รายงานโครงงานฉบับสมบูรณ์

วิชา 2110252 Digital Logic

CEDT Final Project 2568: Mini CPU

จัดทำโดย:

ชื่อทีม : [OCFL]

สมาชิกในกลุ่ม:

นายนรินธร ยางงาม, [6833136121]

เมธาสิทธิ์ พนาวงศ์วัชร์, [6833227621]

กณวรรธน์ ภูมิชัย, [6833002721]

ศิวกร โพธิ์ทอง, [6833299821]

เสนอ
ผศ.ดร.ณรงค์เดช กีรติพรานนท์ และอาจารย์ผู้สอนวิชา Digital Logic

ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ภาคการศึกษาต้น ปีการศึกษา 2568

บทที่ 1: บทนำ

1.1 ที่มาและความสำคัญของโครงงาน

โครงงานนี้เป็นส่วนหนึ่งของวิชา 2110252 Digital Logic ประจำปีการศึกษา 2568 โดยมีเป้าหมายหลักเพื่อให้นิสิตได้นำความรู้ทางด้านวงจรตรรกะ (Logic Circuit) และการออกแบบระบบดิจิทัลมาประยุกต์ใช้ในการสร้างหน่วยประมวลผลกลางขนาดเล็ก (Mini CPU) ที่สามารถทำงานได้จริงตามชุดคำสั่งที่กำหนด โครงงานนี้มุ่งเน้นการออกแบบและสร้าง CPU โดยใช้โปรแกรม Digital ซึ่งเป็นเครื่องมือจำลองการทำงานของวงจรดิจิทัล ทำให้นิสิตได้เรียนรู้กระบวนการออกแบบสถาปัตยกรรมคอมพิวเตอร์ตั้งแต่ระดับพื้นฐาน ทั้งในส่วนของหน่วยประมวลผลข้อมูล (Data Path) และหน่วยควบคุม (Control Unit) ซึ่งเป็นหัวใจสำคัญของระบบคอมพิวเตอร์ทุกชนิด

1.2 วัตถุประสงค์ของโครงงาน

- เพื่อออกแบบและสร้าง Mini CPU ที่สามารถทำงานตามชุดคำสั่งที่กำหนดได้อย่างถูกต้อง
- เพื่อทำความเข้าใจหลักการทำงานของสถาปัตยกรรมคอมพิวเตอร์แบบ Multiple Cycle หรือ Pipeline
- เพื่อฝึกฝนทักษะการใช้เครื่องมือออกแบบและจำลองวงจรดิจิทัล (โปรแกรม Digital)
- เพื่อเรียนรู้การออกแบบ Data Path และ Control Unit และการทำงานร่วมกันของทั้งสองส่วน
- เพื่อสามารถจัดการหน่วยความจำ (RAM) สำหรับโปรแกรมและข้อมูล และจัดการสัญญาณ Input/Output ตามข้อกำหนด
- เพื่อส่งเสริมการทำงานเป็นทีม การวางแผน และการแก้ปัญหาร่วมกัน

1.3 ขอบเขตของโครงงาน

Mini CPU ที่ออกแบบต้องมีคุณสมบัติและส่วนประกอบตามข้อกำหนดดังต่อไปนี้:

1) **Program RAM (pRAM): ขนาด 256 x 14 bits สำหรับจัดเก็บโปรแกรมที่ได้รับจาก Input**
2) **Result RAM (rRAM): ขนาด 256 x 8 bits สำหรับจัดเก็บผลลัพธ์ที่ได้รับจากคำนวณ**

ชื่อสัญญาณ	ขนาด (bits)	หน้าที่การทำงาน
`M (IN)`	8	พารามิเตอร์ที่ 1 สำหรับ CPU
`N (IN)`	8	พารามิเตอร์ที่ 2 สำหรับ CPU
`progIn (IN)`	14	ข้อมูลคำสั่งของโปรแกรมที่จะถูกโหลดเข้า pRAM
`reset (IN)`	1	เมื่อเป็น 1 ให้รีเซ็ตการทำงานของวงจรและเคลียร์ค่า rRAM ทั้งหมดเป็น 0
`progLoad (IN)`	1	เมื่อเป็น 1 ให้เริ่มอ่านค่าจาก `progIn` ไปเก็บใน pRAM ทีละคำสั่ง
`start (IN)`	1	เมื่อเป็น 1 ให้เริ่มประมวลผลโปรแกรมใน pRAM
`result (IN)`	1	เมื่อเป็น 1 ให้เริ่มส่งผลลัพธ์ที่เก็บใน rRAM ออกทาง output
`clk (IN)`	1	สัญญาณนาฬิกา (Positive Edge) สำหรับการทำงานแบบ Synchronous
`valid (OUT)`	1	เป็น 1 เมื่อประมวลผลโปรแกรมเสร็จสิ้น (เจอคำสั่ง STOP) และพร้อมส่งคำตอบ
`done (OUT)`	1	เป็น 1 เมื่อส่งผลคำตอบครบถ้วน หรืออยู่ในสถานะเริ่มต้น พร้อมรับคำสั่งใหม่
`output (OUT)`	8	ค่าผลลัพธ์จาก rRAM (ตำแหน่ง 0x00 - 0x0F) ที่แสดงผลหลังได้รับสัญญาณ `result`

