CA Final Report

工海四 張哲浩 B06505027
工海四 黃日新 B06505048

A. CPU Architecture:

CPU主要分成三個state進行運算：

1) Fetch：抓取instruction並decode，產生出control signals，同時進行ALU的運算。
2) Await：等待MulDiv運算（僅在instruction為mul或div時會進入）。
3) Write：將運算結果寫入對應的register或memory中，並計算下一次PC的值。

首先CPU在抓到新的instruction後會進行decode（如下圖Instruction decode、immediate decode），並根據decode出的結果產生出control signals（如下圖Control singals）。

```
// Instruction decode
assign mem_addr_I = PC;
assign opcode = mem_rdata_I[ 6: 0];
assign rd     = mem_rdata_I[11: 7];
assign funct3 = mem_rdata_I[14:12];
assign rs1    = mem_rdata_I[19:15];
assign rs2    = mem_rdata_I[24:20];
assign funct7 = mem_rdata_I[31:25];

// Immediate decode
always @(*) begin
    case (opcode)
        OP_AUIPC: imm = {{12{mem_rdata_I[31]}},
        OP_JAL  : imm = {{11{mem_rdata_I[31]}},
        OP_B    : imm = {{19{mem_rdata_I[31]}},
        OP_SW   : imm = {{20{mem_rdata_I[31]}},
        default : imm = {{20{mem_rdata_I[31]}},
    endcase
end
```

```
// Control signals
assign is_slti = opcode == OP_I & funct3[1];
assign is_sub  = opcode == OP_R & funct7[5];
assign is_md   = opcode == OP_R & funct7[0];
assign is_PC   = opcode[2] & (opcode[3] | opcode[4]);
assign is_rs2  = opcode == OP_B | opcode == OP_R;
assign j_en    = opcode[2] & opcode[6];
assign b_en    = opcode == OP_B & ((!funct3[2] & (funct3[0]
```

接著再利用instruction和control singals判斷ALU、MulDiv的輸入訊號。例如：如果instruction是B type或R type (is_rs2 = 1)，則ALU的第二個輸入(alu_in_B)應為rs2_data，否則為imm（如下圖ALU controll、MDU control）。

```
// ALU control
assign alu_valid   = state == FETCH & !is_md;
assign alu_mode[2] = opcode == OP_I & funct3[0]; // shift or not
assign alu_mode[1] = funct3[2]; // shift left or right
assign alu_mode[0] = opcode == OP_B | is_slti | is_sub; // sub or not
assign alu_in_A    = (is_PC)? PC : rs1_data;
assign alu_in_B    = (is_rs2)? rs2_data : imm;

// MDU control
assign mdu_valid = state == FETCH & is_md;
assign mdu_mode  = funct3[2];
assign mdu_in_A  = rs1_data;
assign mdu_in_B  = rs2_data;
```

最後在ALU或MulDiv運算完成後，同樣根據instruction、control singals判斷要將register、memory寫入的資料為何，以及計算下一次PC的值。例如：如果branch成立(b_en = 1)，則PC_nxt的值會被改成PC+imm（如下圖PC control、Register write control、memory write control）。

```
// PC control
assign PC_nxt = (j_en)? alu_out[31:0] : (b_en)? PC + imm : PC + 32'd4;

// Register write control
assign regWrite = (state == WRITE & opcode != OP_SW & opcode != OP_B);
always @(*) begin
    if (j_en) rd_data = PC + 32'd4;
    else if (opcode == OP_LW) rd_data = mem_rdata_D;
    else if (is_slti) rd_data = {31'd0, alu_out[31]};
    else rd_data = (is_md)? mdu_out[31:0] : alu_out[31:0];
end

// Memory write control
assign mem_wen_D   = (state == WRITE && opcode == OP_SW)? 1'b1 : 1'b0;
assign mem_addr_D  = alu_out[31:0];
assign mem_wdata_D = rs2_data;
```

B. Data Path:

CPU支援以下指令:

Instruction	alu_out	PC_nxt	reg/mem write
AUIPC	PC + imm	PC + 4	rd <= PC + imm
JAL	PC + imm	PC + imm	rd <= PC + 4
JALR	rs1 + imm	rs1 + imm	rd <= PC + 4
BEQ	rs1 – rs2	PC + (imm or 4)	
BNE	rs1 – rs2	PC + (imm or 4)	
BLT	rs1 – rs2	PC + (imm or 4)	
BGE	rs1 – rs2	PC + (imm or 4)	
LW	rs1 + imm	PC + 4	rd <= mem[ ]
SW	rs1 + imm	PC + 4	mem <= rs2
ADDI	rs1 + imm	PC + 4	rd <= rs1 + imm
SLTI	rs1 – imm	PC + 4	rd <= (rs1 < imm)?
SLLI	rs1 << imm	PC + 4	rd <= (rs1 << imm)
SRLI	rs1 >> imm	PC + 4	rd <= (rs1 >> imm)
ADD	rs1 + rs2	PC + 4	rd <= rs1 + rs2
SUB	rs1 – rs2	PC + 4	rd <= rs1 – rs2
MUL	rs1 * rs2	PC + 4	rd <= rs1 * rs2
DIV	rs1 / rs2	PC + 4	rd <= rs1 / rs2

