Five-Stage Pipelined RISC-V Core with Stall/Forwarding Datapath

G21 Final Project Report

109022210梁皓鈞, 109061204陳永駿, 109061210吳士宣

Introduction

5-staged pipelined risc-v core將direct risc-v core架構分為5個stage，分別為Instrution fetch stage (IF stage)、Instruction decode stage (ID stage)、Execute stage (EXE stage)、Memory stage (MEM stage)以及WriteBack stage (WB stage)，以達到分割datapath促使timing上升的目的。另外，此架構由於快取是multi-port使structure hazard被排除了以外，衍生的data hazard以及control hazard需要跨stage的stall unit以及forward unit來排除。其中，雖然跨stage會導致datapath變長、timing上升以外，area還會因此變大，然則compiler如此便不需要判斷pattern在machine code中何處需額外嵌入no operation (NOP)，減少compiler的負擔並使cycle count、coding length變短；前者能減低程式中loop所產生之大量stall，達到良好的優化、後者能使暫存存取更長的程式達到較佳的spatial locality，同樣能減低cycle count。

Functionality

於IF Stage為了fetch instruction，需要有能以testbench初始化instruction pattern的暫存以及指標來存儲現在instruction的位置，並依序傳遞pattern中的指令到ID Stage。指令包含opcode、funct、兩個source register和一個destination register的address以及一個immediate，為了解碼會將immediate根據opcode作extension、以兩個source register的address從暫存取值出來外，還會以opcode和funct生成對應的控制訊號。另外，為了處理control hazard減少stall cycle，會需要提前輸出兩暫存值是否相等的判比值以及PC與immediate的和。於EXE stage，會以ID stage得到的兩source register值、新的immediate和控制訊號選定需要的運算子和運算符，經過EXE stage的計算單元處理後，傳到Mem stage或WB stage作存儲使用。

Specification

Area : 158545.728um^2 (APR), 211083.768um^2 (core)

Utilization: 75.11%

Power : 1.551mW

#cycle : 1263

Implementation

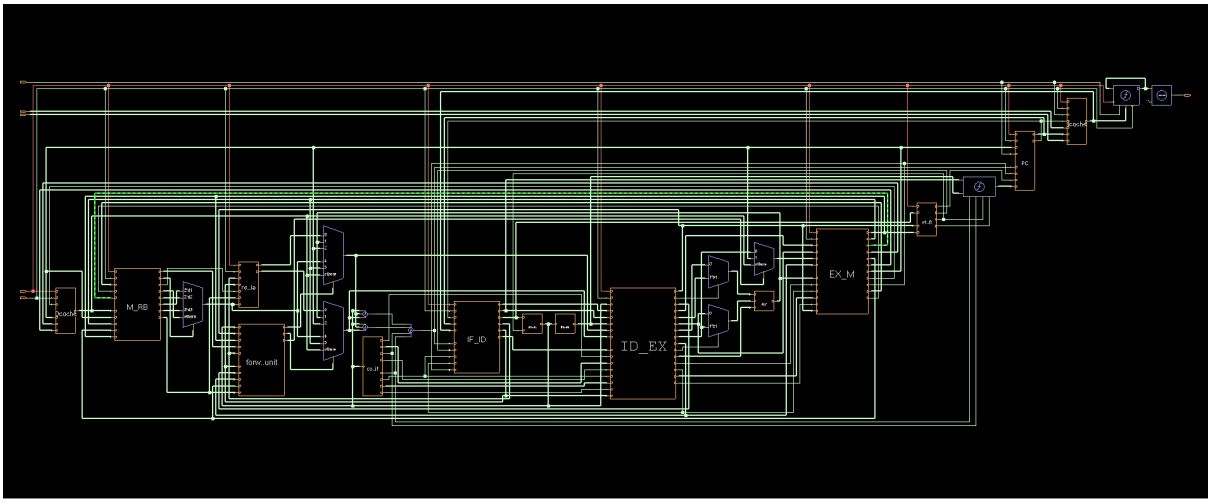


Fig. 1. block diagram of our design

IF Stage

Icache_unit (Icache)

此unit提供riscv core assembly code的instruction pattern以驗證core的functionality，內建2個128x32的register file；會需要這樣設置的原因為4096 bits為spyglass可合成上限。初始將instruction boot進Icache時，testbench會將write enable打開，testbench隨後指定address和machine code到Icache完成初始化後，關閉write enable開始執行pattern，由word addressing以PC除以四作為Icache的address，持續輸出file於該address時register的值至read_data，供ID stage decoding。

