 Thiết kế chi tiết MEM Stage

2.15 WB Stage

Trong WB Stage, bộ xử lí tiến hành ghi giá trị (đọc từ memory hoặc kết quả ALU) vào tệp thanh ghi. Thiết kế chi tiết WB Stage được thể hiện trên hình 2.20.



Hình 2.20 Thiết kế chi tiết WB Stage

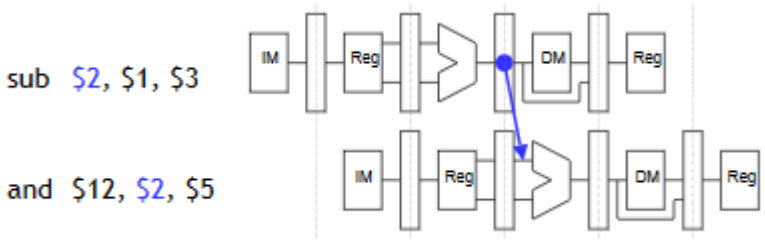
Bộ MUX 2 to 1 dùng để xác định giá trị sẽ write back là đọc từ memory hay kết quả ALU. Các đường tín hiệu dest, Result_WB và WB_EN được đưa vào tệp thanh ghi (hình 2.16 ở mục 2.12).

2.16 Khối phát hiện xung đột và khối chuyển tiếp dữ liệu

2.16.1 Cơ chế phát hiện xung đột và giải quyết xung đột bằng phương pháp chuyển tiếp dữ liệu

- Xung đột EX/MEM xảy ra giữa lệnh đang ở EX Stage và lệnh trước đó nếu:
- Lệnh trước đó sẽ ghi vào tệp thanh ghi
- Địa chỉ thanh ghi sẽ ghi vào là 1 trong 2 toán hạng vào ALU của lệnh đang thực thi ở EX Stage

Xét một ví dụ xung đột EX/MEM (hình 2.21)

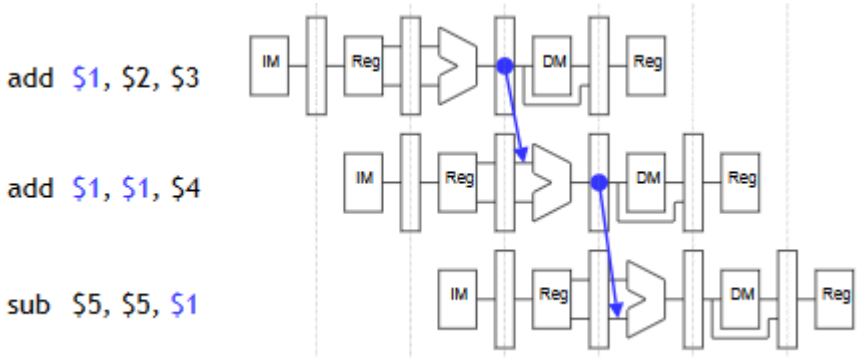


Hình 2.21 Xung đột EX/MEM

Sự xung đột dữ liệu xảy ra do thanh ghi $2 được sử dụng trong EX Stage ở lệnh thứ hai vẫn chưa được cập nhật kết quả tại WB Stage ở lệnh thứ nhất. Để giải quyết xung đột, dữ liệu sau khi tính toán trong EX Stage ở lệnh thứ nhất được đưa vào EX Stage ở lệnh thứ hai (từ thanh ghi EX-MEM) làm một toán hạng vào ALU.

- Xung đột MEM/WB xảy ra giữa lệnh đang ở EX stage và lệnh từ 2 cycles trước với các điều kiện tương tự xung đột EX/MEM

Xét một ví dụ xung đột MEM/WB (hình 2.22)

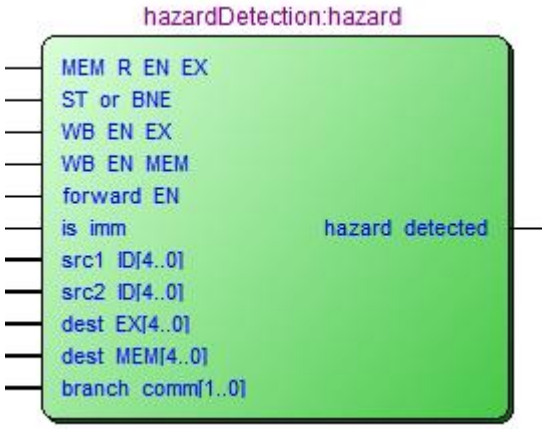


Hình 2.22 Xung đột MEM/WB

Thanh ghi $1 được ghi bởi cả lệnh thứ nhất và thứ hai, tuy nhiên chỉ cần chuyển tiếp dữ liệu kết quả gần nhất (từ lệnh thứ hai) cho lệnh thứ ba.