ข้อกำหนดเพิ่มเติม

- ต้องออกแบบ CPU เป็นแบบ **Multiple Cycle CPU** หรือ **Pipeline CPU** (ไม่อนุญาตให้ทำเป็น Single Cycle CPU)
- การคำนวณทางคณิตศาสตร์ใช้ระบบเลขแบบ **2's complement** (ยกเว้นระบุเป็นอย่างอื่น)
- ใช้ **Positive Edge Clock** ในการ Synchronous วงจร
- ต้องมีการออกแบบ Register ภายใน (เช่น accA, accB, regC, regD) ขนาด 8 bits

1.4 ประโยชน์ที่คาดว่าจะได้รับ

เมื่อสิ้นสุดโครงงาน คาดว่าสมาชิกในกลุ่มจะได้รับประโยชน์ดังนี้:

- มีความเข้าใจอย่างลึกซึ้งเกี่ยวกับหลักการทำงานพื้นฐานของ CPU และสถาปัตยกรรมคอมพิวเตอร์
- สามารถออกแบบและสร้างวงจรดิจิทัลที่ซับซ้อนโดยใช้เครื่องมือมาตรฐานได้
- ได้รับประสบการณ์ในการแก้ปัญหาเชิงวิศวกรรม การดีบักวงจร และการทดสอบระบบ
- พัฒนาทักษะการทำงานร่วมกับผู้อื่น การแบ่งงาน และการบริหารจัดการโครงงานภายใต้ข้อจำกัดด้านเวลา

บทที่ 2: ภาพรวมและแนวคิดการออกแบบ

2.1 ภาพรวมสถาปัตยกรรม

1) เลือกสถาปัตยกรรม **Multiple Cycle** เพราะสอดคล้องข้อกำหนดโครงงานและยืดหยุ่นให้แต่ละคำสั่งใช้ **จำนวน clock cycle ต่างกัน**
2) ช่วยให้คำสั่งซับซ้อน (คูณ/หาร) **มีประสิทธิภาพดีกว่า Single Cycle** ที่ต้องรอช้าที่สุด
3) แบ่งการทำงานเป็นสถานะ: **Fetch → Decode → Execute → Memory Access → Write Back**
4) ใช้ **Control Unit แบบ FSM** คุมการไหลของสถานะและสร้าง **สัญญาณควบคุม** ให้ Datapath ในแต่ละช่วงงาน

2.2 สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture - ISA)

โครงสร้างคำสั่งของ Mini CPU นี้มีขนาด 14 bits แบ่งออกเป็น 2 ส่วนหลัก ดังนี้:

- **Opcode (6 bits):** บิตที่ 13 ถึง 8 ใช้สำหรับระบุประเภทของคำสั่ง
- **Operand (8 bits):** บิตที่ 7 ถึง 0 ใช้สำหรับระบุข้อมูลหรือตำแหน่งหน่วยความจำที่คำสั่งต้องการอ้างอิง

13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode (6 bits)						Operand (8 bits)							

2.3 ภาพรวมการทำงานของวงจร

การทำงานของ CPU สามารถแบ่งออกเป็น 4 เฟสหลักตามสัญญาณควบคุมที่ได้รับ ดังนี้:

1) Reset

• Reset (T24) =1 → เคลียร์ rRAM ทั้งหมดเป็น 0, รีเซ็ต PC และสถานะ FSM
• ตั้ง done=1 เพื่อบอกว่าพร้อมเริ่มงาน

2) Program Loading

• progLoad=1 → อ่านคำสั่งขนาด 14 บิตจาก progIn เข้าสู่ pRAM
• เริ่มที่แอดเดรส 0x00 แล้วเพิ่มทีละ 1 ต่อ clock จน progLoad=0

3) Execution

• รอ start=1 แล้วเริ่มรันจาก pRAM แอดเดรส 0x00
• ทำต่อเนื่องจนพบคำสั่ง STOP (opcode 111111)
• เมื่อจบ ตั้ง valid=1 เพื่อบอกว่าพร้อมส่งผลลัพธ์ 4.

4) Result Output

• รอ result=1 แล้วอ่าน rRAM ตั้งแต่ 0x00 ถึง 0x0F ออกทาง output ครั้งละ 8 บิต/clock
• ส่งครบ 16 ตำแหน่งแล้วตั้ง done=1 และกลับสู่สถานะพร้อมรับคำสั่งใหม่