PC_unit (Program Counter)

此unit提供執行Icache的程式指標位置。內建一個小型的FSM，在boot_up抬到1再降回0使Icache_unit完成pattern初始化後，PC才會從0開始下數。預設情況下，由word addressing，PC會一直累加四，遇到branch、jalr、stall時才會出現不同的運算；branch會將下一個PC指定為PC+immediate，jalr為register+immediate，stall則是會停止PC在目前的位子。

ID Stage

Control_unit (Control unit)

此unit作為控制整個core的核心，而控制單元接收自IF stage傳到ID stage的instruciotn訊號，轉換成funct3、funct7和opcode後，根據排列組合以case控制ALU_src、ALU_ctrl[3:0]、Branch、MemWrite、PMAItoReg、Rd_wen、Jal、Jalr，詳細說明如下：

1. ALU_src控制ALU第二個運算子前的MUX選擇rs2_rdata或immediate，前者是在Rtype、Btype、Lui、Jtype會選定。

2. ALU_ctrl[3:0]控制ALU將執行何種運算，細節於ALU_unit。

3. Branch供PC_unit判斷是否吃Beq或Bne運算。另外，由於需解決control hazard問題，rs1_rdata與rs2_rdata在ID stage便已比較，與Branch共同形成glue logic才輸入PC_unit，供PC_unit判斷是否要選定PC + immediate。

4. MemWrite提供Dcache判斷該或不該該執行寫入的動作，前者是在S-type時會選定。

5. PMAItoReg控制Regfile_unit寫入訊號前的MUX選擇PC + 4、Mem_rdata、ALU_result或immediate，分別是在Jalr、Jtype和Load和Rtype、I_type、Auipc和Stype、Btype、Lui。

6. Rd_wen控制regfile該或不該寫入以PMAItoReg所控制的MUX選定的訊號，後者是在Stype以及Btype時會選定。

7. Jal提供PC判斷是否遇到Jump and Link訊號，Jalr則提供PC判斷是否遇到jump and link register訊號。

Imm_gen (Immediate generator)

這個module以ID stage經過stall unit和nop mux選擇的instruction (instr_ID_mux)為輸入訊號，並根據該instruction後七位的opcode (instr_ID_mux[6:0])，判斷該instruction的類別，並根據RISC-V的規則，產生相對應的immediate (imm_ID)，以供其他module (e.g. ALU)或邏輯判斷(e.g. forwarding)使用。Instruction type與immediate的關係如Table 1所示。因此，在這個module，我們使用簡單的case敘述便能完成immediate generator的功能。

type	immediate (imm[31:0])
R-type	{32'd0}
I-type	{{20{instr[31]}}, instr[31:20]}
S-type	{{20{instr[31]}}, instr[31:25], instr[11:7]}
B-type	{{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'd0}
U-type	{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'd0}
J-type	{instr[31:12], 12'd0}
default	{32'd0}

Table 1. RISC-V instruction-immediate table

Regfile (Register file)

這個module負責描述32個register在讀與寫兩種情形下的表現。由於讀值並不會影響register中所存放的資料，因此我們使用combinational的設計，直接assign register中的值到輸出端。相反地，由於寫入會直接更改register中所存放的資料，因此我們使用sequential的設計，在下一個positive edge clock才會更新register中的資料。

EXE Stage

ALU (Arithmetic Logic Unit)

這個module根據input的ALU_ctrl訊號，對另外兩個input的operand1(opr1)和operand2(opr2)進行不同運算。由於我們發現在RISC-V的32I instruction set架構下，ALU執行的運算種類可以藉由control unit將所有instruction中的funct7以及funct3進行整理，全部類比成R-type instruction的funct7和funct3，又因funct中真正會改變的只有funct7[5]一個bit而已，所以我們將ALU_ctrl設計成一個4 bit的訊號，其中ALU_ctrl[3]為funct7[5]，而ALU_ctrl[2:0]為統整後的funct3。ALU_ctrl與ALU的運算種類如Table 2所示，且可用單純的case敘述完成。另外，ALU還有一個1 bit輸出訊號zero，當ALU的運算結果為32'd0時，zero為1'd1，反之則為1'd0。Zero訊號在我們尚未提前在ID stage判斷B-type instruction時，在EX stage判斷B-type instruction。

