Project 1 Computer Architecture (CPE 261304)

Due Date 13 ตุลาคม 2567 เวลา 23.00 น.

นัดตรวจโปรแกรมในวันจันทร์ที่ 14 ตุลาคม 2567 เวลา 13.30 เป็นต้นไป

ให้ส่งรายงานผ่านระบบ exam.cmu.ac.th เท่านั้น และ 1 กลุ่ม ให้ส่ง 1 file pdf โดยส่งผ่าน 1 คนที่เป็นสมาชิกในกลุ่ม

Problem

Project นี้มีทั้งหมด 3 ส่วนดังนี้

1. เขียน program ที่รับ assembly language program และ สร้าง machine language ตาม assembly program นั้น
2. เขียน behavioral simulator สำหรับ machine code นั้น
3. เขียน assembly language program สั้นๆเกี่ยวกับ การคูณของเลข 2 จำนวน และการทำ recursive function

Instruction Set Architecture

สมมุติว่าเครื่องคอมพิวเตอร์ (SMC) นี้ เป็น 8-register, 32-bit computer ใช้ word-addresses สำหรับทุก addresses และเครื่อง SMC มี 65536 words ของ memory โดยหลักการพื้นฐาน ของ assembly language จะ ตั้งให้ register 0 มีค่าข้างในเป็น 0 เสมอ (ไม่ควรจะเปลี่ยนเป็นค่าอื่น) เราจะใช้ instruction formats ทั้งหมด 4 formats โดยที่ bit ที่ 0 เป็น LSB และ bit ที่ 31-25 ไม่ถูกใช้สำหรับทุก instructions และควรจะมีค่าเป็น 0

R-type instructions (add, nand)
- Bits 24-22 opcode
- Bits 21-19 reg A (rs)
- Bits 18-16 res B (rt)
- Bits 15-3 ไม่ใช้ (ควรตั้งไว้ที่ 0)
- Bits 2-0 destReg (rd)

I-type instructions (lw, sw, beq)
- Bits 24-22 opcode
- Bits 21-19 reg A (rs)
- Bits 18-16 reg B (rt)
- Bits 15-0 offsetField (เลข16-bit และเป็น 2’s complement โดยอยู่ในช่วง –32768 ถึง 32767)

J-Type instructions (jalr)
- Bits 24-22 opcode
- Bits 21-19 reg A (rs)
- Bits 18-16 reg B (rd)
- Bits 15-0 ไม่ใช้ (ควรตั้งไว้ที่ 0)

O-type instructions (halt, noop)
- Bits 24-22 opcode
- Bits 21-0 ไม่ใช้ (ควรตั้งไว้ที่ 0)

ตาราง 1 Description of Machine Instructions

Assembly Language (name of instruction)	Opcodes in binary (bits 24,23,22)	Action
add (R-type format)	000	บวก ค่าใน regA ด้วยค่าใน regB และเอาไปเก็บใน destReg
nand (R-type format)]	001	Nand ค่าใน regA ด้วยค่าใน regB และเอาค่าไปเก็บใน destReg
lw (I-type format)	010	Load regB จาก memory และ memory address หาได้จากการเอา offsetField บวกกับค่าใน regA
sw (I-type format)	011	Store regB ใน memory และ memory address หาได้จากการเอา offsetField บวกกับค่าใน regA
beq (I-type format)	100	ถ้า ค่าใน regA เท่ากับค่าใน regB ให้กระโดดไปที่ address PC+1+offsetField ซึ่ง PC คือ address ของ beq instruction
jalr (J-type format)	101	เก็บค่า PC+1 ไว้ใน regB ซึ่ง PC คือ address ของ jalr instruction และกระโดดไปที่ address ที่ถูกเก็บไว้ใน regA แต่ถ้า regA และ regB คือ register ตัวเดียวกัน ให้เก็บ PC+1 ก่อน และค่อยกระโดดไปที่ PC+1

halt (O-type format)	110	เพิ่มค่า PC เหมือน instructions อื่นๆ และ halt เครื่อง นั่นคือให้ simulator รู้ว่าเครื่องมีการ halted เกิดขึ้น
noop (O-type format)	111	ไม่ทำอะไรเลย

