Tab 1

🔹 รายละเอียดงานส่วนที่ 1: Assembler

เป้าหมาย: เขียนโปรแกรมที่อ่าน assembly language → แปลงเป็น machine code (decimal 32-bit integers) + ตรวจจับ error

1️⃣ ขั้นตอนการทำงานของ Assembler

Assembler ควรทำงานแบบ **2-pass assembler**

Pass 1: สร้าง Symbol Table

- อ่านไฟล์ assembly ทีละบรรทัด
- เก็บ **label** (ถ้ามี) กับ **address** (line number) ลงใน symbol table
- ตรวจสอบ duplicate label → error
- address เริ่มนับจาก 0 และเพิ่มทีละ 1 ต่อบรรทัด

Pass 2: Generate Machine Code

- อ่าน assembly อีกรอบ
- แยก fields: instruction, field0, field1, field2
- แปลงเป็น binary ตาม instruction format (R, I, J, O)
- กรณี `.fill`:
    - ถ้าเป็นตัวเลข → แปลงตรงๆ
    - ถ้าเป็น symbolic label → เอาค่า address ของ label จาก symbol table
- กรณี **I-type (lw, sw, beq)**:
    - ถ้า field2 เป็น label → ต้องคำนวณ offset:
        - lw/sw → address ของ label - ค่าใน regA (ปกติใช้ reg0 = 0 ทำ base addressing)

        - beq → offset = labelAddress – (currentInstructionAddress + 1)
    - ต้องเช็ค offset ว่าอยู่ในช่วง -32768 ถึง 32767 (16-bit signed)

2️⃣ Error Checking ที่ต้องทำ

1. **ใช้ labels ที่ไม่มีใน symbol table** → error
2. **มี label ซ้ำกัน** → error
3. **offsetField เกิน 16-bit signed range** → error
4. **opcode ไม่ถูกต้อง** → error
5. **register index นอกช่วง 0–7** (optional แต่ควรมี)

ถ้ามี error → `exit(1)`
ถ้าสำเร็จ → print machine code ทีละบรรทัด และ `exit(0)`

3️⃣ โครงสร้างการเขียนโค้ด

สามารถแบ่งเป็นโมดูลย่อยดังนี้:

- **parser.c**
    - ฟังก์ชัน `readAndParse()` → อ่าน 1 บรรทัด, คืน label/instruction/fields
- **symbolTable.c**
    - ฟังก์ชัน `addLabel(label, address)`
    - ฟังก์ชัน `findLabel(label)` → คืนค่า address
- **encoder.c**
    - ฟังก์ชัน `encodeR(opcode, regA, regB, destReg)`

- ฟังก์ชัน `encodeI(opcode, regA, regB, offset)`
- ฟังก์ชัน `encodeJ(opcode, regA, regB)`
- ฟังก์ชัน `encodeO(opcode)`

- **main.c**
  - pass1 → สร้าง symbol table
  - pass2 → encode instructions → print machine code

4️⃣ ลำดับการทำงานที่ควรทำ

1. **อ่าน requirement ISA** ให้เข้าใจ (ทุกคนในทีมควรอ่าน)
2. **Implement Pass 1**
   - อ่านบรรทัด, แยก label + instruction
   - สร้าง symbol table (map จาก label → address)
   - ตรวจ error label ซ้ำ
3. **Implement Pass 2**
   - อ่านบรรทัดอีกครั้ง
   - ตรวจว่า instruction ถูกต้อง
   - ถ้าเป็น `.fill` → resolve ค่า (number/label)
   - ถ้าเป็น I-type (beq, lw, sw) → resolve offset และ check range
   - encode เป็น 32-bit integer
4. **เพิ่ม Error Handling**
   - ตรวจ labels ที่ undefined

    - ตรวจ offset เกิน 16-bit
    - ตรวจ opcode นอกตาราง
5. **ทดสอบกับ example program** ที่ให้มาในโจทย์
    - assembly นับจาก 5 ถึง 0
    - ควรได้ output machine code ตรงกับโจทย์
6. **ทดสอบเพิ่มเติม**
    - `.fill` แบบตัวเลขและ symbolic address
    - error cases (label ซ้ำ, label ไม่ define, offset เกิน range, opcode ไม่ถูกต้อง)

5️⃣ ระดับความยาก

- **Parsing (ปานกลาง)** → ต้องระวัง white space, comments
- **Symbol table (ง่าย)** → แค่เก็บคู่ label–address
- **Instruction encoding (ยาก)** → ต้องเข้าใจ format, bit shifting
- **Offset calculation (ยาก)** → โดยเฉพาะ beq (relative jump)
- **Error handling (ปานกลาง)** → logic ชัดเจน แต่ต้องเช็คทุกกรณี