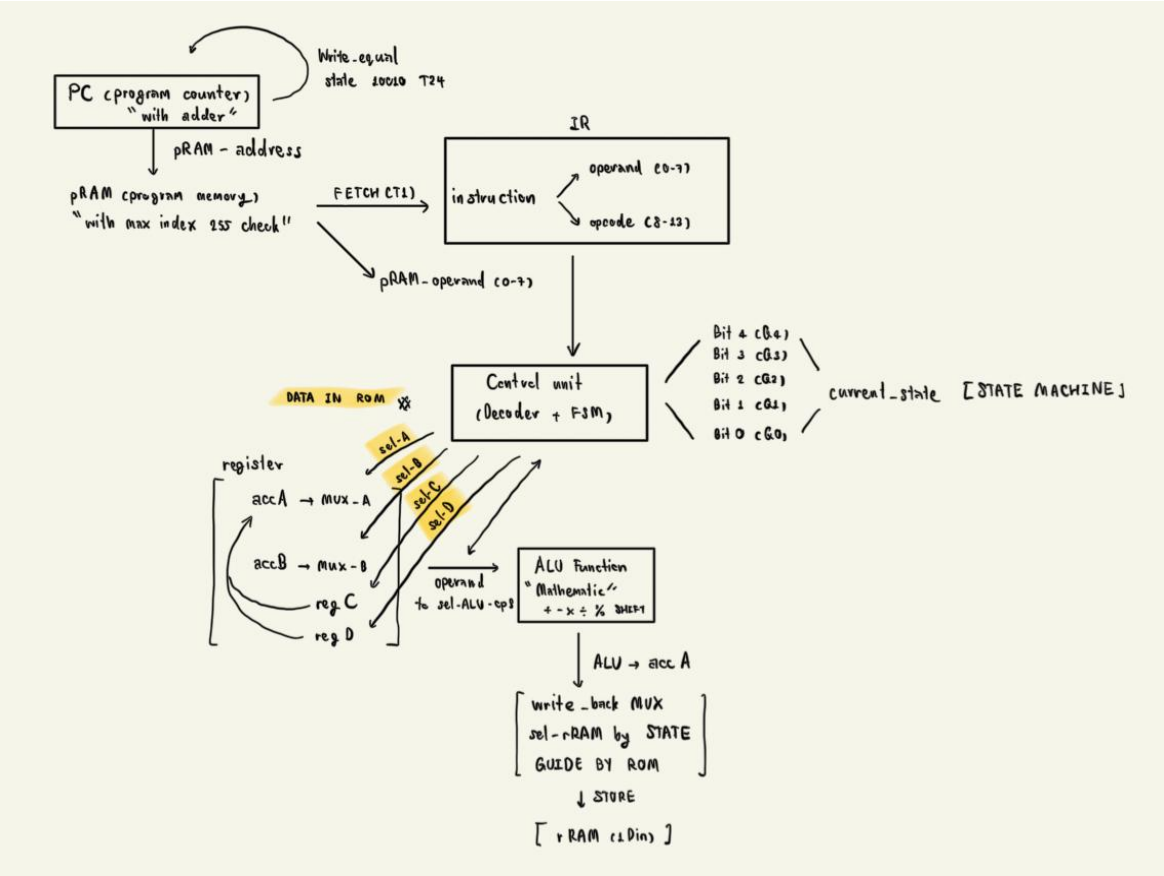
บทที่ 3: การออกแบบและพัฒนา

3.1 การออกแบบหน่วยประมวลผลข้อมูล (Data Path Design)

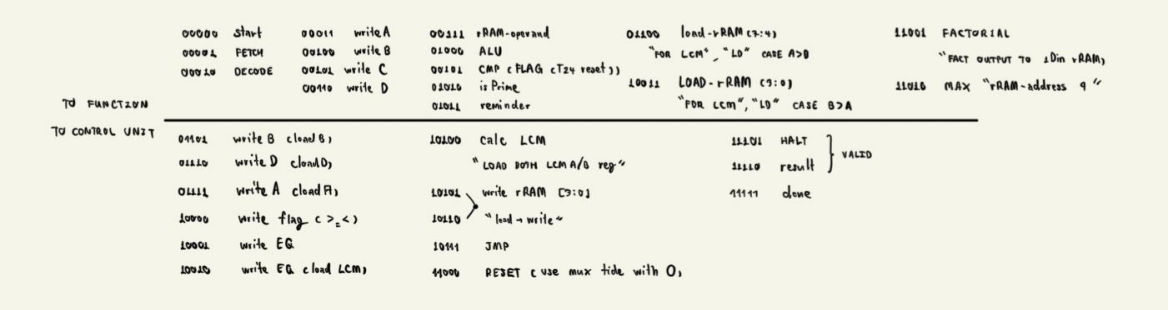
3.1.1 ส่วนประกอบหลัก (Main Components)

- **Program Counter (PC):** Register ขนาด 8 bits สำหรับชี้ตำแหน่งคำสั่งถัดไปใน pRAM
- **Instruction Register (IR):** Register ขนาด 14 bits `accA`, `accB`, `regC`, และ `regD`
- **ALU (Arithmetic Logic Unit):** หน่วยคำนวณและตรรกะ
- **Memory Units:** `pRAM` (256x14) และ `rRAM` (256x8) ตามที่ระบุในข้อกำหนด
- **Multiplexers (MUX):** การ reset ค่าให้ทัน clock ที่โจทย์กำหนด
- **Splitter/ Merger:** วงจรสำหรับขยายและย่อบิตของ Operand จาก IR เพื่อใช้เป็นข้อมูลหรือ Address

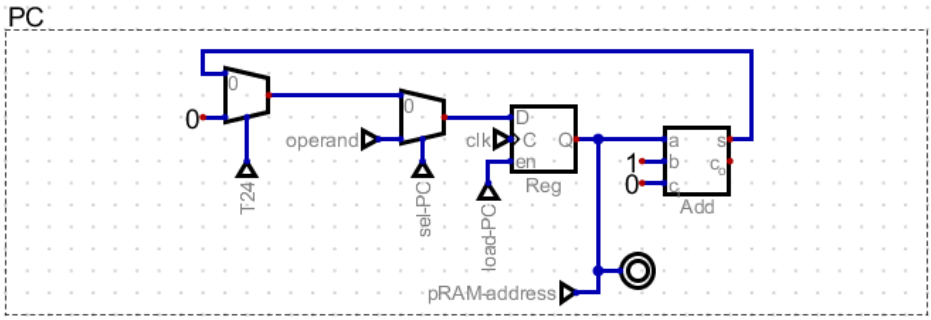
3.1.2 แผนภาพ Data Path (Data Path Diagram)