Assembly language และ assembler

ส่วนแรกของ project คือการอ่าน assembly language และแปลงให้เป็น machine code ยกตัวอย่างเช่น คำสั่ง beq ก็จะถูกแปลงไปเป็น เลข 100 (opcode) และส่วนที่เป็น ชื่อ symbolic ก็จะถูกแปลงไปเป็น ตัวเลข และผลสุดท้ายจะได้ 32-bit instructions (ใน project นี้ bits ที่ 31-25 จะถูก ตั้งไว้ที่ 0) Format ของการเขียน assembly code แสดงข้างล่าง (หมายเหตุ <white> คือ tabs และ/หรือ ช่องว่าง)

label<white>instruction<white>field0<white>field1<white>field2<white>comments

ส่วนซ้ายสุดคือ label field ซึ่งจะมีได้มากที่สุด 6 ตัวอักษรและประกอบด้วยตัวหนังสือและตัวเลขแต่จะต้องเริ่มด้วยตัวหนังสือ โดยปกติจะมีหรือไม่มี Label ก็ได้ หลังจากนั้นก็เป็น white space ซึ่งตามด้วย instruction field ซึ่งจะเป็น instruction ที่อยู่ใน table 1 หลังจากนั้นจะเป็น white space และ fleids ซึ่ง field เหล่านี้จะเป็น ตัวเลข decimal หรือ label จำนวนของ fields จะขึ้นอยู่กับ ชนิดของ instruction ส่วน filed ที่ไม่ถูกใช้ให้คิดว่าเป็น comment

R-type instructions (add, nand) มี 3 fields (field0 คือ regA, field1 คือ regB, field2 คือ destReg)

I-type instructions (lw, sw, beq) มี 3 fields (field0 คือ regA, field1 คือ regB, field2 เป็น ค่าตัวเลขสำหรับ offsetField ซึ่งเป็นได้ทั้ง บวกหรือลบ หรือ symbolic address ซึ่งจะกล่าวถึงข้างล่าง

J-type instructions (jalr) มี 2 fields (field0 คือ regA, field1 คือ regB)

O-type instruction (noop, halt) ไม่มี field

Symbolic address คือ label และassembler จะคำนวณ offsetField โดยให้เท่ากับ address ของ label สำหรับ lw, sw instructions การทำเช่นนี้จะทำได้โดยใช้ zero-base register ในการอ้างถึง label หรือ ใช้ non zero-base register ในการชี้ไปที่ array ที่เริ่มที่ label ส่วน beq instruction assembler จะแปลง label ให้เป็นตัวเลข offsetField เพื่อบอกถึงที่อยู่ที่จะกระโดดไปที่ label นั้น

ส่วนที่เป็น comment จะมี end of line บอกว่า หมดส่วนนี้แล้ว

อีกคำสั่งที่ควรจะมีคือ .fill คำสั่งนี้จะบอก assembler ให้ใส่ตัวเลขในที่ที่ instruction ควรจะอยู่ และคำสั่งนี้ใช้ 1 field นั่นคือ ตัวเลขหรือ symbolic address ตัวอย่างเช่น ".fill 32" คือการที่เอาเลข 32 ไปไว้ที่ที่ instruction ควรจะอยู่ ส่วน .fill ที่ใช้ symbolic address จะเก็บค่า address ของ label เช่น ตัวอย่างข้างล่าง ".fill start" จะเก็บ ค่า 2 เพราะ label "start" อยู่ที่ address 2

Assembler ควรจะคำนวณทุก symbolic address ก่อนและสมมุติว่า instruction แรก อยู่ที่ address 0 และมันจะสร้าง machine code (เลขฐาน 10) สำหรับทุกบรรทัดใน assembly language ตัวอย่างเช่น assembly language program ที่ นับเลขจาก 5 จนถึง 0

```
        lw      0       1       five    load reg1 with 5 (uses symbolic address)
        lw      1       2       3       load reg2 with -1 (uses numeric address)
start   add     1       2       1       decrement reg1
        beq     0       1       2       goto end of program when reg1==0
```

```
        beq     0       0       start   go back to the beginning of the loop
        noop
done    halt                            end of program
five    .fill   5
neg1    .fill   -1
stAddr  .fill   start                   will contain the address of start
```

และนี้คือ machine language ที่ถูกแปลงมาจาก assembly ข้างบน:

```
(address 0): 8454151 (hex 0x810007)
(address 1): 9043971 (hex 0x8a0003)
(address 2): 655361 (hex 0xa0001)
(address 3): 16842754 (hex 0x1010002)
(address 4): 16842749 (hex 0x100fffd)
(address 5): 29360128 (hex 0x1c00000)
(address 6): 25165824 (hex 0x1800000)
(address 7): 5 (hex 0x5)
(address 8): -1 (hex 0xffffffff)
(address 9): 2 (hex 0x2)
```

เนื่องจาก program จะเริ่มที่ 0 เสมอดังนั้นควรจะ แสดงผล ที่ไม่ใช่ address นั่นคือ

```
8454151
9043971
655361
16842754
16842749
29360128
25165824
5
-1
2
```

