CMPE 110 Computer Architecture

Winter 2015, Homework #1

Adel Danandeh

addanand@ucsc.edu

Q1.A

On one hand, current and voltage are decreasing by factor of $1/\sqrt{2}$ and power = current * voltage, therefore power decreases by factor of ½. So over time there is power loss. On the other hand, power converts to heat, and too much heat can destroy a chip in seconds. (chip melts down, and we will be faced with leakage of electrons)

Based on Moore's Law size of transistors are reduced to the scale of atoms, which this means that will be physically impossible to increase the density. Power must decrease; otherwise more power causes temperature to increase. In order to keep the high performance systems cool, we need to have continual cooling in a large machine room, which results in substantial operational costs. In reality, power density cannot stay constant, since power has to decrease, in other words if numerator decreases in the formula of power density in order to make the result to be a constant number density has to increase to balance the equation, which is physically impossible to increase the density.

Q1.B

Near-threshold voltage (V/2):

Power$\propto V^3/2^3$ Delay$\propto 1/(V/2) = 2/V$ Freq. $\propto V/2$

Energy$\propto V^3/2^3 * 2/V = v^2/2^2$ Energy Efficiency $\propto = V^2/2^2 * 2/V = V/2$

Super-threshold voltage (V):

Power$\propto V^3$ Delay$\propto 1/V$ Freq. $\propto V$

Energy$\propto V^3 * 1/V = v^2$ Energy Efficiency $\propto = V^2 * 1/V = V$

We conclude that near-threshold voltage is more energy efficient than super-threshold voltage based on calculations above.

Frequency is lower if we supply a near-threshold voltage. This reduction of the frequency might decrease the performance. Therefore we should analyze the design and goals that we are aiming for to decide if it is better to be more energy efficient even if we are forced to reduce the frequency.

Q2.A

Architecture	Bytes in Program	Bytes Fetched	Bytes Loaded	Bytes Stored
X86	13	89	40	40
MIPS	28	52	40	40
Stack ISA	23	180	20	20

Q2.B

Assumption: $r0: counter, $r2: size = 10, $r3: aptr[i]

```
        subu  $r0, $r0, $r0          // initializing counter i to 0
        j L1                         // Jump to label L1
        lw  $r3, $r0($r1)            // load aptr[i]
        addiu  $r3, $r3, 1           // increment aptr[i]
        sw  $r1, $r0($r3), 1
        addiu $r0, $r0, 1            // increment i (counter)
L1:
        bne  $r0, $r2, -16          // loop until $r0 is not equal to $r2
```

Q2.C

```
        go to cmp     5+1+1+1+1+1+2+1+1+1+2+1+5 = 23 bytes in program
loop:   swap
        dup
        dup
        pushm
        pushm
        pushi #1
        add
        popm
        swap
        pushi #1
        sub
cmp:    bgtz  loop
```

Q2.D

Regarding the static code the size of x86 architecture is better because it has less bytes in program. This is because its instructions have more functionality. For instance with only one instruction, such as “inc”, we can do more than in the other two architectures, which they require more instructions.

Regarding the bytes fetched, x86 architecture is the one that fetches less bytes and stack-ISA is the one that fetches the most bytes. This is because of the number of instructions that program has for each architecture. The more instructions architecture has, more bytes are fetched.

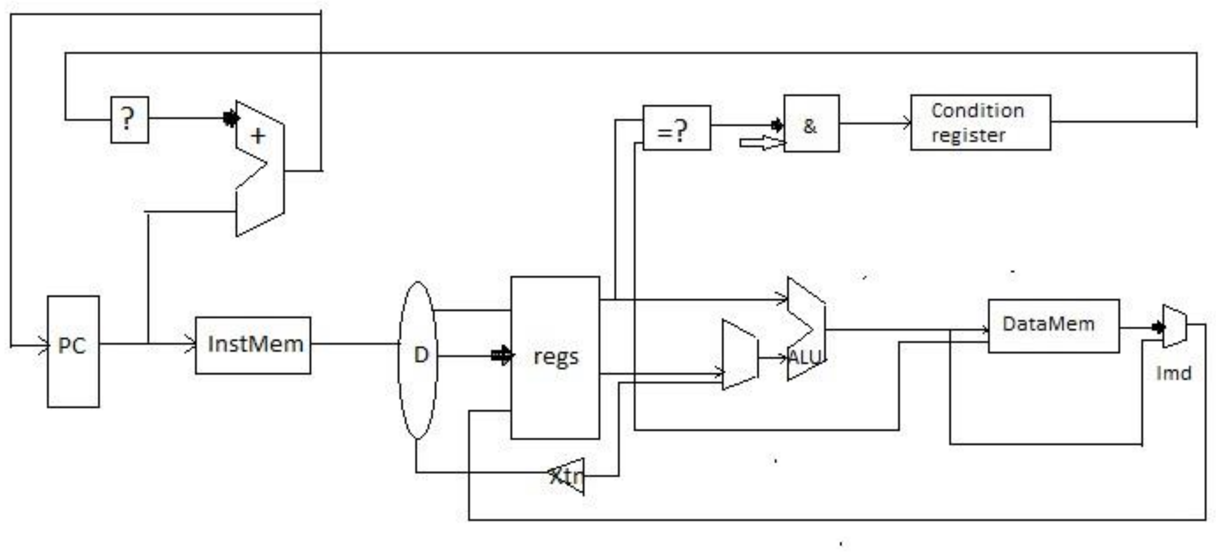
Memory traffic is the same for all of the architectures. However in x86 we load and store more bytes, because the data values are 32-bit instead of 16-bit, but the times we load and store are the same for all three of them.

Q3.

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

Instr	Rwe	Rdst	ALUinB	ALUop	DMwe	Rwd	JP	BR
Load	1	0	1	Sub(1)	0	1	0	0
Store	0	0	0	X	1	X	0	0
Add	1	1	0	Add(0)	0	0	0	0
Jump	0	0	X	X	0	X	1	X

Q4.



One way could be taking advantage of the four flags; the ALU has: Z for zero, N for negative number, V for overflow and C for carry. For the first instruction, the comparison, the ALU will perform a subtraction. In case the values of the registers are the same the result will be zero, so the zero flag will be set to one. Then, for the next instruction depending on the flags of the ALU the value that has to be added to the PC will be chosen. It is assumed that all of that is in the ? box that handles the branch.

The tradeoff for this option is that we have to implement the flags for ALU, which could make it more complicated but on the other hand, we are not adding wires or any other components so we don't make the technology bigger. Furthermore, technology consumes less energy.

The second way would be to add a special register to store the result of the comparison. For the first instruction the comparison would be made as explained in the diagram given in class that I have drawn (the datapath drawn above). After the comparison, instead of going to a multiplexor to decide which value has be added to the PC, we store the result in a register "condition register". Therefore, for the next instruction the value of the register is passed to ? box where the branch is handled. In this way we have kept the value of the comparison for the next instruction.

The tradeoff would be that we need to add a new register (we could also use one of the register file, but we would need to make sure that is only for branch, because otherwise we could overwrite values) and wires, making the technology bigger (also more energy would be consumed). Registers are very fast but expensive. An advantage would be that registers are very fast at accessing memory so even though we are adding wires and components; it may not slow down our technology that noticeable. Also, the new ALU is not modified so it doesn't become more complicated.