Review Practice Exam #2

CPS310 Computer Organization II

TOTAL	/ 19

Architecture and Comparison of Processors

The *Lab1 Computer* is the computer created in the Multimedia logic simulator.

1. Which microprocessor is a RISC processor?
    a. The Lab1 computer.
    b. The 6502.
    c. The ARC.
    d. None of the above.

2. Microprocessor can be classified into Harvard and Von Neumann architectures. What do these terms mean?

3. Which operation would be handled by the hardware of the ALU?
    a. Deciding whether to take a branch or not
    b. *ALU= Arithmetic Logic Unit* – this component performs mathematical operations such as add, subtract as well as logical AND, OR, XOR.
    c. Managing the stack
    d. Decoding an instruction

-Memorize how many 8-bit registers each of the processors have

4. How many working 8-bit registers does the Atmel AVR microprocessor have?
    a. The processor has two registers: A, B.
    b. The processor has one main working register called W.
    c. The processor has one main working register called the *accumulator* and two index registers X, Y.
    d. The processor has 32 general purpose registers, register r0 is not really register since its contents are always zero.
    e. The processor has 32 general purpose registers, the last six are also combined into the 16-bit X, Y, and Z registers.

5. Which is true about the following about x86 jump statements:
    A: JMP is taken if the Z-flag=0, whereas a JZ jump is always taken
    B: JMP is taken if the Z-flag=1, whereas a JZ jump is always taken
    C: JMP is is always taken, whereas a JZ jump is taken if the Z-flag=0
    D: JMP is is always taken, whereas a JZ jump is taken if the Z-flag=1

6. The AVR has:
    A: combined memory and data
    or B: separate memory and data

LAB1 Computer Manual Assembly

Mnemonic	Hex	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Data	JMP	BEQ	LDA	LDB	LDT	MX1	MX0	ALU2	ALU1	ALU0						
										PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
LDT 0x45	0845	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	1
LDA 0x30	1030	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0
JMP 0x27	8027	1	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1

Assemble the following commands. I.e turn the mnemonic code into hexadecimal form. The first three examples are done for you. Hint: convert to the binary bits I15-I0 first and then convert to hex. You can also refer to the Lab1 Computer diagram at the back of this exam.

7. The hex code for mnemonic **LDB 0xEA** is:
   A: 0x10EA
   B: 0x18EA
   C: 0x08EA
   D: 0x20EA

8. The hex code for mnemonic **JMP 0x1F** is:
   A: 0x801F
   B: 0x401F
   C: 0x081F
   D: 0x181F

PIC16C5X Microcontroller

Manually assemble the following PIC16C5X instructions into hexadecimal. Use the reference sheets at the back of the exam

9. MOVLW 0x17 =0x_____

10. BTFSS file13,4 =0x_____ (decimal 13 = 0x0D)

6502

Use the 6502 instruction reference sheet to identify the addressing mode.

11. LDX$2000,Y -this instruction is what addressing mode?

12. TAY -this instruction is what addressing mode?

13. STA ($02,X) -this instruction is what addressing mode?

6502

Use the 6502 instruction reference sheet to **assemble** the following instructions.

Note that unlike the Lab1 computer and PIC16C5X all instructions are not the same length.

- *Remember: 16 bit numbers are entered little-endian (low-order byte comes first).*
- *Note: the first byte is called the opcode.*

14. LDX #$44 =_____

15. TXA =_____

16. STA $E000 =_____

17. lda ($77),Y =_____

Code Fragments

18. The following 6502 code fragment performs which function?

```
CLC
LDA $2000
ADC $2010
STA $2020
LDA $2001
ADC $2011
STA $2021
```