Tab 2

🔹 ส่วนที่ 2: Behavioral Simulator

เป้าหมาย: เขียนโปรแกรมที่รับ **machine code (decimal 32-bit integers)** จาก assembler แล้ว simulate การทำงานของเครื่อง SMC จนกระทั่งเจอ `halt`

1️⃣ Input & Initialization

- Input: ไฟล์ machine code (แต่ละบรรทัดเป็นเลขฐานสิบ)
- Memory: array ของ integers ขนาด **65536** (word-addressable)
    - เก็บเฉพาะบรรทัดที่มีในไฟล์ (เช่น 10 บรรทัด → memory[0..9])
- Registers: array 8 ตัว (`reg[0..7]`) → initialize เป็น 0
- PC (Program Counter): เริ่มที่ 0
- numMemory = จำนวนบรรทัดใน machine code file

2️⃣ Loop ของ Simulation

ทำงานแบบนี้:

```
while (true):
    printState()
    instr = memory[PC]
    decode instr
    execute instr
    if halt: break
```

- **printState** (ให้ตาม requirement):
    - แสดงค่า PC

- แสดงค่า registers ทั้ง 8
- แสดง memory[0..numMemory-1]

3️⃣ การ Decode Instruction

Format ตาม ISA:

- Bits 24–22 → opcode
- Bits 21–19 → regA
- Bits 18–16 → regB
- ที่เหลือขึ้นกับ instruction type

R-type (add, nand)

```
000 add: reg[rd] = reg[regA] + reg[regB]
001 nand: reg[rd] = ~(reg[regA] & reg[regB])
```

- rd อยู่ที่ bits 2–0

I-type (lw, sw, beq)

```
010 lw: reg[regB] = memory[regA + offset]
011 sw: memory[regA + offset] = reg[regB]
100 beq: if (reg[regA] == reg[regB]) PC = PC + 1 + offset
```

- offset = sign-extended 16-bit

J-type (jalr)

```
101 jalr:
    reg[regB] = PC + 1
    PC = reg[regA]
```

O-type (halt, noop)

```
110 halt: stop simulation
111 noop: do nothing
```

4️⃣ Sign Extension (สำคัญมาก)

I-type offset ต้องแปลงจาก 16-bit signed → 32-bit integer

```
int convertNum(int num) {
    if (num & (1 << 15)) {  // ถ้า bit 15 เป็น 1 → negative
        num -= (1 << 16);
    }
    return num;
}
```

5️⃣ ข้อควรระวัง

- **reg[0] ต้องเป็น 0 เสมอ** (ห้ามเปลี่ยนค่า)
- **PC update**:
    - ปกติ PC++ หลัง execute
    - ยกเว้น instruction ที่เปลี่ยน PC (beq, jalr)
- **Memory bound**: ต้องไม่เกิน 65536
- **PrintState**: เรียก 1 ครั้งก่อน execute, และ 1 ครั้งก่อนออกจากโปรแกรม

6️⃣ Error Handling

Simulator ไม่ต้อง detect errors เช่น infinite loop หรือ invalid address (ตาม requirement)
แต่ควร handle กรณี out-of-bounds access (optional safety check)

7️⃣ โครงสร้างโค้ด (Pseudocode)

```
struct State {
    int pc;
    int mem[65536];
    int reg[8];
    int numMemory;
};

void printState(State *state) {
    // แสดงค่า pc, register, memory (0..numMemory-1)
}

int main() {
    State state;
    initialize(&state);

    while (true) {
        printState(&state);

        int instr = state.mem[state.pc];
        int opcode = (instr >> 22) & 0x7;
        int regA   = (instr >> 19) & 0x7;
        int regB   = (instr >> 16) & 0x7;

        switch (opcode) {
            case 0: // add
                {
                    int rd = instr & 0x7;
                    state.reg[rd] = state.reg[regA] + state.reg[regB];
                }
                state.pc++;
                break;

            case 1: // nand
                {
                    int rd = instr & 0x7;
```

```
                    state.reg[rd] = ~(state.reg[regA] &
state.reg[regB]);
                }
                state.pc++;
                break;

            case 2: // lw
                {
                    int offset = convertNum(instr & 0xFFFF);
                    state.reg[regB] = state.mem[state.reg[regA] +
offset];
                }
                state.pc++;
                break;

            case 3: // sw
                {
                    int offset = convertNum(instr & 0xFFFF);
                    state.mem[state.reg[regA] + offset] =
state.reg[regB];
                }
                state.pc++;
                break;

            case 4: // beq
                {
                    int offset = convertNum(instr & 0xFFFF);
                    if (state.reg[regA] == state.reg[regB]) {
                        state.pc = state.pc + 1 + offset;
                    } else {
                        state.pc++;
                    }
                }
                break;

            case 5: // jalr
                {
                    int tmp = state.pc + 1;
```

```
                state.pc = state.reg[regA];
                state.reg[regB] = tmp;
            }
            break;

        case 6: // halt
            printf("machine halted\n");
            printState(&state);
            exit(0);

        case 7: // noop
            state.pc++;
            break;
    }

    state.reg[0] = 0; // enforce $0=0
  }
}
```