2.16.2 Khối phát hiện xung đột

Thiết kế khối phát hiện xung đột được thể hiện trên hình 2.23



Hình 2.23 Khối phát hiện xung đột

Các đầu vào gồm:

- MEM_R_EN_EX, WB_EN_EX: Các tín hiệu điều khiển từ thanh ghi ID-EX
- WB_EN_MEM: Tín hiệu điều khiển từ thanh ghi EX-MEM
- ST_or_BNE, is_imm: Các tín hiệu điều khiển từ bộ điều khiển
- forward_EN: Xác định có dùng chuyển tiếp dữ liệu không
- src1_ID, src2_ID (5 bits): Từ ID Stage, src1 là rt, src2 là rs hoặc rd
- dest_EX (5 bits): Địa chỉ write back tệp thanh ghi từ thanh ghi ID-EX
- dest_MEM (5 bits): Địa chỉ write back tệp thanh ghi từ thanh ghi EX-MEM
- branch_comm (2 bits): Loại lệnh rẽ nhánh, từ bộ điều khiển

Đầu ra chỉ có:

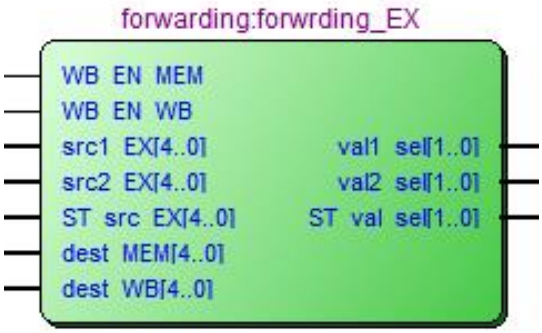
- hazard_detected: Xác định có xung đột không

Dựa vào các điều kiện phát hiện xung đột ở mục 2.16.1, các xung đột EX/MEM và MEM/WB được xác định trong code Verilog như sau:

```
assign src2_is_valid= (~is_imm)||ST_or_BNE;
assign EX_has_hazard=WB_EN_EX && (src1_ID==dest_EX || (src2_is_valid && src2_ID==dest_EX));
assign mem_has_hazard=WB_EN_MEM && (src1_ID==dest_MEM || (src2_is_valid && src2_ID==dest_MEM));
```

2.16.3 Khối chuyển tiếp dữ liệu

Thiết kế khối chuyển tiếp dữ liệu được thể hiện trên hình 2.24.



Hình 2.24 Khối chuyển tiếp dữ liệu

Các đầu vào gồm:

- WB_EN_MEM: Tín hiệu điều khiển từ thanh ghi EX/MEM
- WB_EN_WB: Tín hiệu điều khiển từ thanh ghi MEM/WB
- src1_EX, src2_EX, ST_src_EX (5 bits): Từ ID Stage, src1 là rt, src2 là rs (nếu lệnh không phải loại I) hoặc 0 (nếu lệnh loại I), ST_src_EX là địa chỉ truy cập memory
- dest_MEM (5 bits): Địa chỉ write back tệp thanh ghi từ thanh ghi EX-MEM
- dest_WB (5 bits): Địa chỉ write back tệp thanh ghi từ thanh ghi MEM-WB

Các đầu ra gồm:

- val1_sel, val2_sel, ST_val_val (2 bits): Các tín hiệu lựa chọn của 3 bộ MUX chọn toán hạng vào ALU và giá trị ghi vào memory