STATE-MACHINE (TO FUNCTION | TO CONTROL UNIT)



3.1.3 คำอธิบายการทำงานของ Data Path

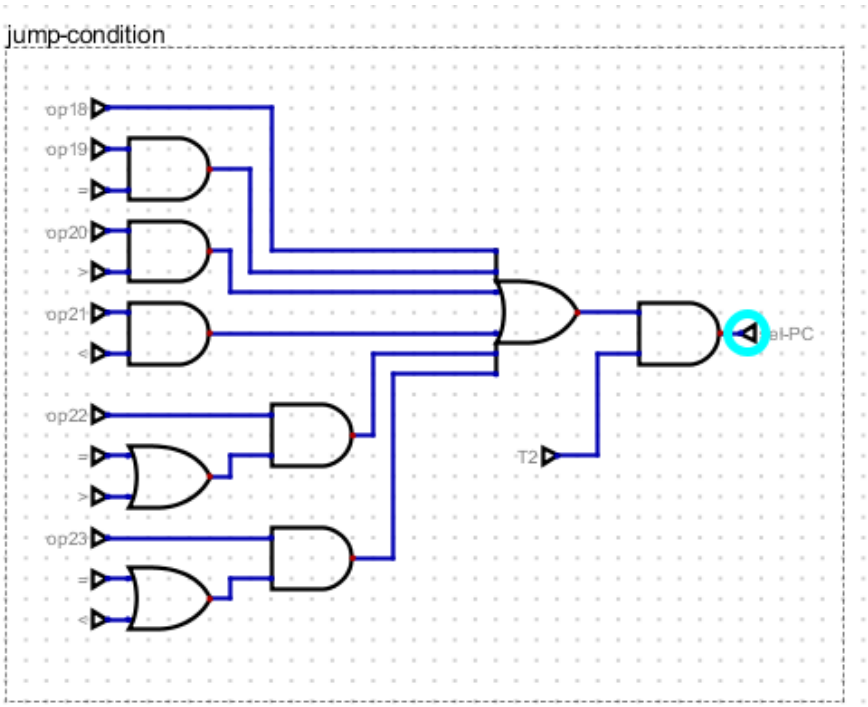


เริ่มจาก Program Counter ที่เป็นตัวโหลดคำสั่ง operand 8 bits, เราทำการ mux กับ T24 ซึ่งคือการ reset ทุกตำแหน่งบน rRAM, operand เป็นตัวขับเคลื่อนค่าตำแหน่งของ rRAM ที่เราจะเขียนข้อมูลลงไป

ส่วนที่เราจะเน้นคือ sel-PC ซึ่งคือการ or ของคำสั่ง JMP, การที่เราจะทำให้ JMP ค่าไม่เพี้ยนเราต้องคุมด้วย FLAG >,<,= ในแต่ละเคสจะไม่เหมือนกัน เป็นการเปรียบเทียบค่า accumulator A และ B

OPCODE 010010 คือการ JMP แบบไม่มีเงื่อนไข
010011 JMP ไปที่ operand ถ้าค่า accA, accB เท่ากัน
010100 Jump ไปที่ไปที่ Operand ถ้าค่า accA มากกว่า accB
010101 Jump ไปที่ Operand ถ้าค่า accA น้อยกว่า accB
010110 Jump ไปที่ Operand ถ้าค่า accA มากกว่าเท่ากับ accB
010111 Jump ไปที่ Operand ถ้าค่า accA น้อยกว่าเท่ากับ accB

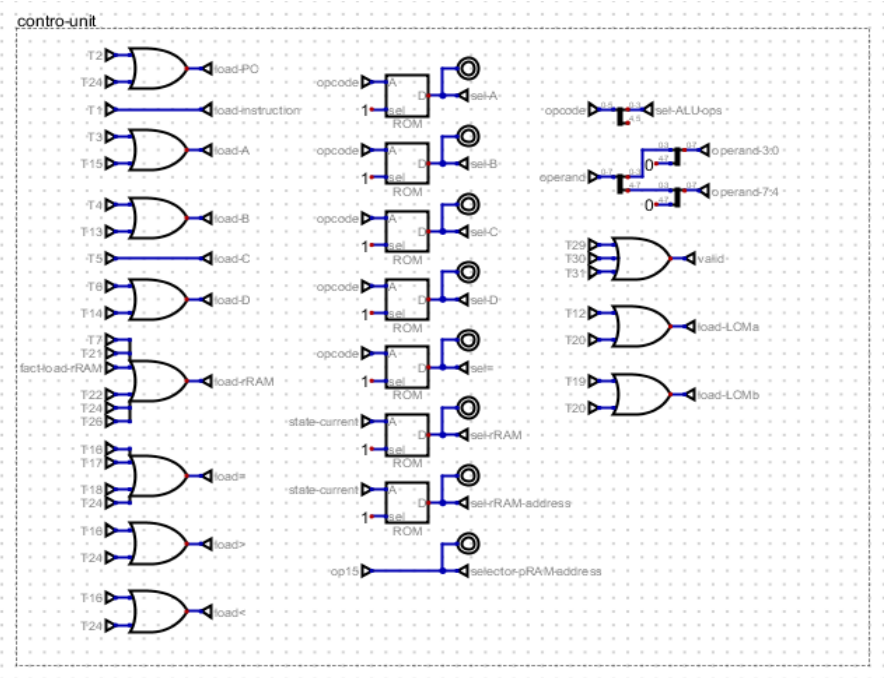
เราใช้ mux เพื่อคุมในกรณีที่ถูกต้องตามเงื่อนไข จะได้ย้ายไปที่ operand ที่ถูกต้อง



