

# CMPE 110 Homework #1

John Allard  
January 19th, 2015  
jhallard@ucsc.edu

## 1. Question #1 - Power

- (a) Question #1A : ‘...However, we discussed that nowadays power density and heat has become an issue preventing scaling of frequency. Discuss why power and temperature are becoming an issue’.

**Answer :**

As we try to increase the frequency, we must scale the power and voltage of the chip. If you combine this with shrinking transistors, you can end up with leakage between the transistors. This can lead to many different types of problems, including increased power consumption and junk values being inserted into places they weren't meant to be. In addition to electrical leakage, the temperature of the chips is scaling with frequency at too great of a rate. A tiny increase in frequency can lead to a disproportionally large increase in power consumption and temperature, thus engineers have looked for other places to speed up the CPU as apposed to increasing the raw frequency (by doing things like adding more computing cores to the processor). Frequency is also limited by the time it takes the electrical signals to propagate through the circuit, this propagation time means that the clock edges are not straight vertical, but rather sloped. If we try and make the frequency too fast, these slopes will overlap and this would imply that not all of the signals have had time to propagate through all of the required circuit components. So in summation, even though the power density stays the same when we shrink the transistors, it does not stay the same when we increase the frequency and shrink the size of the transistors, which causes engineers to have to look for different ways to speed up a cpu.

- (b) Question #1B : Given the rough formulations governing power and frequency for each voltage region, discuss which region (consider Near and Super-threshold only) is more energy efficient.

**Answer :** To start, we will make the following assumptions :

Near-threshold voltage =  $V_{nth} = k$ , Super-threshold voltage =  $V_{sth} = 2k : k \in \mathbb{R}^+$

Power  $\propto V^3$ , Delay  $\propto \frac{1}{V}$ , Energy  $\propto$  Power  $\times$  Delay

Energy Efficiency  $\propto$  Energy  $\times$  Delay = Power  $\times$  Delay<sup>2</sup>

The energy efficiency can then be calculated for both the near and super threshold voltage levels, as shown below.

Table 1: Efficiency Calculations

Step	Near-Threshold	Super-Threshold
1	$P_{nth} = V_{nth}^3 = k^3$	$P_{sth} = V_{sth}^3 = (2k)^3 = 8k^3$
2	$D_{nth} = \frac{1}{V_{nth}} = \frac{1}{k}$	$D_{sth} = \frac{1}{2k}$
3	$E_{nth} = P_{nth} * D_{nth} = \frac{k^3}{k} = k^2$	$E_{sth} = P_{sth} * D_{sth} = \frac{8k^3}{2k} = 4k^2$
4	$EE_{nth} = E_{nth} * D_{nth} = \frac{k^2}{k} = k$	$EE_{sth} = E_{sth} * D_{sth} = \frac{4k^2}{2k} = 2k$

Thus voltage levels that are near the threshold are more energy efficient.

## 2. Question #2 Computing ISA's

Table 2: Question #2 Answers

Architecture	Bytes in Program	Bytes Fetched	Bytes Loaded	Bytes Stored
x86	13	89	40	40
MIPS	32	248	40	40
Stack ISA	24	195	20	20

## (a) Question #2A x86 CISC ISA

Fill out the first row of the above table (from the handout), but assume 32-bit data values.

**Answer :** (See Table #2, row #1)

**Explanation** - There are 13 bytes in the program, we know this by summing the 6 instructions by their byte count. 89 bytes fetched is determined by tracing the routine through 10 iterations plus the initializing instructions. Bytes loaded and Bytes stored all come from the `inc(ra, rb, imm)` instruction, which loads and stores 4 bytes for each of the 10 calls.

## (b) Question #2B MIPS RISC ISA

Write the assembly code that would generate if the C code were compiled on a machine that uses the MIPS ISA. Fill out the second row of the above table.

**Answer :**

`r1` is the counter variable. `r2` contains the size variable (10). `r3` contains the address of the first item in the array.

```

        subu r1, r1, r1    # initialize counter to zero
        j CMPR             # jump to comparison
LOOP:   lw r4, r3           # load current array value
        addiu r4, r4, #1    # increment current array value
        sw r4, r3          # store the incremented value
        addiu r3, r3, #4    # move to the next index in the array
        addiu r1, r1, #1    # increment the counter
CMPR:   bne r1, r2, #-6     # compare counter with size, loop again if !=

```

**Explanation** - The second row table values were calculated as follows. There are 8 instructions in the program, consisting of a constant 4 bytes each, giving 32 total bytes. There are 6 instructions inside of the loop, which iterates 10 times, giving  $6 \times 4 \times 10$  bytes fetched for the loop, plus the 2 instructions before the loop. This totals 248 bytes fetched. 40 bytes are loaded and stored as is the same with the x86 code, 4 bytes for each iteration of the loop.