8️⃣ ระดับความยาก

- **ง่ายกว่า assembler** เพราะ logic ชัดเจนตาม ISA
- จุดยากอยู่ที่:
  - การ decode bit-field ของ instruction
  - การจัดการ offset (sign extension)
  - การทำ jalr ให้ถูก (เก็บ return address ก่อนเปลี่ยน PC)
- ใช้เวลา debug เยอะ (เพราะถ้า offset คลาดนิดเดียว จะ jump ผิด address)

Tab 3

🔹 ส่วนที่ 3: Disassembler

เป้าหมาย:

- อ่าน **machine code (decimal 32-bit integers)**
- แปลงกลับไปเป็น **assembly code** ที่มนุษย์อ่านออก (ตามรูปแบบที่ assembler ใช้)
- เขียนผลลัพธ์ออกเป็นไฟล์ text

1️⃣ Input & Output

- **Input**: ไฟล์ที่มีเลขฐานสิบ 32 บิต (หนึ่งคำสั่งต่อบรรทัด)
- **Output**: ไฟล์ assembly แต่ละบรรทัด 1 คำสั่ง

ตัวอย่าง:

```
59962371
-1409286144
```

→ disassemble →

```
lw 0 1 123
beq 0 0 end
```

2️⃣ ขั้นตอนการทำงาน

Step 1: อ่านไฟล์ machine code

- โหลดทุกบรรทัดเป็น integer 32-bit
- เก็บใน array memory[]

Step 2: ถอดรหัส (Decode)

ใช้ format เดียวกับ Simulator:

R-type (opcode 000, 001)

```
 add regA regB destReg
nand regA regB destReg
```

-

I-type (opcode 010, 011, 100)

```
 lw regA regB offset
sw regA regB offset
beq regA regB offset
```

-

J-type (opcode 101)

```
 jalr regA regB
```

-

O-type (opcode 110, 111)

```
 halt
noop
```

-

  หมายเหตุ: offset ต้อง **sign-extend 16-bit** ด้วยฟังก์ชันเดียวกับ simulator

Step 3: Label resolution (optional enhancement)

- เวลามี **beq offset** → อาจเขียนเป็น label แทน เช่น `beq 1 2 loop`
- แต่ minimum requirement อาจไม่ต้อง resolve label → แค่แสดง offset ตรง ๆ ก็พอ

3️⃣ โครงสร้างโค้ด (Pseudocode)

```
int main() {
    int instr, opcode, regA, regB, destReg, offset;
    while (read next instr) {
        opcode = (instr >> 22) & 0x7;
        regA   = (instr >> 19) & 0x7;
        regB   = (instr >> 16) & 0x7;

        switch (opcode) {
            case 0: // add
                destReg = instr & 0x7;
                printf("add %d %d %d\n", regA, regB, destReg);
                break;

            case 1: // nand
                destReg = instr & 0x7;
                printf("nand %d %d %d\n", regA, regB, destReg);
                break;

            case 2: // lw
                offset = convertNum(instr & 0xFFFF);
                printf("lw %d %d %d\n", regA, regB, offset);
                break;

            case 3: // sw
                offset = convertNum(instr & 0xFFFF);
                printf("sw %d %d %d\n", regA, regB, offset);
                break;

            case 4: // beq
                offset = convertNum(instr & 0xFFFF);
                printf("beq %d %d %d\n", regA, regB, offset);
                break;

            case 5: // jalr
                printf("jalr %d %d\n", regA, regB);
                break;

            case 6: // halt
```

```
                printf("halt\n");
                break;

            case 7: // noop
                printf("noop\n");
                break;
        }
    }
}
```

---

4 จุดยาก

- ต้อง decode bit-field ให้ถูกต้องเหมือน simulator
- ต้องระวัง sign extension ของ offset
- ถ้าทำ optional label resolution → ต้อง scan memory 2 รอบ (รอบแรกสร้างตาราง label, รอบสองพิมพ์คำสั่งพร้อม label)

---

5 ระดับความยาก

- ง่ายที่สุดใน 3 ส่วน เพราะเป็นการ **ถอดรหัสตรงไปตรงมา**
- ใช้เวลาน้อยกว่า assembler และ simulator มาก
- เหมาะให้เป็นงานของคนที่ถนัด **string formatting** หรือคนที่มีเวลาน้อยสุดในทีม

Tab 4

ขอโทดค้าบ