Dựa vào các điều kiện phát hiện xung đột ở mục 2.16.1, các tín hiệu lựa chọn val1_sel, val2_sel và ST_val_val được xác định trong code Verilog như sau:

```
always @*
begin
	//Mac dinh =0, neu co forwarding cac gia tri nay se thay doi
	{val1_sel, val2_sel, ST_val_sel} <= 0;

	//Chon du lieu chuyen tiep ghi vao memory
	if(WB_EN_MEM && ST_src_EX==dest_MEM)
		ST_val_sel<=2'd1;
	else if(WB_EN_WB && ST_src_EX==dest_WB)
		ST_val_sel<=2'd2;

	//Chon du lieu chuyen tiep cho toan hang thu nhat vao ALU
	if(WB_EN_MEM && src1_EX==dest_MEM)
		val1_sel<=2'd1;
	else if(WB_EN_WB && src1_EX==dest_WB)
		val1_sel<=2'd2;

	//Chon du lieu chuyen tiep cho toan hang thu hai vao ALU
	if(WB_EN_MEM && src2_EX == dest_MEM)
		val2_sel <= 2'd1;
	else if(WB_EN_WB && src2_EX == dest_WB)
		val2_sel <= 2'd2;
end
```

2.17 Bộ xử lí MIPS 32 bits Pipeline hoàn chỉnh

Ghép các module đã thiết kế lại với nhau, ta có bộ xử lí MIPS 32 bits Pipeline hoàn chỉnh như ở hình 2.1. Đoạn code Verilog sau thực hiện ghép các module:

```
regFile regFile(
	// INPUTS
	.clk(clock),
	.rst(rst),
	.src1(src1_ID),
	.src2(src2_regFile_ID),
	.dest(dest_WB),
	.writeVal(WB_result),
	.writeEn(WB_EN_WB),
	// OUTPUTS
	.reg1(reg1_ID),
	.reg2(reg2_ID),
	.r1(r1),
	.r2(r2),
	.r3(r3),
	.r4(r4)
);

hazardDetection hazard (
	// INPUTS
	.forward_EN(forward_EN),
	.is_imm(is_imm),
	.ST_or_BNE(ST_or_BNE),
	.src1_ID(src1_ID),
	.src2_ID(src2_regFile_ID),
	.dest_EX(dest_EX),
	.dest_MEM(dest_MEM),
	.WB_EN_EX(WB_EN_EX),
	.WB_EN_MEM(WB_EN_MEM),
	.MEM_R_EN_EX(MEM_R_EN_EX),
	// OUTPUTS
	.branch_comm(branch_comm),
```

```
      .hazard_detected(hazard_detected)
);

forwarding forwrding_EX (
              //INPUTS
      .src1_EX(src1_forw_EX),
      .src2_EX(src2_forw_EX),
      .ST_src_EX(dest_EX),
      .dest_MEM(dest_MEM),
      .dest_WB(dest_WB),
      .WB_EN_MEM(WB_EN_MEM),
      .WB_EN_WB(WB_EN_WB),
              //OUTPUTS
      .val1_sel(val1_sel),
      .val2_sel(val2_sel),
      .ST_val_sel(ST_val_sel)
);

IFStage IFStage (
      // INPUTS
      .clk(clock),
      .rst(rst),
      .freeze(hazard_detected),
      .brTaken(Br_Taken_ID),
      .brOffset(val2_ID),
      // OUTPUTS
      .instruction(inst_IF),
      .PC(PC_IF)
);

IDStage IDStage (
      // INPUTS
      .clk(clock),
      .rst(rst),
```

```
	.hazard_detected_in(hazard_detected),
	.instruction(inst_ID),
	.reg1(reg1_ID),
	.reg2(reg2_ID),
	// OUTPUTS
	.src1(src1_ID),
	.src2_reg_file(src2_regFile_ID),
	.src2_forw(src2_forw_ID),
	.val1(val1_ID),
	.val2(val2_ID),
	.brTaken(Br_Taken_ID),
	.EX_CMD(EX_CMD_ID),
	.MEM_R_EN(MEM_R_EN_ID),
	.MEM_W_EN(MEM_W_EN_ID),
	.WB_EN(WB_EN_ID),
	.is_imm_out(is_imm),
	.ST_or_BNE_out(ST_or_BNE),
	.branch_comm(branch_comm)
);

EXStage EXStage (
	// INPUTS
	.clk(clock),
	.EX_CMD(EX_CMD_EX),
	.val1_sel(val1_sel),