เรา and กับ opcode เพื่อทำให้ function ทำงานเมื่อมีคำสั่งเท่านั้น

3.2 การออกแบบหน่วยควบคุม (Control Unit Design)

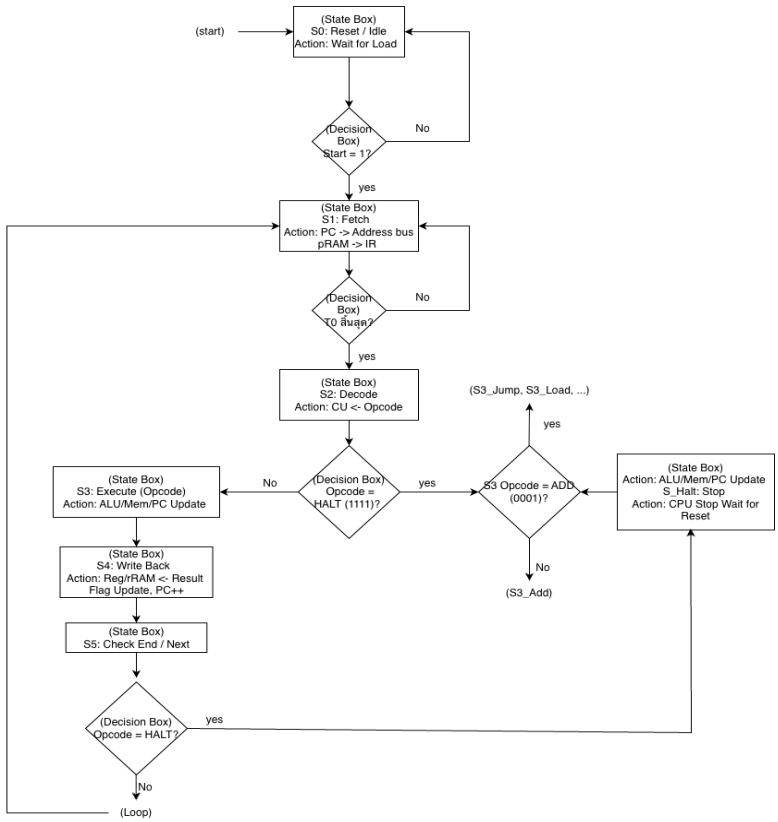
3.2.1 รูปแบบการออกแบบ



ในส่วนนี้เราจะนำข้อมูลจาก STATE MACHINE มาใช้คำนวณว่าเราจะ enable register ตัวไหน หรือจะทำการอ่านหรือส่งค่าไปที่ function ใดๆ

เราได้ใช้ rom ในการกำหนด selector เมื่อ opcode หรือ state current เข้ารูปแบบของแต่ละ mux เพื่อให้การทำงานเป็นระเบียบและ debug ได้สะดวก เราจึงใช้ output ในการดูค่าในแต่ละขั้นการทำงาน

3.2.2 แผนภูมิ ASM หรือ FSM (ASM or FSM Chart)



3.2.3 สัญญาณควบคุม (Control Signals)

ชื่อสัญญาณควบคุม	หน้าที่การทำงาน
`load-PC`	[เปิด/ปิด การเขียนค่าใหม่ลงใน Program Counter]
`load-instruction`	[เปิด-ปิดการแปลงค่า operand (0-7) และ opcode (8-13) จาก pRAM (2D)]
`load-A`	[เปิด/ปิด การเขียนค่าบน accumulatorA]
`load-B`	[เปิด/ปิด การเขียนค่าบน accumulatorB]
`load-C`	[เปิด/ปิด การเขียนค่าบน regC]
`load-C`	[เปิด/ปิด การเขียนค่าบน regD]
`load_=`	[เปิด/ปิด การเช็คค่า = บน Flag Register =]
`load_>`	[เปิด/ปิด การเช็คค่า > บน Flag Register >]
`load_<`	[เปิด/ปิด การเช็คค่า < บน Flag Register <]
`sel-A`	[เลือกค่า selector ที่ใช้เลือกค่าที่จะส่งให้ accumulatorA โดยอ่านค่าจาก ROM]
`sel-B`	[เลือกค่า selector ที่ใช้เลือกค่าที่จะส่งให้ accumulatorB โดยอ่านค่าจาก ROM]
`sel-C`	[เลือกค่า selector ที่ใช้เลือกค่าที่จะส่งให้ regC โดยอ่านค่าจาก ROM]
`sel-D`	[เลือกค่า selector ที่ใช้เลือกค่าที่จะส่งให้ regD โดยอ่านค่าจาก ROM]
`sel_=`	[เนื่องจาก Function และ opcode ที่ใช้คำสั่ง = มีถึง 3 กรณี เราให้ mux คุมการส่งค่า]
`sel-rRAM`	[เพื่อคุมว่าเราจะส่ง accA (ปกติ) หรือ LCM-A LCM-B ตามข้อมูลใน rom]
`sel-rRAM-address`	[คุม mux ที่ใช้ส่งค่า operand และ factorial เข้า 1A ของ rRAM]
`selector-pRAM-address`	[เลือกว่าจะใช้ค่าจาก progLoad Counter หรือ operand เข้า 1A ของ pRAM]
`sel-ALU-ops`	[ปรับ opcode เหลือ 4 bit แรก เพื่อใช้ในการเลือก ALU mux เมื่อ state ถูกต้อง]
`operand-3:0`	[ส่งค่า 4 bit แรกของ operand ตามด้วย 0000 เข้า mux pRAM-1A]
`operand-7:4`	[ส่งค่า 4 bit ท้ายของ operand ตามด้วย 0000 เข้า mux pRAM-1A]
`valid`	[ส่งค่า valid เมื่อ state คือ HALT, RESULT, DONE]
`load-LCMa`	[เปิด/ปิด การทำงานของ register LCMa]
`load-LCMb`	[เปิด/ปิด การทำงานของ register LCMb]