ALU_ctrl	ALU_result
4'h0000	$signed(operand1) + $signed(operand2)
4'h1000	$signed(operand1) - $signed(operand2)
4'b0100	operand1 ^ operand2
4'b0110	operand1 | operand2
4'b0111	operand1 & operand2
4'b0001	operand1 << operand2[4:0]
4'b0101	operand1 >> operand2[4:0]
4'b1101	$signed(operand1) >>> operand2[4:0]
4'b0010	($signed(operand1) < $signed(operand2)) ? 1 : 0
4'b0011	(operand1 < operand2) ? 1 : 0
default	0

Table 2. ALU_ctrl-ALU_result table

MEM Stage

Dcache_unit (Dcache)

此unit作為core的暫存器，供load以及store使用，內建初始被reset成0的128x32的register_file，以ALU_result作為file的address，持續輸出file於該address時register的值至mem_rdata，其中，mem_rdata會根據funct3的值，分別為0: Load_byte，1: Load_half，2: Load_word，以mask和signed extension做調整。當Memwrite為1時，同樣以ALU_result作為file的address，透過funct3的值，分別為0: Store_byte，1: Store_half，2: Store_word，更新該register的存值。

Hazard Control Units

Forward_unit (Forward unit)

此unit處理經過pipeline後所有產生的data hazard問題。Forward_unit根據ID stage、EXE stage、MEM stage、WB stage的instruction以及rs1、rs2 address的排列組合，控制ALU第一個運算子、第二個運算子以及EXE stage的rs2前的MUX和是否要forward訊號，以避免資料來不及存到Dcache或寫回regfile但後面的instrucion便已decode出regfile較舊的值或讀取到Dcache較舊的資料，導致後續的運算產生錯誤。

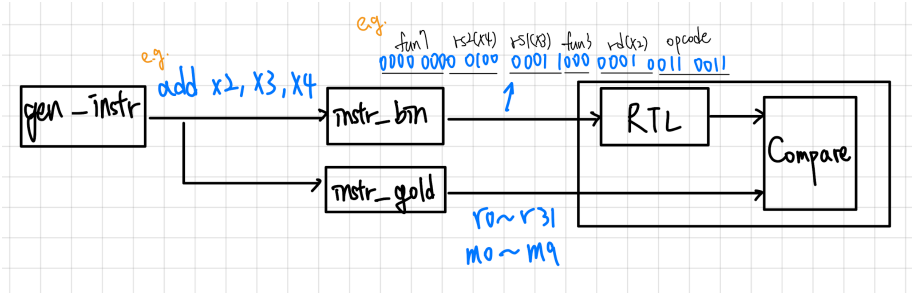
Stall_unit (Stall unit)

當data hazard發生時，有些情況會是froward unit無法解決的，換言之，CPU必須讓instruction「空轉」，再進行下一個instruction。在我們的設計中，我們會偵測CPU pipeline中ID stage、EXE stage和MEM stage的instruction，並根據需要空轉的instruction type組合，以及instruction之間register address的相同與否寫case敘述，決定是否要stall。Stall的方式則是在ID stage時透過nop_sel控制nop mux，選擇是否將ID stage的instruction服寫成NOP(32'd0)，同時用keep_instr控制IF_ID的DFF將instruction「hold住」，再用keep_PC將PC的counter「hold住」。另外還有一列邏輯判斷式，會將IF_ID中於決定stall時才判斷不應該執行的instruction(e.g. beq instruction後的PC+4)洗成NOP。由於stall unit的設計很大程度取決於CPU整體的結構以及和forward unit之間的相互配合，因此stall unit所處理的instruction組合並不固定。而在我們目前的設計中，stall unit所負責的case如Table 3所示。

ID	EXE	MEM	address	stall
R	I-load	-	rd_EX = rs1_ID(rs2_ID)	T
I-imm	I-load	-	rd_EX = rs1_ID	T
S	I-load	-	rd_EX = rs1_ID	T
B	I-load	-	rd_EX = rs1_ID(rs2_ID)	T
B	NOP	I-load	rd_EX = rs1_ID(rs2_ID)	T
~J	B	-	-	T
other				F

Table 3. stall table

Verification



在Verification中，主要可以分成兩部分，Testbench與生成test pattern和golden result的python script。Testbench會利用python script產生的instruction與golden做驗證。