*SLLI和SRLI的imm代表shamt

*完整版請參閱附錄一：CPU Instruction List

CPU的各種control singals以及不同instruction下資料的data path即是利用上述表格歸納產生。例如：

1) AUIPC:將PC、imm輸入ALU進行加法，運算結果寫入rd中。
2) JAL:將PC、imm輸入ALU進行加法，運算結果寫入PC_nxt中，同時將PC+4寫入rd中。
3) JALR:將rs1、imm輸入ALU進行加法，運算結果寫入PC_nxt中，同時將PC+4寫入rd中。

C. Multi-cycle instructions handling:
乘法計算的部分因為要多個cycle才能完成，所以我們在Fetch的state讀到輸入進來的instruction為mul或div時，下一個cycle不會馬上進到Write back的state，而是設一個Await的state，先進到這個state裡面讓乘除器的module計算完成後，ready會拉起來，狀態機判讀到之後就會離開Await state，進到Write back state把計算好的值寫進暫存器。

```
// State trasition
always @(*) begin
    case (state)
        FETCH  : next_state = (mem_addr_I == 32'b0)? FETCH : (is_md)? AWAIT : WRITE;
        AWAIT  : next_state = (mdu_ready)? WRITE : AWAIT;
        WRITE  : next_state = FETCH;
        default: next_state = FETCH;
    endcase
end
```

D. Simulation time:
<leaf>

```
==========================================================

Success!
The test result is .....PASS :)

==========================================================

Simulation complete via $finish(1) at time 485 NS + 0
```

<fact>

```
==========================================================

Success!
The test result is .....PASS :)

==========================================================

Simulation complete via $finish(1) at time 2135 NS + 0
```

<hw1>

```
==========================================================

Success!
The test result is .....PASS :)

==========================================================

Simulation complete via $finish(1) at time 845 NS + 0
```

E. Observation:

感受到 instruction format 有經過設計的好處，其中之一就是當不同的指令要做ALU運算時，運算的資訊內容、資料長度等皆有幾組相同規範之處，可以將運算值經整理選定後傳入一個ALU運算元，算好再傳回寫入暫存器。這樣就不用每個指令都對應一套自己的加減法器，達到hardware sharing以降低不必要的空間成本。

F. Register Table:

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
state_reg	Flip-flop	2	Y	N	Y	N	N	N	N
PC_reg	Flip-flop	31	Y	N	Y	N	N	N	N
PC_reg	Flip-flop	1	N	N	N	Y	N	N	N

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
mem_reg	Flip-flop	995	Y	N	Y	N	N	N	N
mem_reg	Flip-flop	29	Y	N	N	Y	N	N	N

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
shreg_reg	Flip-flop	64	Y	N	Y	N	N	N	N
cnt_reg	Flip-flop	5	Y	N	Y	N	N	N	N
state_reg	Flip-flop	2	Y	N	Y	N	N	N	N
alu_in_reg	Flip-flop	32	Y	N	Y	N	N	N	N

G. Word Distribution:

討論好CPU & Data path & Register統一架構後各寫一份，再進行整合及程式碼優化。

張哲浩	黃日新
50%	50%

附錄一：CPU Instruction List

31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	Instruction	alu_out	PC_nxt	reg/mem write
imm[31:12]				rd	0010111	AUIPC	PC + imm	PC + 4	rd <= PC + imm
imm[20|10:1|11|19:12]				rd	1101111	JAL	PC + imm	PC + imm	rd <= PC + 4
imm[11:0]		rs1	000	rd	1100111	JALR	rs1 + imm	rs1 + imm	rd <= PC + 4
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ	rs1 – rs2	PC + (imm or 4)	
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE	rs1 – rs2	PC + (imm or 4)	
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT	rs1 – rs2	PC + (imm or 4)	
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE	rs1 – rs2	PC + (imm or 4)	
imm[11:0]		rs1	010	rd	0000011	LW	rs1 + imm	PC + 4	rd <= mem[ ]
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW	rs1 + imm	PC + 4	mem <= rs2
imm[11:0]		rs1	000	rd	0010011	ADDI	rs1 + imm	PC + 4	rd <= rs1 + imm
imm[11:0]		rs1	010	rd	0010011	SLTI	rs1 – imm	PC + 4	rd <= (rs1 < imm)?
imm[11:0]		rs1	001	rd	0010011	SLLI	rs1 << imm	PC + 4	rd <= (rs1 << imm)
imm[11:0]		rs1	101	rd	0010011	SRLI	rs1 >> imm	PC + 4	rd <= (rs1 >> imm)
0000000	rs2	rs1	000	rd	0110011	ADD	rs1 + rs2	PC + 4	rd <= rs1 + rs2
0100000	rs2	rs1	000	rd	0110011	SUB	rs1 – rs2	PC + 4	rd <= rs1 – rs2
0000001	rs2	rs1	000	rd	0110011	MUL	rs1 * rs2	PC + 4	rd <= rs1 * rs2
0000001	rs2	rs1	100	rd	0110011	DIV	rs1 / rs2	PC + 4	rd <= rs1 / rs2