October 24

- Programming problems?
- Read Section 6.1 for November 5
- How instructions execute
- Test Preview
- Ask Questions!

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Memory Read Completion

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

- A FSM looks at the op-code to determine how many...

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];  IR is "Instruction Register"
PC = PC + 4;
```

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

```
ALUOut = A + sign-extend(IR[15-0]);
```

- R-type:

```
ALUOut = A op B;
```

- Branch:

```
if (A==B) PC = ALUOut;
```


Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR = Memory[ALUOut];
      or
Memory[ALUOut] = B;
```

MDR is Memory Data Register

- R-type instructions finish

```
Reg[IR[15-11]] = ALUOut;
```

Step 5 Memory Read Completion

- `Reg[IR[20-16]]= MDR;`

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC]			
	PC = PC + 4			
Instruction	A = Reg [IR[25-21]]			
decode/register fetch	B = Reg [IR[20-16]]			
	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address	ALUOut = A op B	ALUOut = A + sign-extend	if (A ==B) then	PC = PC [31-28] II
computation, branch/		(IR[15-0])	PC = ALUOut	(IR[25-0]<<2)
jump completion				
Memory access or R-type	Reg [IR[15-11]] =	Load: MDR = Memory[ALUOut]		
completion	ALUOut	or		
		Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Test Preview

- You should be able to answer all of these

1a-1b

1. Consider the hex number 0x0000A120

a.[4] What bit pattern (base 2 number) does it represent?

0000 0000 0000 0000 1010 0001 0010 0000

b. [4] If you complement it (that is one's complement), what hex number do you get?

0xFF FF 5E DF

1c-1d

Consider the hex number 0x0000A120

c. [4] If you negate it, what hex number do you get?

0xFF FF 5E E0

d. [4] What decimal value does it represent when interpreted as a signed integer?

$10x16^3 + 1x16^2 + 2*16$
= 40960 + 256 + 32
= 41248

2a-2b

2. Consider the 32 bit binary number
1100 0000 0000 0000 0000 0000 0000 0000

a. [4] What is its hex representation?

0xC0000000

b. [4] Interpret it as an unsigned integer; what is its value in decimal?

2^31 + 2^30 = 3G = 3221225472

2c-2d

Consider the 32 bit binary number
1100 0000 0000 0000 0000 0000 0000 0000

c. [4] Interpret it as a signed integer; what is its value in decimal?

$-2^{31} + 2^{30} = -1073741824$

d. [4] Interpret it as an IEEE floating point single precision number (sign, 8-bit exponent with bias=127, 23-bit mantissa); what is its value in decimal?

$(-1)^{sign} * 1.0 * 2^{(128-127)} = -2.0$

$2^7=128$

1100 0000 0000 0000 0000 0000 0000 0000

3

3. Using 8-bit twos-complement binary numbers (that is, numbers only 8 bits long to save you work)

a. [8] Show how to add 23 and –17.

23=00010111, 17=00010001, ~17=11101110,
-17 = 11101111, 00010111 + 11101111 = 00000110

b. [8] Show how to subtract 15 from 20 by negating and adding.

20=00010100, 15=00001111, ~15=11110000,
-15=11110001, 00010100+11110001 = 00000101

4

4. The branch instructions bne (branch-not-equal) and beq (branch-equal) use PC-relative addressing.

a. [8] Assuming the condition is true, show how to determine the byte address of the next instruction to be executed, given the PC value and 16 bits of the immediate address field of the instruction.

M = immediate field 16 bits interpreted as a signed integer.

PC = PC+4+4*M

b. [4] What characteristics of the branches in real programs justify the choice of PC-relative addressing?

branches tend to be to nearby addresses, this is called "locality"

5

5. [8] Suppose the target address "L1" for a beq instruction is 500k bytes away. This is too far away to represent in the 16 bit immediate address of the beq instruction. What instruction sequence should we use to accomplish the goal of branching to L1 on equal compare without using any registers? (I'm interested in the idea here, not the exact instructions).

500K is slightly smaller than 2^19 (=512K).

change the beq to a bne that skips the next instruction on true. Make the next instruction a jump which has a 25 bit address field.

```
e.g.,          beq    $t0, $t1, L1
becomes
               bne    $t0, $t1, Next
               j      L1
        Next:  ...
```

6a

6. When we access an array of integers, we have to multiply the index by 4 to get the byte offset. A certain program has an average CPI of 2.5 with 10% of the cycles devoted to converting array indices to addresses with an instruction sequence like:

```
mul $t0, $t1, 4     % convert index to a byte offset
add $t0, $t0, $a0   % add offset to the base address
                    % of the array
```

Assume the above sequence takes 10 cycles.

a. [4] Show how to implement this address calculation by replacing the multiply with adds.

```
add $t0, $t1, $t1
add $t0, $t0, $t0
add $t0, $t0, $a0
```


6b

...average CPI of 2.5 with 10% of the cycles devoted to converting...

b. [4] Assuming the add sequence takes 3 cycles, what is the speedup for new program over using the multiply sequence?

10 Cycles takes 10%, so:

10/0.1 = 100 cycles total. (10 in conversion, 90 in others)

New C=93. (3 in conversion, 90 in others)

Speedup = 100/93 = 1.075

Or using Amdahl's Law:

$$\frac{1}{0.9 + 0.1*3/10}$$

6c-6d

...average CPI of 2.5 with 10% of the cycles devoted to converting...

c. [4] Show how to implement this address calculation by replacing the multiply with a shift-left-logical.

```
sll $t0, $t1, 2
add $t0, $t0, $a0
```

$t1 = $b_{31}b_{30}b_{29}b_{28}……b_1b_0$ --> $t0 = $b_{29}b_{28}……b_1b_000$

$t0 = $t1 * 4 if $b_{31}b_{30}$=00

d. [4] Assuming the shift/add sequence takes 2 cycles, what is the average CPI for the new program?

10/0.1 = 100 cycles total. CPI=2.5 so I = 40. (CPI = C / I)

New I=40, (Still 2 in conversion, 38 in others)

New C=92. (2 in conversion, 90 in others)

New CPI=92/40 = 2.3.

7

7. [8] Suppose we want to design a 1-bit subtracter like our 1-bit adder. The inputs are A, B, and Borrow-In. The outputs are A-B and Borrow-out. Make a truth table for outputs A-B and Borrow-out.

A	B	BI	A-B	BO
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

e.g.

...0 1 A 0 1...

- ...0 0 B 1 0...

8

8. [8] Consider 3 functions, F1, F2, and F3. F1 calls F2 and F2 calls F3, which is a leaf function. Function F1 uses and overwrites registers **t0-7 and s0-s3**. Function F2 uses and overwrites registers **t0-3 and s0-1**. Function F3 uses and overwrites registers **t0-t4**. At the beginning of each function we save some registers on the stack and at the end of each function we restore these registers. For each function, specify the minimal set of registers that must be saved and restored using the MIPS register saving conventions.

   MIPS convention: no need to save 't' registers.

   F1 must save s0, s1, s2, s3, RA.

   F2 must save s0, s1, RA.

   F3 doesn't have to save anything. Only –1 if you forgot RA.

   Why RA?

   F1 uses "jr $ra" at the end. But it changes $ra when it calls F2 by "jal F2".

Assignment

- Email to me (gb@cs.unc.edu) by Friday 5PM.
  - at least 2 questions on material we've covered that you would like to review
  - OR a statement that you don't have any questions