	.val2_sel(val2_sel),
	.ST_val_sel(ST_val_sel),
	.val1(val1_EX),
	.val2(val2_EX),
	.ALU_res_MEM(ALURes_MEM),
	.result_WB(WB_result),
	.ST_value_in(ST_value_EX),
	// OUTPUTS
	.ALUResult(ALURes_EX),
	.ST_value_out(ST_value_EX2MEM)
);
```

```
MEMStage MEMStage (
                // INPUTS
      .clk(clock),
      .rst(rst),
      .MEM_R_EN(MEM_R_EN_MEM),
      .MEM_W_EN(MEM_W_EN_MEM),
      .ALU_res(ALURes_MEM),
      .ST_value(ST_value_MEM),
      // OUTPUTS
      .dataMem_out(dataMem_out_MEM)
);

WBStage WBStage (
      // INPUTS
      .MEM_R_EN(MEM_R_EN_WB),
      .memData(dataMem_out_WB),
      .aluRes(ALURes_WB),
      // OUTPUTS
      .WB_res(WB_result)
);

IF2ID IF2IDReg (
      // INPUTS
      .clk(clock),
      .rst(rst),
      .flush(IF_Flush),
      .freeze(hazard_detected),
      .PCIn(PC_IF),
      .instructionIn(inst_IF),
      // OUTPUTS
      .PC(PC_ID),
      .instruction(inst_ID)
);
```

```
ID2EX ID2EXReg (
	.clk(clock),
	.rst(rst),
	// INPUTS
	.destIn(inst_ID[25:21]),
	.src1_in(src1_ID),
	.src2_in(src2_forw_ID),
	.reg2In(reg2_ID),
	.val1In(val1_ID),
	.val2In(val2_ID),
	.PCIn(PC_ID),
	.EX_CMD_IN(EX_CMD_ID),
	.MEM_R_EN_IN(MEM_R_EN_ID),
	.MEM_W_EN_IN(MEM_W_EN_ID),
	.WB_EN_IN(WB_EN_ID),
	.brTaken_in(Br_Taken_ID),
	// OUTPUTS
	.src1_out(src1_forw_EX),
	.src2_out(src2_forw_EX),
	.dest(dest_EX),
	.ST_value(ST_value_EX),
	.val1(val1_EX),
	.val2(val2_EX),
	.PC(PC_EX),
	.EX_CMD(EX_CMD_EX),
	.MEM_R_EN(MEM_R_EN_EX),
	.MEM_W_EN(MEM_W_EN_EX),
	.WB_EN(WB_EN_EX),
	.brTaken_out(Br_Taken_EX)
);

EX2MEM EX2MEMReg (
	.clk(clock),
	.rst(rst),
	// INPUTS
```

```
        .WB_EN_IN(WB_EN_EX),
        .MEM_R_EN_IN(MEM_R_EN_EX),
        .MEM_W_EN_IN(MEM_W_EN_EX),
        .PCIn(PC_EX),
        .ALUResIn(ALURes_EX),
        .STValIn(ST_value_EX2MEM),
        .destIn(dest_EX),
        // OUTPUTS
        .WB_EN(WB_EN_MEM),
        .MEM_R_EN(MEM_R_EN_MEM),
        .MEM_W_EN(MEM_W_EN_MEM),
        .PC(PC_MEM),
        .ALURes(ALURes_MEM),
        .STVal(ST_value_MEM),
        .dest(dest_MEM)
);

MEM2WB MEM2WB(
        .clk(clock),
        .rst(rst),
        // INPUTS
        .WB_EN_IN(WB_EN_MEM),
        .MEM_R_EN_IN(MEM_R_EN_MEM),
        .ALUResIn(ALURes_MEM),
        .memReadValIn(dataMem_out_MEM),
        .destIn(dest_MEM),
        // OUTPUTS
        .WB_EN(WB_EN_WB),
        .MEM_R_EN(MEM_R_EN_WB),
        .ALURes(ALURes_WB),
        .memReadVal(dataMem_out_WB),
        .dest(dest_WB)
);
```

CHƯƠNG 3. Mô phỏng và triển khai trên kit FPGA DE2

3.1 Chương trình 1

3.1.1 Chương trình thực hiện

Vì tài nguyên hiển thị của kit DE2 có hạn nên em chỉ nạp vào bộ nhớ lệnh đoạn chương trình sau:

TT	Địa chỉ	Lệnh
1	0	addi $5, $0, 105
2	4	addi $6, $0, 106
3	8	add $1, $5, $6
4	12	and $2, $1, $5
5	16	or $3, $1, $6
6	20	sub $4, $6, $5