3.3 การจัดการสถานะและแฟล็ก (State and Flag Management)

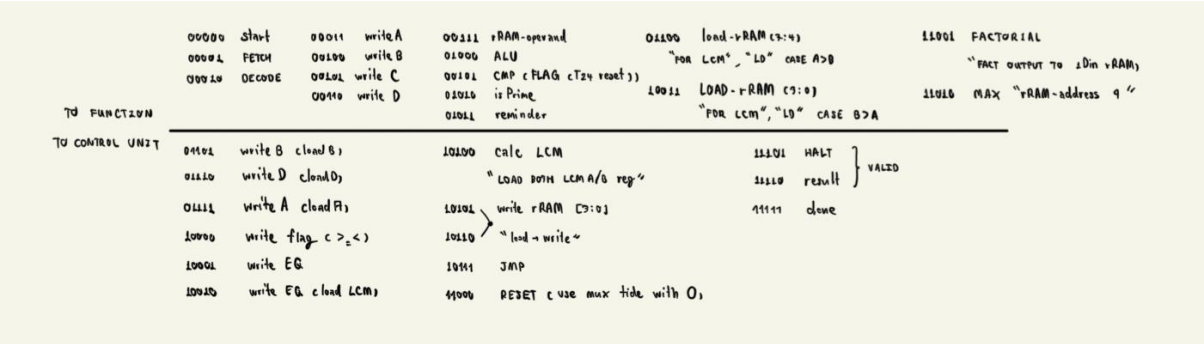
1) มี **Flag Register 3 บิต**: equal_flag, greater_flag, lesser_flag ใช้ชี้ผลการเปรียบเทียบ (เท่ากัน/มากกว่า/น้อยกว่า) เพื่อควบคุมคำสั่งแบบมีเงื่อนไข
2) คำสั่ง **CMP (accA CMP accB)** ให้ ALU ทำ **accA – accB** โดย **ไม่เก็บผลลัพธ์** แล้วตั้ง Flag ตามผล:

- ผลลัพธ์ = 0 → equal_flag = 1
- ผลลัพธ์ > 0 → greater_flag = 1
- ผลลัพธ์ < 0 → lesser_flag = 1

3) คำสั่งกระโดดตรวจแฟล็กเพื่ออัปเดต **PC**:

- **JEQ**: ถ้า equal_flag=1 → เปิด PCWrite=1 และโหลด PC จาก Operand ใน IR
- **JGT / JLT**: ตรวจ greater_flag / lesser_flag ตามลำดับ (กลไกเดียวกัน)

4) วงจรแฟล็ก **เชื่อมตรง** กับ ALU และ Control Unit ทำให้ตัดสินใจเชิงเงื่อนไขได้ **ทันที** และเปลี่ยนลำดับโปรแกรมได้มีประสิทธิภาพ
5) ฝั่ง **State Machine**: มีการ **map ค่าตามภาพ** โดย **แยกบิต** แล้ว **รวมภายหลัง** เพื่อควบคุมลอจิกของสถานะและสัญญาณที่เกี่ยวข้อง



บทที่ 4: การทดสอบและผลการดำเนินงาน

4.1 แผนการทดสอบ

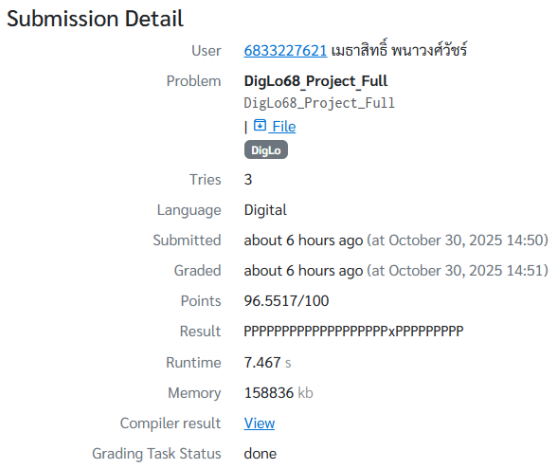
1) Unit Testing (ทดสอบรายโมดูล)

- **ALU:** ป้อนชุดทดสอบยืนยันผลของ **ADD/SUB/AND/CMP**
- **Register File:** ตรวจการอ่าน–เขียนของรีจิสเตอร์ **accA, accB** ว่าถูกตำแหน่ง/ไม่สูญหาย
- **Control Unit (FSM):** ป้อน **opcode** แล้วตรวจ **state transition** และ **control signals** ตรงตามสเปก
- **Memory Interface:** ตรวจ **MemRead/MemWrite** สื่อสารกับ **pRAM** และ **rRAM** ได้ถูกต้อง
- วิธีตรวจ: ใช้ **probe** และเอาต์พุตแบบไม่ระบุชื่อ เพื่อดูค่า **operand/opcode/mux** เป็นหลัก