Assembler ที่เขียนขึ้นนี้ควรจะมีส่วนที่จะหา errors ที่จะกล่าวถึงต่อไปนี้

1. การใช้ labels ที่ undefine

2. การใช้ label ที่เหมือนกัน
3. การใช้ offsetField ที่มีจำนวน bit มากกว่า 16 bits
4. การใช้ opcode นอกเหนือจากที่กำหนด

Assembler ควรจะใช้ exit(1) ถ้ามี error และ exit(0) ถ้า program สามารถทำงานได้โดยที่ไม่เจอ error แต่เนื่องจากนี้ไม่ใช่ simulator ดังนั้น assembler ไม่ควรจะ detect error ที่จะเกิดในขั้นตอน ของการ simulation เช่น infinite loop หรือ กระโดดไปที่ address –1 ฯลฯ

ในการทดสอบ assembler จงใช้ assembly program ข้างต้น และ assembly program อย่างอื่น อีกเพื่อเป็นการทดสอบ assembler และอย่าลืมทดสอบ การ check error ของ assembler ด้วย

อย่าลืมว่า offsetField ใช้ 2's complement ดังนั้นมันสามารถเก็บเลขได้ตั้งแต่ –32768 ถึง 32767 และสำหรับ symbolic addresses assembler จะทำการคำนวณ offsetField เพื่อที่จะอ้างถึง label ที่ถูกต้อง อย่าลืม check ว่า เครื่องที่ใช้ เป็น integer 32 หรือ 16 bits ถ้าเป็น 32 bits ต้องตัดเอาเฉพาะ 16 bits negative values ของ offsetField เพราะ assembler ที่สร้างใช้ 16 bits 2's complement number สำหรับ offsetField

Behavioral Simulator

ส่วนที่ 2 ของ project นี้ก็คือการเขียน program ที่สามารถ simulate machine code program input ของ ส่วนนี้ก็คือ machine code ที่ถูกสร้างจาก assembler program นี้ควรเริ่มจากการ initialize registers ทุกตัวและ set program counter เป็น 0 และมันจะ simulate จนกระทั่ง halt simulator ควรจะเรียก printState (อธิบายข้างล่าง) ก่อนที่จะทำแต่ละ instruction และก่อนที่จะ exit จาก program function นี้จะ print state ปัจจุบันของ machine (program counter, registers, memory) มันจะ print memory contents สำหรับแต่ละ memory location ที่ถูกกำหนดใน machine code file (addresses 0-9 ในตัวอย่างข้างบน) ใช้ machine code จาก assembler เป็น test case

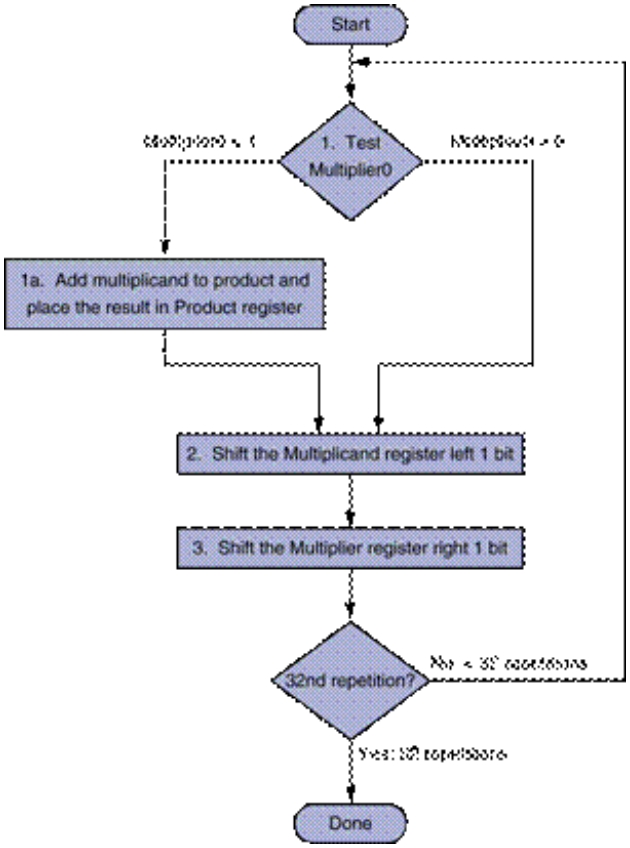
สำหรับ lw, sw และ beq อย่าลืมว่า offsetField เป็น 16 bits 2's complement number ดังนั้นต้องแปลง negative ให้เป็น 32 bits negative integer ถ้าใช้เครื่องที่เป็น 32 bit integer โดยใช้ sign extension ใช้ function ในการแปลง

```
int
convertNum(int num)
{
    /* convert a 16-bit number into a 32-bit integer */
    if (num & (1<<15) ) {
        num -= (1<<16);
    }
    return(num);
}
```