Testbench

這部分與lab6驗證cpu的方式類似，利用readmem的方式讀入txt file裡的binary code，並刻入cpu裡的icache。CPU處理完icache裡的instruction後，testbench會將cpu裡32個register與10個memory的值與golden作對比來確認RTL的正確性。instruction與golden這樣為一個set，為了確保RTL在不同測資的正確性，我們也使他能在一次測試中連續測試100個或以上的不同set。

Test Pattern (instruction and golden)

Test Pattern 中可分為instruction與golden兩份不同的txt file，instruction指的是cpu能讀懂的binary code，也就是從riscv instruction set decode出來的，而golden是從riscv interpreter得到register和memory的值，用來與cpu運算完的結果做比較。

Test Pattern 的兩個部分分別用三個部份的python script做處理。第一個是產生可閱讀的instruction set。這個instruction set會方便我們在debug時做閱讀，也會在後面的步驟轉為cpu使用的binary code。首先將RV32I的instruction做分類，並在分類後根據會產生data hazard的組合產生pattern。主要會產生hazard的組合如下：

3 forward : ALU->ALU, ALU->LW, ALU->SW, AUIPC->LUI, LUI->AUIPC, LW->LW, LW->SW
2 forward+1 stall : ALU->BRANCH, ALU->JAL, LW->BRANCH, LW->ALU

A->B 指在A operation後接B operation會產生data hazard。每一個data hazard都可以藉由塞三個NOP operation，所以3 bypass是指這三個NOP都可以用forward unit來做跳過，而2 forward 1 stall則是指兩個需要forward unit 一個需要 stall unit 才能跳過。由於我們有將不同hazard種類做分類，所以在實作時我們可以依據進度來調整我們生成的test pattern。在未implement forward, stall 時 test pattern 先加NOP，implement 後則可以測不需要NOP的。第二個部分則是將可閱讀的insrutcion set轉為cpu可使用的binary code。我們利用網路上的riscv instruction decoder (https://luplab.gitlab.io/rvcodecjs/)，將剛剛第一部份產生的instruction set逐行的寫入該網站，並讀出我們需要的binary code寫入intruction.txt。這部分藉由python selenium 的library 來達成。第三個部分則是將可閱讀的instruction set寫入網路上的riscv interpreter(https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/)，等該網頁執行完所有instruction後，讀出registers與memory的值並寫入golden.txt。這三個部分都有利用shell script來使他們可以連續產生100或以上個不同pattern。

另外在會產生hazard的pattern外，我們也另外做了QuickSort的instruction set進行測試，產生binary code與golden的方法一樣利用第二部分與第三部分的script。

Experimental Results

紅色粗體處為回答助教之問題。

由於本專題旨在利用硬體處理hazard情形，必免直接在instruction set (Icache)中直接插入NOP，藉此減少CPU執行instruction set的cycle count，增加CPU的運算速度，因此我們的實驗結果將會著重於比較加入hazard control unit前後兩種CPU設計的性能表現。我們以同一份quicksort的instruction set作為輸入，差別在於輸入無**hazard control unit的CPU時(我們剛開始設計沒有stall、forward unit的備份版本)**，我們必須加入NOP以避免hazard發生，而加入了hazard control unit的CPU則不需要加入NOP便能正確執行quicksort。兩種CPU的性能表現如Table 4所示。由實驗結果可知，對於每一種會產生hazard的instruction的組合，加入hazard control unit的CPU都會比原本快1~3個cycle，因此在worst case (all hazard)下，本專題所實做出來的CPU將會比原本的快接近四倍。此外，由於新的CPU不再需要NOP的輸入，因此可以有效減少I-cache的更新率，提高spatial locality。

	Basic CPU	improved CPU	Enhance
instruction set length	199	84	58% improve
#cycle(pattern dependent)	~2.7k (3 NOP / hazard)	~1.2k(0~2 NOP / hazard)	**total 54% improve (~10% improve if pc + imm moved from EXE to ID stage)**
area (synthesize)	$157836.0815\ um^2$	$158257.4255\ um^2$	almost same
timing (synthesize)	4.5	5.58	24% delay

Table 4. simulation results

Contributions

RTL Code Construction

109022210梁皓鈞 : top module, I-cache, forward unit

109061204陳永駿 : PC unit, control unit, D-cache

109061210吳士宣 : immediate generator, register file, ALU, stall unit

Simulation

109061204陳永駿 : testbench, python code(random pattern and quicksort)

109022210梁皓鈞、109061210吳士宣 : debug, CPU structure adjustment

Synthesis

三人平行進行

LEC

109022210梁皓鈞

APR

三人平行進行