2) Integration Testing (ทดสอบระบบรวม)

- **พื้นฐาน:** รันคำสั่งทั่วไป **LOAD/MOV/ADD/SUB** ตรวจลำดับ **fetch / execute**
- **หน่วยความจำ:** ทดสอบอ่าน–เขียน **pRAM** และ **rRAM**
- **ควบคุมการไหล:** ทดสอบ **jump** ทั้งมี/ไม่มีเงื่อนไข (เช่น **JEQ, JG, JL)** ตรวจการจัดการ **flag** อัปเดต **PC**
- **คำสั่งพิเศษ:** ตรวจวงจรเฉพาะ (เช่น **isPrime, LCM)** ว่าผลลัพธ์และสถานะถูกต้อง
- **ตาม Grader:** ใช้ไฟล์ **Testcasetemplate2568.xlsx** ตรวจการตอบสนองสัญญาณ **reset, progLoad, start, result** ภายใต้เงื่อนไขเวลาแตกต่างกัน

4.2 ผลการทดสอบและวิเคราะห์



จากการทดสอบ Multi process CPU ที่เราได้ออกแบบมาพบว่า ผลลัพธ์ก่อนหน้ามี bug คือ ram address เก็บมาเกิน 255 bits หรือที่เราเรียกว่า ram-overflow, เราจึงมีการวางแผนใหม่โดย ตรวจสอบว่าถึง pRAM-max-address หรือยัง ถ้าถึงแล้วให้หยุดการเพิ่ม operand ทำให้ bug ใน testcase ท้ายหายไป และเราได้คะแนนที่ 96.5517

Test Case ID	คำอธิบาย	ผลการทดสอบ (ผ่าน/ไม่ผ่าน)	หมายเหตุ (ถ้ามี)
01_P01T01	ทดสอบการโหลดโปรแกรมและการส่งค่าทั่วไป	[ผ่าน]	[ทำงานได้สมบูรณ์]
02_P01T02	Request result เป็นครั้งที่ 2	[ผ่าน]	[วงจรสามารถกลับสู่สถานะ Done และตอบสนองต่อสัญญาณ result ใหม่ได้]
03_P02T01	Test simple jump	[ผ่าน]	[คำสั่ง Jump ทำงานถูกต้อง]
04_P03T01	pRamLoad_Test	[ผ่าน]	[ทำงานได้สมบูรณ์]
05_P03T02	pRamLoad_Test	[ผ่าน]	[ทำงานได้สมบูรณ์]
06_P04T01	isPrime_Test	[ผ่าน]	[เช็คค่าจำนวนเฉพาะตั้งแต่ 0-255 ได้สมบูรณ์]
07_P05T01	isPrime + CMP	[ผ่าน]	[ทำงานได้สมบูรณ์]
08_P06T01	ทดสอบการอ่านค่า M และ N	[ผ่าน]	[ทำงานได้สมบูรณ์]
09_P06T02	ทดสอบการอ่านค่า M และ N / แล้วก็ทดสอบการ run ใหม่โดยไม่ได้โหลดโปรแกรมใหม่	[ผ่าน]	[ทำงานได้สมบูรณ์]
10_P07T01	ทดสอบคำสั่งพิเศษ LCM	[ผ่าน]	[หา ครน. ได้สมบูรณ์ทั้ง a>b และ b<a]

11_P07T02	ทดสอบคำสั่งพิเศษ LCM โดยให้ค่าคำตอบจะเกิน 8 บิต และเก็บตัวต้นไว้ที่ address 0x0F ซึ่งจะไว้เก็บคำตอบด้วย	[ผ่าน]	[splitter, merger ทำงานได้สมบูรณ์]
12_P08T01	ทดสอบการ + - * / % ^	[ผ่าน]	[ส่งค่าจาก ALU ทำงานได้สมบูรณ์]
13_P08T02	ทดสอบการ + - * / % ^ โดยการสั่งแก้บางส่วนของโปรแกรมแล้ว ทำงานเลย	[ผ่าน]	[ส่งค่าจาก ALU ทำงานได้สมบูรณ์]
14_P09T01	ทดสอบ bitwise operation NOT AND OR XOR <<	[ผ่าน]	[ส่งค่าจาก ALU ทำงานได้สมบูรณ์]
15_P09T02	ทดสอบ bitwise operation NOT AND OR XOR << แล้วก็ reset ระหว่างทำงาน	[ผ่าน]	[ส่งค่าจาก ALU ทำงานได้สมบูรณ์ reset สำเร็จ]
16_P01T03	ทดสอบการโหลดโปรแกรมและการส่งค่า ทั่วไป แล้วก็ทดลอง request result ซ้ำ ทดลองใช้ regC regD	[ผ่าน]	[ทำงานได้สมบูรณ์]
17_P02T02	Jumpทุกรูปแบบ และมีการหยุดระหว่างทาง	[ผ่าน]	[ทำงานได้สมบูรณ์ sel-PC ทำงานได้]
18_P03T03	pRamLoad_Test extend	[ผ่าน]	[ทำงานได้สมบูรณ์]
19_P04T02	isPrime_Test Extend	[ผ่าน]	[ทำงานได้สมบูรณ์ อ่านค่าที่เราลงไว้ใน rom]
20_P04T03	accB | aacA ทดสอบว่า accA หารด้วย accB ลงตัวหรือไม่	**[ไม่ผ่าน]**	[ติด bug ขึ้นค่า 14 ที่ output เมื่อ stop (OP63)]
21_P05T02	isPrime_Test + flag condition	[ผ่าน]	[ทำงานได้สมบูรณ์]
22_P06T03	ทดสอบการอ่านค่า M และ N ใส่ regC regD	[ผ่าน]	[ทำงานได้สมบูรณ์]
23_P07T03	ทดสอบคำสั่งพิเศษ LCM	[ผ่าน]	[ทำงานได้สมบูรณ์]
24_P08T03	ทดสอบการ + - * / % ^ โดยการสั่งแก้บางส่วนของโปรแกรมแล้ว ทำงานเลย	[ผ่าน]	[ทำงานได้สมบูรณ์]
25_P09T03	ทดสอบ bitwise operation NOT AND OR XOR << <<< แล้วก็ reset ระหว่างทำงาน	[ผ่าน]	[ทำงานได้สมบูรณ์ reset ทำงานบน rRAM]
26_P10T01	ทดสอบคำสั่งพิเศษ factorial	[ผ่าน]	[ทำงานได้สมบูรณ์]
27_P10T02	ทดสอบคำสั่งพิเศษ factorial	[ผ่าน]	[ทำงานได้สมบูรณ์]
28_P11T01	ทดสอบคำสั่งพิเศษ max	[ผ่าน]	[ทำงานได้สมบูรณ์]
29_P11T02	ทดสอบคำสั่งพิเศษ max	[ผ่าน]	[ทำงานได้สมบูรณ์]