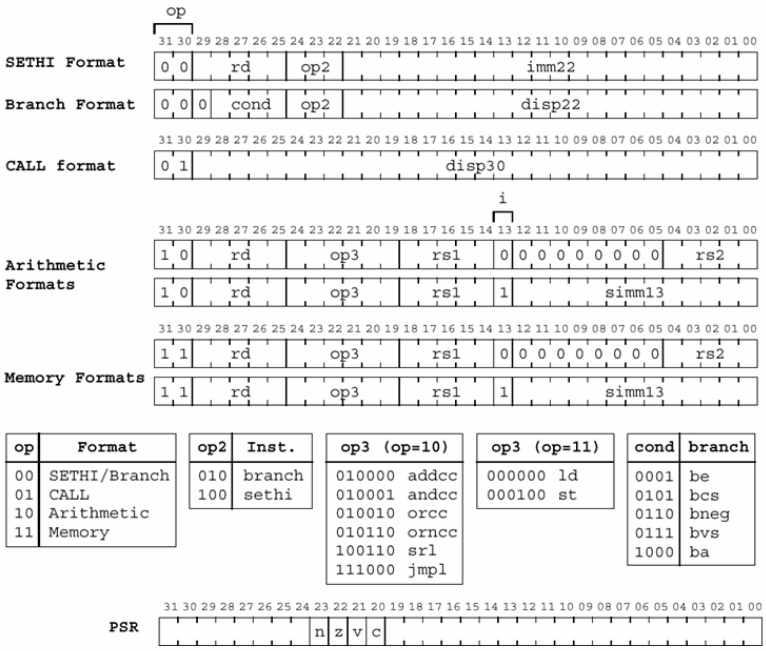
a. Adds a 16-bit number in $2000,2001 to one in $2010,2011 and puts the result in $2020,2021
b. Adds a 16-bit number in $2010,2011 to one in $2000,2001 and puts the result in $2020,2021
c. . Adds a 16-bit number in $2020,2021 to one in $2000,2001 and puts the result in $2010,2011
d. Decrements a 16-bit number.

X86

19. Which register is automatically decremented and compared with the **LOOP** instruction?
    a. AX
    b. BX
    c. CX
    d. DX

Some reference slides from the textbook **Computer Architecture and Organization: An Integrated Approach** by **Miles J. Murdocca**

ARC Instruction and PSR Formats



op	Format
00	SETHI/Branch
01	CALL
10	Arithmetic
11	Memory

op2	Inst.
010	branch
100	sethi

op3 (op=10)
010000 addcc
010001 andcc
010010 orcc
010110 orncc
100110 srl
111000 jmpl

op3 (op=11)
000000 ld
000100 st

cond	branch
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba

Table 4.10 Branch on condition codes

Opcode	cond	Operation	Flags tested
ba	1000	Branch Always	1
bn	0000	Branch Never	0
bne	1001	Branch on Not Equal	not Z
be	0001	Branch on Equal	Z
bg	1010	Branch on Greater	not (Z or (N xor V))
ble	0010	Branch on Less or Equal	Z or (N xor V)
bge	1011	Branch on Greater or Equal	not (N xor V)
bl	0011	Branch on Less	N xor V
bgu	1100	Branch on Greater Unsigned	not (C or Z)
bleu	0100	Branch on Less or Equal Unsigned	C or Z
bcc	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	not C
bcs	0101	Branch on Carry Set (Less than, Unsigned)	C
bpos	1110	Branch on Positive	not N
bneg	0110	Branch on Negative	N
bvc	1111	Branch on Overflow Clear	not V
bvs	0111	Branch on Overflow Set	V

Arc Assembly language code (simplified SPARC code)

Instruction type	Symbolic Rep.	Description
Data Transfer		
	ld [x], %r1	load %r1 with the contents of X
	ld %r0+x, %r1	load %r1 with the from the address of %r0+x
	st %r1, [x]	store the contents of %r1 in X
	st %r1, %r0+x	store the contents of %r1 in the address of %r0+x
Branch		
	be label	Branch when z condition is 1 to label
	bne label	Branch when z condition is 0 to label
	bcs label	Branch when c condition is 1 to label
	bcc label	Branch when c condition is 0 to label
	bneg label	Branch when n condition is 1 to label
	bpos label	Branch when n condition is 0 to label
	bvs label	Branch when v condition is 1 to label
	bvc label	Branch when v condition is 0 to label
	ba label	Branch to label
	jmpl %r15+4, %r2	set the PC to %r15+4 and set current PC in %r2
	call label	branch to the label, and put PC of call instruction in %r15
	Condition z,c,n,v are set by arithmetic and logic opcodes n =1, when negative z =1, when zero v =1, when overflow c =1, when carry	
Arithmetic		
	addcc %r1, %r2, %r3	%r3 = %r1 + %r2 (or number) and set condition codes
	add %r1, %r2, %r3	%r3 = %r1 + %r2 (or number)
	subcc %r1, %r2, %r3	%r3 = %r1 - %r2 (or number) and set condition codes
	sub %r1, %r2, %r3	%r3 = %r1 - %r2 (or number)
	srl %r1, %r2, %r3	Shift %r1 right by the value in %r2 (or number) and store in %r3. Vacant bits are filled with 0
	sll %r1, %r2, %r3	Shift %r1 left by the value in %r2 (or number) and store in %r3. Vacant bits are filled with 0
	sra %r1, %r2, %r3	Shift %r1 right by the value in %r2 (or number) and store in %r3. The sign bit is replicated as the value is shifted.
Logic		
	orcc %r1, %r2, %r4	%r4 = %r1 OR %r2 (or number) and set N and Z conition
	or %r1, %r2, %r3	%r3 = %r1 OR %r2 (or number)
	andcc %r1, %r2, %r3	$r3 = %r1 AND %r2 (or number) and set N and Z conition
	and %r1, %r2, %r3	$r3 = %r1 AND %r2 (or number)
Program markers		
	halt	Stops the simulator
	nop	Performs no operation, but increments the program counter
	.begin	start of assembly code
	.end	end of assembly code
	!	comment. Begins with ! and continues to end of line.
	.org 2048	Start assembly code at address 2048