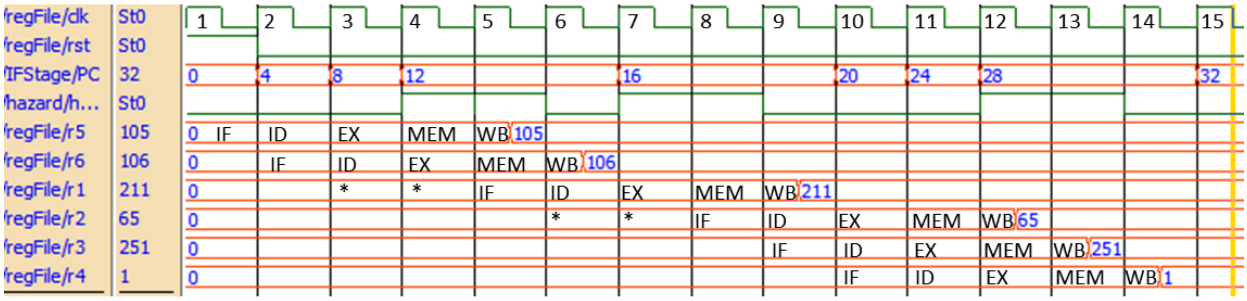
Có thể thấy xung đột dữ liệu xảy ra giữa lệnh 1 và 3 (do $5), lệnh 2 và 3 (do $6), lệnh 3 và 4, 3 và 5 (do $1). Theo tính toán giá trị 4 thanh ghi $1, $2, $3, $4 lần lượt là: 211 ($D3_{16}$), 65 ($41_{16}$), 251 ($FB_{16}$), 1 ($01_{16}$). Sự xung đột và kết quả tính toán sẽ được kiểm tra ở phần mô phỏng và triển khai trên kit DE2.

3.1.2 Kết quả mô phỏng

- Khi không dùng chuyển tiếp dữ liệu

Khi không dùng chuyển tiếp dữ liệu, các chu kì đợi được chèn vào giữa các lệnh có xung đột (phương pháp dừng và chờ). Trong các chu kì này, bộ xử lí thực hiện lện NOP (No operation, không làm gì cả).

Kết quả mô phỏng được thể hiện trên hình 3.1.



Hình 3.1 Kết quả mô phỏng chương trình 1 (không dùng chuyển tiếp dữ liệu)

Sau mỗi xung clk, bộ xử lí truy cập lệnh tiếp theo qua địa chỉ trên thanh ghi PC, mỗi stage của lệnh được thực hiện trong 1 clk. Lệnh đầu tiên được truy cập tại địa chỉ 0, và sau 5 chu kì clk cho ra kết quả thanh ghi $5=105. Điều tương tự xảy ra với lệnh thứ hai tại địa chỉ 4.

Tại clk 4, xung đột được phát hiện sau khi vừa thực hiện xong EX Stage của lệnh 1, vì vậy lệnh 3 tại địa chỉ 8 phải đợi 2 chu kì . Theo lí thuyết giữa lệnh 2 và lệnh 3 cũng có xung đột, nhưng sau khi đợi 2 chu kì thì không còn xung đột nữa do ở clk 6 giá trị $6 đã được cập nhật ở WB Stage của lệnh 2 trước khỉ truy cập ở ID Stage của lệnh 3.

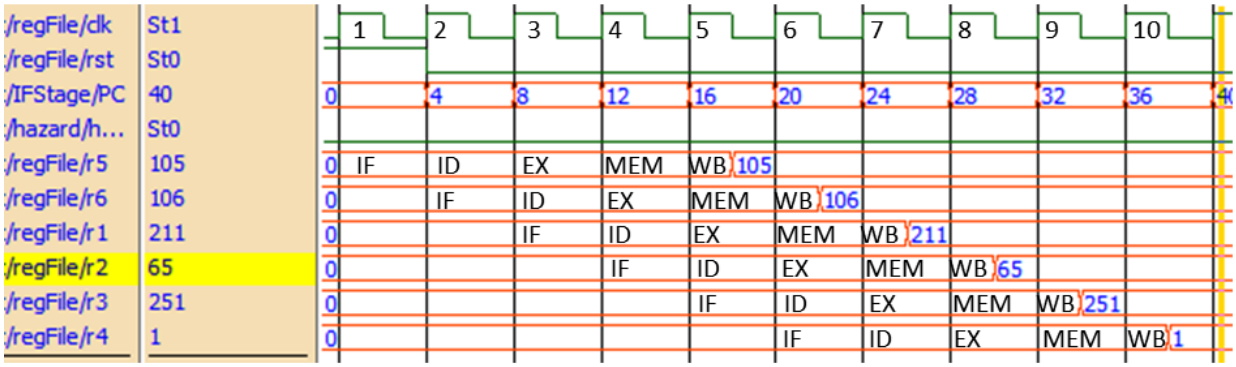
Giữa lệnh 3 và 4 xảy ra xung đột, vì vậy lệnh 4 phải đợi 2 chu kì. Theo lí thuyết giữa lệnh 3 và 5 cũng xảy ra xung đột, nhưg với lí do tương tự trên thì trên thực tế không còn xung đột nữa. Các lệnh 5 và 6 được thực hiện một các bình thường, kết quả ra đúng như tính toán. Bộ xử lí mất tổng cộng 14 clock cycles để thực hiện 6 lệnh.