Assembly-Language สำหรับ Multiplication และ Combination

ส่วนที่ 3 ของ project นี้คือการเขียน assembly language program เพื่อทำการ**คูณเลข 2 ตัว** เลขทั้ง 2 ควรจะอยู่ที่ memory locations "mcand" และ "mplier" ผลลัพธ์ควรจะถูกเก็บที่ register 1 เราอาจจะสมมุติว่า เลขทั้ง 2 ตัวมีอย่างมาก 15 bits และเป็นเลขบวก อาจจะใช้ algorithm ดังแสดงในรูป

ก็ได้



อย่าลืมว่าการ shift left 1 bit คือการบวกตัวมันเอง จาก instruction set ที่กำหนด จะเป็นการง่ายในการแก้ไข algorithm เพื่อหลีกเลี่ยงการใช้ right shift และส่ง program ที่ทำการคูณ 32766*10383

multiplication program ไม่ควรจะมีมากกว่า 50 บรรทัด และ execute อย่างมาก 1000 instructions

และให้เขียน assembly-language program ในการคำนวณหา **combination(n,r)** แบบ recursive ซึ่ง คำนิยามของ combination(n,r) คือ

combination(n,0) = 1

combination(n,n) = 1

combination(n,r) = combination(n-1,r) + combination(n-1,r-1)

สำหรับ 0 £ r £ n

function ในภาษา C ในการทำ combination(n,r) คือ

int combination(int n, int r)

```
    {
        if(r == 0 || n == r)
           return(1);
        else
           return(combination(n-1,r) + combination(n-1, r-1));
    }
```

อย่าลืมว่า combination(n,r) เป็น recursive function ดังนั้น assembly-language program จะต้องใช้ recursive function call ด้วย นั่นคือ ใน program ต้องมี function ที่เรียกตัวเอง 2 ครั้ง (คำนวณ combination(n-1,r) และ combination(n-1, r-1)) และ นำผลลัพธ์มารวมกัน วิธีการนี้ไม่ใช่วิธีที่ดีที่สุด (เช่น n=9 จะใช้เวลานาน) แต่ใช้ recursive จะทำให้ นศ. เข้าใจเกี่ยวกับ recursive และ stack ใน program ควรจะรับค่า n และ r จาก Memory ที่ location ที่มี Label เป็น n และ r และ ผลลัพธ์ควรจะเก็บที่ register 3 เมื่อ program halt คำตอบที่ดีควรจะมีเพียง 45 บรรทัด ถ้า Program มีความยาวมากเกินไป ควรจะเช็คว่าทำอะไรผิดหรือไม่ และถ้าเขียน Program ได้ดี ควรจะมีไม่เกิน 5000 Instructions สำหรับ n = 7 และ r = 3

การส่งผ่านตัวแปรไปยัง subroutine อาจจะมีความสับสนในการเขียน assembly language ได้ ดังนั้นวิธีที่ง่ายคือ นศ. ใช้ function ที่ถูกเรียก save register ส่วนใหญ่ ทำให้ function ที่เรียกสามารถนำตัวแปรเหล่านั้นมาใช้ได้โดยที่ค่าของ register ไม่มีการเปลี่ยนแปลง การ save register จะต้อง save โดยการ push ลง stack ในตอนเริ่มต้นของ function และ pop ออกจาก stack ในตอนท้ายของ function ก่อนที่ function จะ return