## (c) Question #2C STACK ISA

Write the assembly code that would generate if the C code were compiled on a machine that uses this stack-based architecture. Assume that going into the code, that the top of stack contains size and the second entry in the stack contains `aptr`. Fill out the third row in the above table

**Answer :**

```

        goto CMPR          # Compare counter at stack top
LOOP:   swap               # put address at top of stack
        dup                # dup the address so popm doesn't destroy it
        pushm              # get array value at address on stack top
        pushi #1           # put 1 on stack
        add                # increment array value
        popm               # put inc'd value back at memory address
        pushi #2           # put 2 on stack
        add                # increment the address by 2 bytes
        swap               # put counter at the top, address at 2nd spot
        pushi #-1          # put -1 on stack
        add                # decrement the counter by 1
CMPR:   bgtz LOOP          # if counter not zero, loop again

```

**Explanation** - The third row table values were calculated as follows. There are 13 instructions in the program, summing up over their respective instruction lengths yields 24 bytes. Of these 24 bytes, 19 are inside of the loop, which iterates 10 times, which gives 190 bytes. Add in the 5 bytes to do this initial goto and you end up with 195 bytes fetched. The program performs 10 loads and 10 stores, each with 2 bytes, giving 20 bytes total load/stored.

(d) Question #2D COMPARISON

Compare the three ISAs studied with respect to static code size, number of instruction bytes fetched during execution, and memory traffic. Dont simply summarize the table but analyze what the numbers mean.

**Answer :**

For static code size, it makes sense that the **x86 RISC ISA** had a smaller code size than **MIPS**, because they are both 32-bit but the **x86 ISA** has variable length instructions, meaning it doesn't need to waste 4 bytes for instructions that need less. It also had the fewest lines of code because each line could accomplish so much. Compare this to the **STACK ISA**, which was much longer because each instruction only does a little bit of work. The static code size between **MIPS ISA** and the **STACK ISA** seems like it could go either way, **MIPS** instructions do more work, but each instruction is twice as large as the **STACK ISA** instructions, so depending on the cmplexity of the code it seems like either one could be bigger than the other.

For the bytes fetched, this parallels the static code size in a way. Because each of these functions has to do 10 iterations plus some outside work, each of the bytes fetched will be roughly proportional to ten times the static code size. The **x86 ISA** fetched less than 10 times its static code size, because over  $\frac{1}{3}$  of the bytes in the static code are outside of the loop, and thus are only executed once. The **MIPS ISA** and **STACK ISA** are a little closer to 10 times their static code size, because a larger proportion of the instructions are inside of the loop

For memory traffic, all of these different ISAs needed to do the same number of loads and stores, 10 each (one for each iteration). This means that the **MIPS ISA** and the **x86 ISA** loaded and stored the same number of bytes, because they both work on 32-bit data values. The **STACK ISA** only works on 16 bit values, so it only had half as much traffic from the data memory.

3. Question #3 DataPath

Write the control signal and wire values for the following instructions:

**Notes :** For all vertical MUXs, a 0 control input value selects the top wire, 1 selects the bottom. For all horizontal MUXs, a 0 control value selects the left input, a 1 selects the write input. For example, **ALUinB** would select the **SX** value if it recieves a 1 control value, and selects from the **E** wire if it receives a 0 control value.

Table 3: Question #3A Answer Part 1

Instr	A	B	C	D	E	F	G	H	I	J	K
Load	PC+4	PC	ld \$r2, [\$r1-20]	r1	X	-20	X	r1-20	r1-20	100	100
Store	PC+4	PC	st [\$r2], \$r1	r2	r1	0	r1	r2+0	r2+0	X	X
Add	PC+4	PC	add \$r1, \$r2, \$r3	r1	r2	0	X	X	r1+r2	X	r1+r2
Jump	4000	PC	jmp #1000	X	X	1000	X	X	X	X	X

Table 4: Question #3A Answers Part 2

Instr	Rwe	Rdst	ALUinB	ALUop	DMWe	Rwd	JP	BR
Load	1	1 (r2)	1	ADD	0	J	0	-16
Store	0	X	1	ADD	1	X	0	4
Add	1	1 (r3)	0	ADD	0	I	0	X
Jump	0	X	X	X	0	X	1	X

- (a) `ld $r2, [$r1-20]`

**Explanation :** The answers to this problem are contained in the first rows of the two tables above. This load instruction only increases the PC by a single instruction, so the PC is only increased by 4 bits. This instruction uses an immediate value, (-20), to offset the value contained in register `r1`. This adjusted value serves as a memory address, which means `r1-20` is the address value going into the data memory. Since we are not storing, the data value line going into the data memory is not important. The ALU receives the ADD operation and performs the offset of `r1` by -20. The destination register is `r2`, which means the `Rwe` signal is high. The `Rwd` MUX selects from the data memory and not directly from the ALU. The `JP` muX selects the PC value incremented by 4, which is the top wire above the `BR` wire. The `K` wire contains the value selected by the `Rwd` mux. The `BR` wire is not important in this circuit.