Số clock cycles trung bình để thực hiện một lệnh là:

$$\mathrm{CPI_{tb}} = \frac{\mathrm{C}}{\mathrm{I}} = \frac{14}{6} \approx 2.33$$

- Khi dùng chuyển tiếp dữ liệu

  Kết quả mô phỏng được thể hiện trên hình 3.2.



Hình 3.2 Kết quả mô phỏng chương trình 1 (dùng chuyển tiếp dữ liệu)

Khi dùng chuyển tiếp dữ liệu, hiệu năng được cải thiện đáng kể khi các lệnh có xung đột không phải đợi mà có thể thực hiện bình thường như pipeline lí tưởng. Trong trường hợp này bộ xử lí mất 10 clock cycles để thực hiện 6 lệnh.

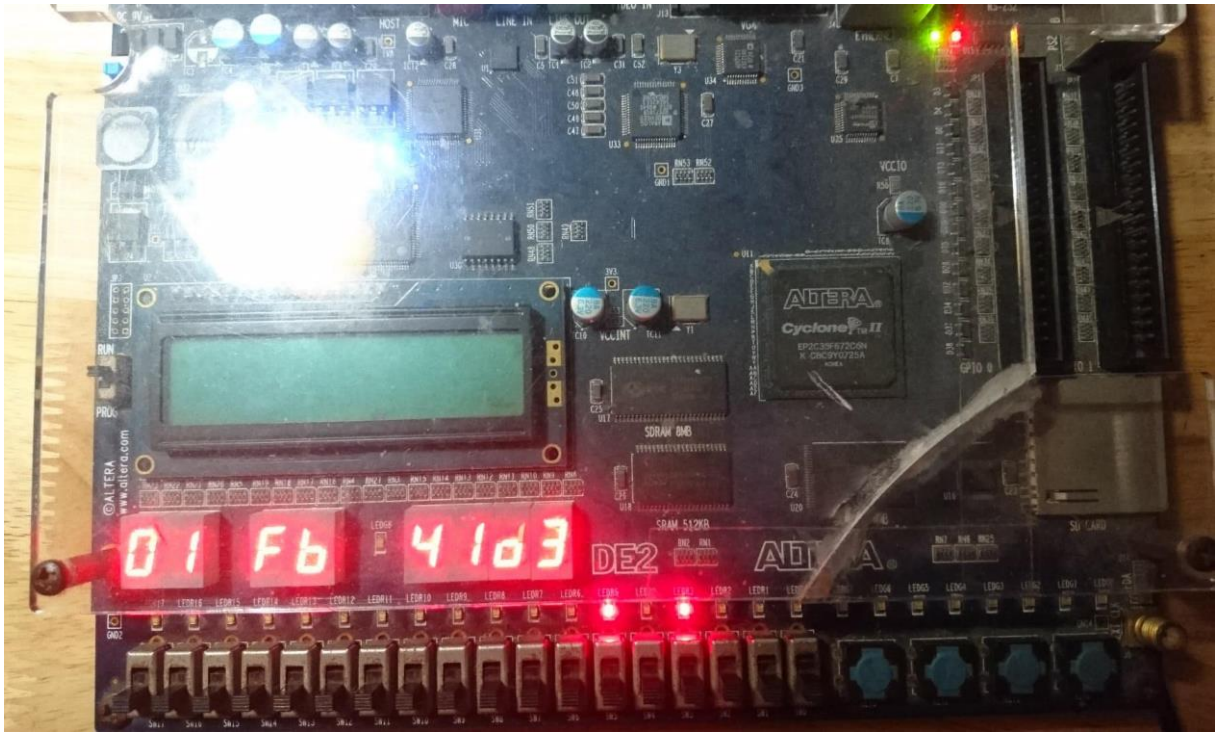
Số clock cycles trung bình để thực hiện một lệnh là:

