CS/ECE 252: INTRODUCTION TO COMPUTER ENGINEERING

UNIVERSITY OF WISCONSIN—MADISON

Prof. Karthikeyan Sankaralingam, Pradip Vallathol

TAs: Deepika Muthukumar, Sujith Surendran, Murali Sivalingam, Lisa Ossian

Midterm Examination 4
In Class (50 minutes)
Wednesday, May 09, 2014
Weight: 17.5%

NO: BOOK(S), NOTE(S), OR CALCULATORS OF ANY SORT.

The exam has nine pages. You must turn in the pages 1-7. Use the blank sides of the exam for scratch work.

Circle your final answers. Plan your time carefully since some problems are longer than others.

Note:

● The Instruction set is provided on Page 9
● TRAP codes and Assembler directives are provided on Page 10

LAST NAME: __________________________________________________________
FIRST NAME: __________________________________________________________
ID# ___________________________________________________________________
<table>
<thead>
<tr>
<th>Problem</th>
<th>Maximum Points</th>
<th>Points Earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
Problem 1: Short answer type questions (7 Points)

a) (1 Point) Mention what problem could come up if DSR is not checked before writing into DDR.

You could overwrite the previous data even before it is displayed.

b) (2 Points) What is the difference between polling and interrupt based I/O ? Briefly explain a scenario where you would prefer interrupt based I/O over polling based I/O?

In Polling based I/O, the program polls checks continuously for the status of the data. In an interrupt based I/O, an interrupt will be generated to signal the program of the completion of work.
If the I/O device takes a lot of time to execute the command, then polling consumes a lot of cycles. In these cases, interrupt driven I/O is preferred

c) (2 Points) Briefly mention what happens during linking and loading phases of a program.

During the Linking phase, the symbols between different object files which are linked together gets resolved.
During the loading phase, the executable image is copied onto the memory.

d) (2 Points) What will be the value in R2 if you execute the following program (ie, when you reach the HALT instruction) ?

```
.ORIG x3000
AND R0, R0, #0
ADD R0, R0, #7
STI R0, DATA1
LD R2, DATA2
HALT

DATA1 .FILL x3006
DATA2 .FILL x3
Answer: 7
```
Problem 2: Assembly Errors (2 Points)

Identify 2 assembly errors in the following program:

```
.ORIG x3001
AND R5, R5, ZERO
LD R5, FOOBAR
NEXT ADD R5, R5, #1
BRz NEXT
LDR R4, R2, #0
ST R4, FOOBAR
NEXT HALT

ZERO .FILL #0
FOOBAR .STRINGZ "YAY!! LAST EXAM"
.END
```

a. double declaration of NEXT
b. AND with ZERO (address)
Problem 3: Traps and Subroutines (6 Points)

Suppose we want to write a new TRAP subroutine, TRAP x01. This subroutine takes two inputs from the caller of the subroutine through registers R2 and R1. R2 has the memory address of the first character of a string and R1 has the number of characters to be printed. The subroutine then prints R1 number of characters from the starting of the string (whose address is located in R2). Fill in the missing blanks to complete this subroutine code.

Assume that we are implementing callee save subroutine. Save only those registers that are needed.

Assume the trap vector table (also known as system control block) is as shown below:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>x3000</td>
</tr>
<tr>
<td>x0001</td>
<td>x4000</td>
</tr>
<tr>
<td>x0002</td>
<td>x5000</td>
</tr>
</tbody>
</table>

```
.ORIG x4000

ST __R0__, SAVREG_LOCATION1
ST __R1__, SAVREG_LOCATION2
ST __R2__, SAVREG_LOCATION3
ST __R7__, SAVREG_LOCATION4

LOOP LDR R0, R2, #0 ; Load the character to be printed
    OUT ; Print the extracted character
    ADD R2, R2, #1 ; Point R2 to the next character
    ADD R1, R1, #-1 ; Set the condition flags if this is the last character
    B Rp LOOP ; If this is not the last character, branch to LOOP

LD __R0__, SAVREG_LOCATION1
LD __R1__, SAVREG_LOCATION2
LD __R2__, SAVREG_LOCATION3
LD __R7__, SAVREG_LOCATION4

RET
```

SAVREG_LOCATION1 .BLKW 1
SAVREG_LOCATION2 .BLKW 1
SAVREG_LOCATION3 .BLKW 1
SAVREG_LOCATION4 .BLKW 1
Problem 4: I/O (4 Points)

For the assembly program, assume all the registers (R0 - R7) are initialized to the value of zero.

```
.ORIG x3000
LD R1, NEG
LD R2, SET
ILOOP
LDI R3, KBSR
BRzp ILOOP
LDI R4, KBDR
STI R2, KBSR
OLOOP
LDI R2, DSR
BRzp OLOOP
ADD R0, R4, #0
STI R0, DDR
ADD R5, R5, 1
ADD R6, R5, R1
BRnp OLOOP
HALT

NEG .FILL xFFFB
SET .FILL x4000
DSR .FILL xFE04 ; Address of DSR
DDR .FILL xFE06 ; Address of DDR
KBSR .FILL xFE00 ; Address of KBSR
KBDR .FILL xFE02 ; Address of KBDR
.END
```

(a) (2 Points) What does the above LC-3 program do?
- Enables Keyboard interrupt
- Displays the ASCII character that the user entered 5 times back to the user.