- (b) `st [$r2], $r1`

**Explanation :** The answers to this problem are contained in the second rows of the two tables above. This Store instruction only increases the PC by a single instruction, so the PC is only increased by 4 bits. This instruction uses an immediate value of 0 as the offset for `r2`. There are no registers being written to, so the `Rwe` signal is low. The value in `R1` is passed to the `E` wire right to the data input for the data memory, while `r2` and the 0 immediate value pass through the ALU with an ADD operation (this isn't necessary but it works just fine), which then gets fed to the address input to the data memory. The `Rwd` muX can select either one, because no registers are enabled for writing anyways, thus it is labeled X. The `JP` muX selects the PC value incremented by 4, which is the top wire above the `BR` wire. The `BR` wire is not important in this circuit.

- (c) `add $r1, $r2, $r3` **Explanation :** The answers to this problem are contained in the third rows of the two tables above. This Add instruction only increases the PC by a single instruction, so the PC is only increased by 4 bits. Both of the source registers are used this time and passed straight through to the ALU, which also receives an ADD opcode. This summed value is fed into the data memory via the `I` wire to the `Rwd` mux, which selects the `I` wire instead of the junk/unimportant value coming from the data memory. No immediate value is used in this computation, thus the `ALUinB` muX selects the `E` wire instead of the sign-extended immediate value that doesn't exist here. `JP` once again selects the `PC+4` value instead of the value that is made by offsetting the PC by immediate. The `BR` wire is not important in this circuit.

- (d) `jmp #1000`

**Explanation :** The answers to this problem are contained in the fourth rows of the two tables above. I had some trouble understanding what exactly is going on with this circuit. The PC value is not going to be simply incremented by 4, it is going to be completely overridden by the immediate value. The immediate value is 1000, indicating that the user wants to jump to the 1000th instruction. The `ALUop` is undefined/not important here, because we don't need to perform any data manipulation with register data. Likewise, the data memory isn't touched, so `I`, `G`, `H`, `J` are unimportant, as is the `Rwd` muX because no registers are written to during this operation. The `JP` muX selects the bottom wire, which is the immediate value (1000) after it has been left-shifted twice (multiplied by 4), thus it is not the value 4000. This value is fed along the `A` wire back to the PC, which is now in the correct place.

4. Propose two ways to support ARM-like branches on the datapath given in lecture and discuss their tradeoffs.

- (a) One way to do this would be to add 2 flip-flops that the decoder has access to. The reason for two flip flops is to ensure that a branch instruction is only called after a compare instruction, if a compare instruction was not called then the first flip flop will contain a zero value and we will know that there was an error in the code. The first of these flip flops is near the decoder and signals whether or not there was just a compare (`cmp`) instruction. If there was a compare instruction this flip-flop will store a 1. The second flip flop will store the result of the compare instruction, 1 if true 0 if false. This means the second flip flop will be set based on the value of the `I` wire, if it is high then the second flip flop will be high and visa versa. This second flip flop will be on the right of the data memory. The branch instruction can then use the value in the second flip flop to determine if it should branch or not. So this second flip flop would actually act as a control signal for the `JP` mux, if it is high then the mux selects the shifted immediate value. Otherwise it would select the

PC+4 value that propagates through the BR signal. No registers would have to be set during this routine, and no data memory has to be stored.

- (b) A second way to do this would be to perform the subtraction of the two input registers in the ALU. We then use this output as a control signal for the JP mux to select between going to the next instruction or jumping to two instruction ahead (thus skipping the branch). So we would need the decoder to add an artificial immediate value of 2 to the instruction when it sees a compare instruction. This immediate value will be shifted left twice, making it 8. This value of 8 will get added to the current PC by the mux before the BR signal. This value will get fed through to the JP mux, which will select between the PC+8 or the PC+4 signal based on the result of the subtraction in the ALU. This choice will be fed along the A wire back to the PC, setting it up to take the branch instruction if the subtraction yielded a 0 value and skipping ahead by two instructions (8 bytes) if the control signal yielded anything else.
- (c) The tradeoffs of the first idea are as follows. It seems to work well because we only need to add in two simple flip-flops, which is not a large expense at all. These two bits will let the circuit know both that it is appropriate for a branch command to be seen (meaning it follows a compare command), and also whether or not this branch should actually be taken. The down side of this approach is that it seems kind of ad-hoc, if we just throw in a few flip-flops for every single function we needed are CPU design would be too complicated. It seems that it would make more sense for the CPU to compress the two instructions into one in some manner, by perhaps computing the ALU inputs while waiting to grab the jump value. The second option seems to work pretty well, it will only ever look at the branch instruction if the comparison instruction yielded a difference of 0 between the values in the two registers, which would save us an instruction to look at if the comparison fails.