$$\mathrm{CPI_{tb}} = \frac{\mathrm{C}}{\mathrm{I}} = \frac{10}{6} \approx 1.67$$

3.1.3 Kết quả triển khai trên kit FPGA DE2

Trên kit DE2, SW[0] được dùng làm tín hiệu reset, SW[1] chọn có sử dụng chuyển tiếp dữ liệu hay không, các led 7 thanh HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 hiển thị giá trị các thanh ghi $1, $2, $3, $4 (ở hệ 16), LEDR[15:0] hiển thị địa chỉ lệnh hiện tại (thanh ghi PC), LEDG[7] hiển thị tín hiệu phát hiện xung đột.

Kết quả triển khai trên kit DE2 được thể hiện trên hình 3.3.



Hình 3.3 Triển khai bộ xử lí MIPS 32 bits pipeline trên kit DE2 (chương trình 1)

Như vậy kết quả triển khai trên kit DE2 đúng như lí thuyết và mô phỏng.

3.2 Chương trình 2

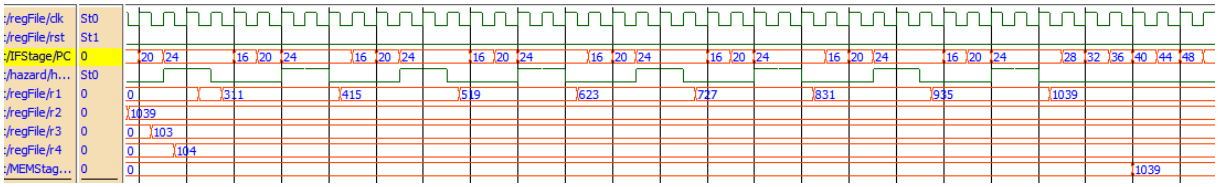
3.2.1 Chương trình thực hiện

TT	Địa chỉ	Lệnh
1	0	addi $2, $0, 1039
2	4	addi $3, $0, 103
3	8	addi $4, $0, 104
4	12	add $1, $3, $4
5	16 (L1)	add $1, $1, $4
6	20	bne $1, $2, L1
7	24	sw $1, 0($0)

Trong đoạn chương trình trên xảy ra xung đột dữ liệu giữa lệnh 2,3 với lệnh 4 (do $3, $4), lệnh 4 với lệnh 5 (do $1) và xung đột điều khiển giữa lệnh 5 với lệnh 6 (do $1). Theo tính toán giá trị cuối cùng của $1 là 1039 ($40F_{16}$).

3.2.2 Kết quả mô phỏng

Kết quả mô phỏng được thể hiện trên hình 3.4.