(b) (2 Points) How is the operation of the keyboard affected by the instruction \texttt{STI R2, KBSR}?
- No impact to the program logic. It just enables the Keyboard interrupt. If we are polling for next character, there can be a error window where the same old data is used even if new character is entered.
Problem 5: Two Stage Assembly Process (10 Points)

Consider the following assembly program.

.ORG x3000
LD R0, DATA
LEA R1, ZERO
STR R0, R1, #-3
LEA R5, STR1

LOOP  LDR R0, R5, 0  
BRz END
OUT
ADD R5, R5, #1
BR LOOP

END    HALT

ZERO    .FILL  #0
STR1    .STRINGZ "Wierd_Question"
ARRAY    .BLKW x5
DATA    .FILL x1B62

(a) (3 Points) In the first pass, the assembler creates symbol table. Fill in the symbol table created by the assembler for this program (in Problem 2(a))

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>3004</td>
</tr>
<tr>
<td>END</td>
<td>3009</td>
</tr>
<tr>
<td>ZERO</td>
<td>300A</td>
</tr>
<tr>
<td>STR1</td>
<td>300B</td>
</tr>
<tr>
<td>ARRAY</td>
<td>301A</td>
</tr>
<tr>
<td>DATA</td>
<td>301F</td>
</tr>
</tbody>
</table>
(b) (2 Points) In the second pass, the assembler creates a binary (.obj) version of the program, using the entries from the symbol table. Write the binary code generated for the first two instructions (LEA and LD) in the table below.

<table>
<thead>
<tr>
<th>Assembly code</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R0, DATA</td>
<td>0010 000 000011110</td>
</tr>
<tr>
<td>LEA R1, ZERO</td>
<td>1110 001 000001000</td>
</tr>
</tbody>
</table>

(c) (3 Points) Complete the missing comments for this program (For loop, consider only first iteration):

```
.ORIG x3000
LD R0, DATA ; Value loaded into R0 is ___X1B62_______
LEA R1, ZERO ; Loads __address of ZERO_____ into R1
STR R0,R1,#-3 ; Stores value X1B62 into address __X3007____
LEA R5, STR1 ; Loads __address of STR1_____ into R5

LOOP
  LDR R0, R5, 0 ; Value at R0 in the first iteration is X57
  BRZ END ; Branch to END if ZERO flag is set
  OUT ; Print the character at ___R0____
  ADD R5, R5, #1 ;
  BR LOOP ; Branch to LOOP

END
HALT ; HALT
```

ZERO .FILL #0
STR1 .STRINGZ "Wierd_Question"
ARRAY .BLKW x5
DATA .FILL x1B62

(d) (3 Points) What will be printed on the console if this program is run on PennSim?

WedQeto
LC-3 Instruction Set (Entered by Mark D. Hill on 03/14/2007; last update 03/15/2007)

PC*: incremented PC. setcc(): set condition codes N, Z, and P. mem[A]: memory contents at address A.
SEXT(immediate): sign-extend immediate to 16 bits. EEXT(immediate): zero-extend immediate to 16 bits.

1 5 14 12 11 10 9 8 7 6 5 4 3 2 1 0

AND DR, SR1, SR2; Addition
<table>
<thead>
<tr>
<th>0 0 0 1</th>
<th>DR</th>
<th>SR1</th>
<th>0 0 0</th>
<th>SR2</th>
</tr>
</thead>
</table>
| DR ← SR1 + SR2 also setcc()

ADD DR, SR1, imm5; Addition with Immediate
<table>
<thead>
<tr>
<th>0 0 0 1</th>
<th>DR</th>
<th>SR1</th>
<th>1</th>
<th>imm5</th>
</tr>
</thead>
</table>
| DR ← SR1 + SEXT(imm5) also setcc()

AND DR, SR1, SR2; Bit-wise AND
<table>
<thead>
<tr>
<th>0 1 0 1</th>
<th>DR</th>
<th>SR1</th>
<th>0 0 0</th>
<th>SR2</th>
</tr>
</thead>
</table>
| DR ← SR1 AND SR2 also setcc()

AND DR, SR1, imm5; Bit-wise AND with immediate
<table>
<thead>
<tr>
<th>0 1 0 1</th>
<th>DR</th>
<th>SR1</th>
<th>1</th>
<th>imm5</th>
</tr>
</thead>
</table>
| DR ← SR1 AND SEXT(imm5) also setcc()