ไม่จำเป็นต้อง save register ทุกตัว ยกตัวอย่างเช่น register ตัวหนึ่งจะเก็บค่าที่ return จาก subroutine และคำสั่ง jalr ถูกใช้ในการ return จาก subroutine จะใส่ค่าใน register ด้วยค่าที่ต่างไปจากตอนที่ subroutine ถูกเรียก ตัวอย่างข้างล่างเป็น program ที่ใช้ subroutine call ซึ่งรับ Input และเรียก subroutine ในการคำนวณ 4*input โดยที่ register 1 ถูกใช้ในการส่ง Input และ register 3 ถูกใช้ในการส่งค่า return กลับ ตำแหน่งปัจจุบันของ stack (ตำแน่งแรกจะว่าง) อยู่ที่ stack + register 7

```
        lw      0       5       pos1    $5 = 1
        lw      0       1       input   $1 = memory[input]
        lw      0       2       subAdr  prepare to call sub4n. $2=sub4n
        jalr    2       4               call sub4n; $4=return address; $3=answer
        lw      0       5       pos1    $5 = 1
        lw      0       1       input   $1 = memory[input]
        lw      0       2       subAdr  prepare to call sub4n. $2=sub4n
        jalr    2       4               call sub4n; $4=return address; $3=answer
        halt
sub4n   sw      7       4       stack   save return address on stack
        add     7       5       7       increment stack pointer
        sw      7       1       stack   save $1 on stack
        add     7       5       7       increment stack pointer
        add     1       1       1       compute 2*input
        add     1       1       3       compute 4*input
        lw      0       2       neg1    $2 = -1
```

```
        add     7       2       7       decrement stack pointer
        lw      7       1       stack   recover original $1
        add     7       2       7       decrement stack pointer
        lw      7       4       stack   recover original return address
        jalr    4       2               return.  $2 is not restored.
pos1    .fill   1
neg1    .fill   -1
subAdr  .fill   sub4n                   contains the address of sub4n
input   .fill   10                      input = 10
stack .fill  0              beginning of stack (value is irrelevant)
```

stack array เริ่มที่ Label stack และถูก extend ด้วย address ที่ใหญ่กว่า ดังนั้น label stack จำเป็นต้องอยู่ที่บรรทัดสุดท้ายของ program

อย่าลืมเช็คก่อนว่า program ที่เขียน สามารถทำงานได้กับ assembly language program ที่เป็นตัวอย่างทั้งหมดรวมทั้ง การคูณก่อนที่จะทำ combination(n,r)

Tips

สิ่งที่ยากใน program นี้คือการ allocate variable ใน 8 register ดังนั้นเพื่อทำให้งานง่ายขึ้น นศ.อาจจะเลือกใช้การ assign แบบนี้ก็ได้คือ

$0 value 0
$1 n input to function
$2 r input to function
$3 return value of function
$4 local variable for function
$5 stack pointer
$6 temporary value (can hold different values at different times, e.g. +1, -1, function address)
$7 return address

และในการทำให้ printState print content ของ stack ควรจะเพิ่ม ".fill 0" ไว้ที่ท้าย Program หลังจาก stack label เยอะๆ

Rules ของ project description

1) (assembler) outputting the machine-code file.
2) (assembler) Call exit(1) ถ้า มี errors ใน assembly-language program. Call exit(0) ถ้าทำเสร็จโดยไม่มี errors.
3) (simulator) อย่าแก้ printState หรือ stateStruct
4) (simulator) Call printState 1 ครั้ง ก่อน instruction executes และ 1 ครั้ง ก่อน simulator exits. อย่า call printState ที่อื่น.

5) (simulator) อย่า print "@@@" ที่ใดยกเว้นใน printState.
6) (simulator) state.numMemory ต้อง เท่ากับ จำนวนบรรทัด ใน machine-code file.
7) (simulator) Initialize ทุก registers ให้เป็น 0.
8) (multiplication) เก็บ ผลลัพธ์ ใน register 1.
9) (multiplication) labeled "mcand" และ "mplier" (lower-case) เป็นที่ที่ ตัวเลข 2 ตัว อยู่
10) (combination) ควรจะรับค่า n และ r จาก Memory ที่ location ที่มี Label เป็น n และ r และ ผลลัพธ์ควรจะเก็บที่ register 3

บางส่วนของ program

อยู่ที่ http://myweb.cmu.ac.th/sansanee.a/ComputerArchitecture/Project/CodeFragment.txt

ตัวอย่างของการ run simulator

อยู่ที่ http://myweb.cmu.ac.th/sansanee.a/ComputerArchitecture/Project/ExSimulator.txt

สิ่งที่ต้องส่ง

1. รายงานของแต่ละส่วนที่ประกอบด้วยรายละเอียดของโปรแกรม(เช่น Pseudo code) ผลการทดลอง และวิเคราะห์ผลการทดลอง
2. โปรแกรม assembly ให้ทดลอง **ทำอย่างอื่น** นอกเหนือจาก **การคูณ และ combination(n,r)** และให้นำ assembly นั้น ไปผ่าน assembler และ simulator ด้วย
3. โปรแกรมในแต่ละส่วน พร้อมทั้ง comment ซึ่งให้แนบมาใน **ภาคผนวก**
4. สัดส่วนการทำงานของสมาชิกในทีม (รวมถึงหน้าที่ของแต่ละคนด้วย)
5. ตารางเวลาทำงานของนักศึกษาแต่ละคนในกลุ่ม