X86 Instructions

```
push <reg32>
push <mem>
push <con32>

pop <reg32>
pop <mem>
```

Some examples of mov instructions using address computations are:

```
mov eax, [ebx]          ; Move the 4 bytes in memory at the address contained in EBX into EAX
mov [var], ebx          ; Move the contents of EBX into the 4 bytes at memory address var. (Note, var is a 32-bit constant).
mov eax, [esi-4]        ; Move 4 bytes at memory address ESI + (-4) into EAX
mov [esi+eax], cl       ; Move the contents of CL into the byte at address ESI+EAX
mov edx,                ; Move the 4 bytes of data at address ESI+4*EBX into EDX
[esi+4*ebx]
```

Some examples of invalid address calculations include:

```
mov eax, [ebx-ecx]          ; Can only add register values
mov [eax+esi+edi], ebx      ; At most 2 registers in address computation
```

Syntax

```
lea <reg32>,<mem>
```

Examples

`lea eax, [var]` — the address of *var* is placed in EAX.
`lea edi, [ebx+4*esi]` — the quantity EBX+4*ESI is placed in EDI.

```
call <label>
ret
```

Syntax

`je <label>` (jump when equal)
`jne <label>` (jump when not equal)
`jz <label>` (jump when last result was zero)
`jg <label>` (jump when greater than)
`jge <label>` (jump when greater than or equal to)
`jl <label>` (jump when less than)
`jle <label>` (jump when less than or equal to)

```
cmp DWORD PTR [var], 10
jeq loop

cmp eax, ebx
jle done
```

x86 Instructions

Examples copied from Evans' tutorial

ALU operations : Add, subtract, logical operations

`add eax, 10` — EAX ← EAX + 10
`add BYTE PTR [var], 10` — add 10 to the single byte stored at memory address var

`sub al, ah` — AL ← AL - AH
`sub eax, 216` — subtract 216 from the value stored in EAX

`dec eax` — subtract one from the contents of EAX.
`inc DWORD PTR [var]` — add one to the 32-bit integer stored at location *var*

`and eax, 0fH` — clear all but the last 4 bits of EAX.
`xor edx, edx` — set the contents of EDX to zero.
`not BYTE PTR [var]` — negate all bits in the byte at the memory location *var*.

`shl eax, 1` — Multiply the value of EAX by 2 (if the most significant bit is 0)
`shr ebx, cl` — Store in EBX the floor of result of dividing the value of EBX by 2^n where *n* is the value in CL.

Integer Multiplication and Division :

- Two operand version : 1st times 2nd result overwrites 1st (leftmost) operand
- three operand version : 2nd times 3rd result goes into 1st (leftmost) operand

Syntax

```
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

`imul eax, [var]` — multiply the contents of EAX by the 32-bit contents of the memory location *var*. Store the result in EAX.

`imul esi, edi, 25` — ESI → EDI * 25

Syntax

```
idiv <reg32>
idiv <mem>
```

`idiv ebx` — divide the contents of EDX:EAX by the contents of EBX. Place the quotient in EAX and the remainder in EDX.

`idiv DWORD PTR [var]` — divide the contents of EDX:EAS by the 32-bit value stored at memory location *var*. Place the quotient in EAX and the remainder in EDX.