Hình 3.4 Kết quả mô phỏng chương trình 2

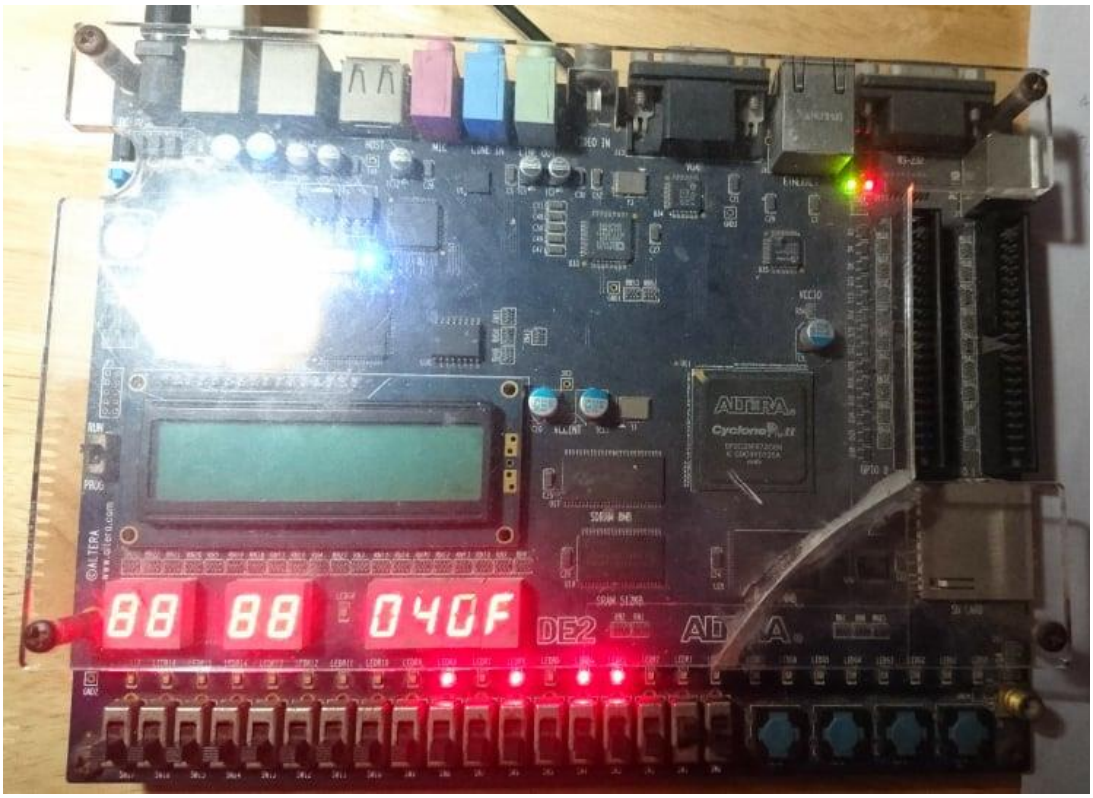
Các giá trị $2, $3, $4 được tính toán bình thường. Khi vào vòng lặp, $1 được cộng với $4 đến khi bằng $2 là 1039. Ở đây có xung đột điều khiển do $1 trong vòng lặp, tuy nhiên không thể chuyển tiếp dữ liệu từ thanh ghi EX-MEM của lệnh 5 đến EX Stage ở lệnh 6 vì lệnh 6 cần xác định địa chỉ lệnh tiếp theo ở IF Stage, vì vậy vẫn phải đợi 2 chu kì. Sau khi thoát khỏi vòng lặp, giá trị $1 được ghi vào memory tại địa chỉ 0.

Số clock cycles trung bình để thực hiện một lệnh là:

$$\mathrm{CPI_{tb}} = \frac{\mathrm{C}}{\mathrm{I}} = \frac{48}{21} \approx 2.29$$

3.2.3 Kết quả triển khai trên kit FPGA DE2

Kết quả triển khai trên kit DE2 được thể hiện trên hình 3.5.



Hình 3.5 Triển khai bộ xử lí MIPS 32 bits pipeline trên kit DE2 (chương trình 2)

Giá trị $1 được hiển thị ra đúng theo lí thuyết và mô phỏng

KẾT LUẬN

Như vậy em đã thực hiện thành công việc thiết kế, mô phỏng và triển khai trên kit DE2 của bộ xử lí MIPS 32 bits pipeline. Đây là một bài tập tương đối phức tạp, đòi hỏi nhiều thời gian, công sức để nghiên cứu thực hiện. Qua bài tập lớn nay đã giúp nem hiểu sâu hơn về code Verilog, thiết kế bộ xử lí pipeline và sử dụng kit để triển khai mạch.

TÀI LIỆU THAM KHẢO

[1] David A. Patterson and John L. Hennessy, *Computer Organization and Design, The Hardware/ S oftware Interface,* 4th Edition, Elsevier Inc, 2014

[2] CS61C Course Summer 2017, *Great Ideas in Computer Architecture (Machine Structures),* University of California, Berkeley, 2017

[3] CSE378 Course Autumn 2007, *Machine Organization & Assembly Language*, Paul G.Allen School, 2007

[4] https://github.com/mhyousefi/MIPS-pipeline-processor, truy cập lần cuối ngày 16/12/2019

PHỤ LỤC

Toàn bộ mã nguồn của bài tập lớn này nằm trong link sau:

https://drive.google.com/open?id=1nPMiWybXs5w3UjU6A9U69Yk4t-9_I_vB