บทที่ 5: สรุปและข้อเสนอแนะ

5.1 สรุปผลการดำเนินงาน

กลุ่มของเราประสบความสำเร็จในการออกแบบและสร้าง Mini CPU แบบ Multiple Cycle ตามข้อกำหนดของโครงงาน CPU ที่สร้างขึ้นสามารถประมวลผลชุดคำสั่งที่กำหนด, จัดการหน่วยความจำ, และตอบสนองต่อสัญญาณควบคุมภายนอกได้อย่างถูกต้อง ซึ่งยืนยันได้จากผลการทดสอบที่ผ่าน Test Case ของ Grader ได้ 96.5517 โครงงานนี้ทำให้เราได้เรียนรู้และเข้าใจกระบวนการออกแบบสถาปัตยกรรมคอมพิวเตอร์อย่างเป็นรูปธรรม ตั้งแต่การออกแบบ Data Path, การสร้าง Control Unit ด้วย FSM, ไปจนถึงการทดสอบและดีบักระบบที่ซับซ้อน

5.2 ปัญหาและอุปสรรค

- ความซับซ้อนของ Control Unit: ในช่วงแรก การออกแบบ FSM สำหรับคำสั่งทั้งหมดทำได้ยากและเกิดข้อผิดพลาดบ่อยครั้ง เราแก้ไขโดยการแบ่ง FSM ออกเป็นส่วนย่อยๆ ตามประเภทของคำสั่ง (เช่น ALU, Memory, Branch) และใช้สถานะหลัก (Fetch, Decode) ร่วมกัน
- Timing Issues: พบปัญหาการเขียนและอ่านข้อมูลจาก Register/Memory ไม่ทันใน 1 Clock Cycle สำหรับบางเส้นทาง เราแก้ไขโดยการเพิ่มสถานะ (State) เข้าไปใน FSM เพื่อให้มีเวลาเพียงพอสำหรับข้อมูลที่จะเดินทางไปถึงปลายทางก่อน Clock Edge ถัดไป
- การดีบักคำสั่งที่ซับซ้อน: คำสั่งอย่าง LCM หรือ isPrime ต้องการวงจรย่อยที่ซับซ้อน การดีบักทำได้ยาก เราใช้วิธีการแสดงผลค่า Register และสถานะภายในต่างๆ ออกมายัง Probe ในโปรแกรม Digital เพื่อติดตามการทำงานทีละขั้นตอน

5.3 ข้อเสนอแนะสำหรับการพัฒนาในอนาคต

- การเปลี่ยนไปใช้สถาปัตยกรรม **Pipeline:** เพื่อเพิ่มความเร็วในการประมวลผล สามารถพัฒนาต่อยอด **CPU** นี้ให้เป็นสถาปัตยกรรมแบบ **Pipeline 5** ขั้นตอน (**IF, ID, EX, MEM, WB)** ซึ่งจะต้องมีการจัดการกับปัญหา **Data Hazard** และ **Control Hazard** เพิ่มเติม
- การเพิ่มชุดคำสั่ง: สามารถเพิ่มคำสั่งที่มีประโยชน์มากขึ้น เช่น คำสั่งสำหรับการจัดการ **Stack (Push, Pop)** หรือคำสั่งสำหรับการคูณ/หารเลขที่มีขนาดใหญ่ขึ้น
- การจัดการ **Interrupt:** เพิ่มความสามารถในการจัดการ **Interrupt** เพื่อให้ **CPU** สามารถตอบสนองต่อเหตุการณ์ภายนอกได้โดยไม่ต้องรอให้โปรแกรมจบ
- การปรับปรุง **ALU:** ออกแบบ **ALU** ให้มีประสิทธิภาพมากขึ้น เช่น การใช้วงจร **Carry-Lookahead Adder** เพื่อลดดีเลย์ในการบวกเลข
- ลดการใช้ **opcode** เยอะเกินความจำเป็น
- เปลี่ยนวิธีการใช้ **stage machine** จาก **5 mux** เป็น **3 mux**