BBx label (where x=r, p, sp, rp, rs, rsp): Branch
| 0 0 0 0 0 | m | p | PCoffset9 |
|GO ← ((n and N) OR (SR AND P)) OR (p AND F))
|---------|---|---|-----|
|if(GO is true) then PC<∈PC' + SEXT(PCoffset9)

JMP BaseR: Jump
<table>
<thead>
<tr>
<th>1 1 0 0 0</th>
<th>0 0 0</th>
<th>BaseR</th>
<th>0 0 0 0 0 0</th>
</tr>
</thead>
</table>
|PC ← BaseR

JSR label: Jump to Subroutine
<table>
<thead>
<tr>
<th>0 1 0 0 1</th>
<th>PCoffset11</th>
</tr>
</thead>
</table>
|R7 ← PC', PC ← PC' + SEXT(PCoffset11)

JSR BaseR: Jump to Subroutine in Register
<table>
<thead>
<tr>
<th>0 1 0 0 0</th>
<th>0 0 0</th>
<th>BaseR</th>
<th>0 0 0 0 0 0</th>
</tr>
</thead>
</table>
|temp ← PC', PC ← BaseR, R7 ← temp

LD DR, label: Load PC-Relative
<table>
<thead>
<tr>
<th>0 0 1 0 1</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
</table>
|DR ← mem[PC' + SEXT(PCoffset9)] also setcc()

LDI DR, label: Load Indirect
<table>
<thead>
<tr>
<th>1 0 1 0</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
</table>
|DR ← mem[PC' + SEXT(PCoffset9)] also setcc()

LDR DR, BaseR, offset6: Load Base+Offset
<table>
<thead>
<tr>
<th>0 1 1 0</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
</table>
|DR ← mem[BaseR + SEXT(offset6)] also setcc()

LEA, DR, label: Load Effective Address
<table>
<thead>
<tr>
<th>1 1 1 0</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
</table>
|DR ← PC' + SEXT(PCoffset9) also setcc()

NOT DR, SR; Bit-wise Complement
<table>
<thead>
<tr>
<th>1 0 0 1</th>
<th>DR</th>
<th>SR</th>
<th>1 1 1 1 1</th>
</tr>
</thead>
</table>
|DR ← NOT(SR) also setcc()

RET: Return from Subroutine
<table>
<thead>
<tr>
<th>1 1 0 0 0</th>
<th>0 0 0</th>
<th>1 1 1 0 0 0 0 0</th>
</tr>
</thead>
</table>
|PC ← R7

RTI: Return from Interrupt
| 1 0 0 0 0 | 0 0 0 0 0 0 0 0 0 |
|---------|-----|-----|-----|-----|
|See textbook (2nd Ed. page 537).

ST DR, label: Store PC-Relative
<table>
<thead>
<tr>
<th>0 0 1 1</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
</table>
|mem[PC' + SEXT(PCoffset9)] ← SR

STI, SR, label: Store Indirect
<table>
<thead>
<tr>
<th>1 0 1 1</th>
<th>SR</th>
<th>PCoffset9</th>
</tr>
</thead>
</table>
|mem[mem[PC' + SEXT(PCoffset9)]] ← SR

STR DR, BaseR, offset6: Store Base+Offset
<table>
<thead>
<tr>
<th>0 1 1 1</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
</table>
|mem[BaseR + SEXT(offset6)] ← SR

TRAP: System Call
<table>
<thead>
<tr>
<th>1 1 1 1</th>
<th>0 0 0 0 0</th>
<th>trap&lt;ect0</th>
</tr>
</thead>
</table>
|R7 ← PC', PC ← mem[SEXT(trap<ect0)]

: Unused Opcode
| 1 1 1 0 | |
|---------|

Initiate illegal opcode exception
| 1 1 1 0 | |
|---------|

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

10
### TRAP CODES

<table>
<thead>
<tr>
<th>Code</th>
<th>Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>TRAP x25</td>
<td>Halt execution and print message to console.</td>
</tr>
<tr>
<td>IN</td>
<td>TRAP x23</td>
<td>Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>OUT</td>
<td>TRAP x21</td>
<td>Write one character (in R0[7:0]) to console.</td>
</tr>
<tr>
<td>GETC</td>
<td>TRAP x20</td>
<td>Read one character from keyboard. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>PUTC</td>
<td>TRAP x22</td>
<td>Write null-terminated string to console. Address of string is in R0.</td>
</tr>
</tbody>
</table>

### ASSEMBLER DIRECTIVES

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>address</td>
<td>starting address of program</td>
</tr>
<tr>
<td>.END</td>
<td></td>
<td>end of program</td>
</tr>
<tr>
<td>.BLKW</td>
<td>n</td>
<td>allocate n words of storage</td>
</tr>
<tr>
<td>.FILL</td>
<td>n</td>
<td>allocate one word, initialize with value n</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>n-character string</td>
<td>allocate n+1 locations, initialize w/characters and null terminator</td>
</tr>
</tbody>
</table>