University of California at Berkeley
College of Engineering
Computer Science Division – EECS

CS 152
Fall 1995

D. Patterson & R. Yung

Computer Architecture and Engineering
Midterm I Solutions

Question #1: Technology and performance **[20 pts]**

a) Calculate the average execution time for each instruction with an infinitely fast memory. Which is faster and by what factor? Show your work. **[6 pts]**

$$\text{InstructionTime GaAS} = \frac{\text{CPI}}{\text{Clock rate}} = \frac{2.5}{1000\text{MHz}} = 2.5 \text{ nanoseconds}$$

$$\text{InstructionTime CMOS} = \frac{\text{CPI}}{\text{Clock rate}} = \frac{0.75}{200\text{MHz}} = 3.75 \text{ nanoseconds}$$

$$\frac{\text{InstructionTime CMOS}}{\text{InstructionTime GaAS}} = \frac{3.75}{2.5} = 1.5$$

So, the GaAS microprocessor is 1.5 times faster than a CMOS microprocessor.

Grading : 4 points for showing your work. 1 point for having correct instruction execution times. 1 point for having correct performance factor.

b) How many seconds will each CPU take to execute a 1 billion instruction program? **[3 pts]**

Execution Time of Program = (number of instructions) × (avg inst exec. time)

Execution Time of Program on GaAS = $(1 \times 10^9) \times (2.5 \times 10^-9) = 2.5$ seconds

Execution Time of Program on CMOS = $(1 \times 10^9) \times (3.75 \times 10^-9) = 3.75$ seconds

Grading : 1 point for showing your work. 2 points for having correct program execution times.

c) What is the cost of an untested GaAs die for this CPU? Repeat the calculation for a CMOS die. Show your work. **[7 pts]**

$$\text{dies/wafer} = \left\lfloor \frac{\pi \times (\text{wafer diameter}/2)^2}{\text{die area}} - \frac{\pi \times \text{ wafer diameter}}{\sqrt{2 \times \text{ die area}}} - \text{test dies per wafer} \right\rfloor$$

$$\text{die yield} = \text{wafer yield } \times \left(1 + \frac{\text{defects per unit area} \times \text{die area}}{\alpha}\right)^{-\alpha}$$

$$\text{cost of die} = \frac{\text{cost of wafer}}{\lfloor \text{dies per wafer} \times \text{die yield} \rfloor}$$

$$\text{dies/wafer GaAS} = \left\lfloor \frac{\pi \times (10/2)^2}{1} - \frac{\pi \times 10}{\sqrt{2 \times 1}} - 4 \right\rfloor = 52$$

$$\text{die yield GaAS} = 0.8 \times \left(1 + \frac{4 \times 1}{2}\right)^{-2} = 0.088$$

cost of die GaAS $= \frac{\$2000}{[52 \times 0.088]} = \$500$

dies/wafer CMOS $= \left\lfloor \frac{\pi \times (20/2)^2}{2} - \frac{\pi \times 20}{\sqrt{2 \times 2}} - 4 \right\rfloor = 121$

die yield CMOS $= 0.9 \times \left(1 + \frac{1 \times 2}{2}\right)^{-2} = 0.225$

cost of die CMOS $= \frac{\$1000}{[121 \times 0.225]} = \$37.04$

Grading : 1 point for correctly using floor functions in formulas. 1 point deducted if formulas used are good, but computed values are wrong. Rest of the points on showing your work.

d) What is the ratio of the cost of the GaAs die to the cost of the CMOS die? **[1 pt]**

$\frac{\text{Cost of GaAS die}}{\text{Cost of CMOS die}} = \frac{\$500.00}{\$37.04} = 13.5$

Grading : 1 point for using an equation which sets up the ratio correctly.

e) Based on the costs and performance ratios of the CPU calculated above, what is the ratio of cost/performance of the CMOS CPU to the GaAs CPU? **[3 pts]**

performance $= \frac{1}{\text{execution time}}$

$\frac{\frac{\text{cost CMOS}}{\text{perf. CMOS}}}{\frac{\text{cost GaAS}}{\text{perf. GaAS}}} = \frac{\text{cost CMOS} \times \text{exec time CMOS}}{\text{cost GaAS} \times \text{exec time GaAS}} = \frac{\$37.04 \times 3.75\text{ns}}{\$500.00 \times 2.5\text{ns}} = 0.111$

Grading : 2 points for using the correct formula. 1 point for correct value.

Question #2: Instruction Set Architecture and Registers Sets **[15 pts]**

Imagine a modified MIPS instruction set with a register file consisting of 64 general-purpose registers rather than the usual 32. Assume that we still want to use a uniform instruction length of four bytes and that the opcode size is constant. Also assume that you can expand and contract fields in an instruction, but that you cannot omit them.

a) How would the format of R-type (arithmetic and logical) instructions change? Label all the fields with their name and bit length. What is the consequence of this change? **[4 pts]**

opcode	rs	rt	rd	shamt	func
6 bits	6 bits	6 bits	6 bits	2 bits	6 bits

Since you have 64 registers, you must have 6 bits for the register fields. The function field stays at 6 bits since you do not want to delete any instructions. The consequence of this is that the shift amount field is reduced to 2 bits, thus allowing you to do max of 3-bit shifts per instruction. Therefore, if you want to shift more than 3 places, you must use more than one instruction.

Grading :
-1 for non-reasonable consequences, i.e. it is just shorter.
-2 if the function field was reduced.
-3 for a decent attempt.

b) How does this change the I-type instructions? What is the consequence of this change? **[3 pts]**

opcode	rs	rt	immediate
6 bits	6 bits	6 bits	14 bits

Since the register fields are 6 bits, the immediate field is reduced to just 14 bits. The consequence of this is that for branches, your range of addresses is cut down by a factor of 4.

Grading :
-1 for non-reasonable consequences, i.e. it is just shorter.
-2 for a decent attempt

c) How does this change the J-type instructions? What is the consequence of this change? **[3 pts]**

jump	target
6 bits	26 bits

Nothing needs to be changed since the "jump" instruction does not use any registers.

Grading :
-1 for non-reasonable consequences, i.e. it is just shorter.
-2 for a decent attempt

d) Imagine we are translating machine code to use the larger register set. Give an example of an instruction that used to fit into the old format, but is impossible to translate directly into a single instruction in the new format. Write a short sequence of instructions that could replace it. **[5 pts]**

One simple example is the "sll" instruction.
Old instruction that fits in 16-bit immediate field:
sll $4,$4,9
Now becomes:
sll $4,$4,3
sll $4,$4,3
sll $4,$4,3

Grading :
-1 for mistakes of the form: addi 4,0, 32768 as an example for the old format. 16-bit values are signed for arithmetic operations, therefore the range is from -32768 to +32767.
-2 for a decent attempt.
-4 for writing anything that resembles an example.

Question #3: Left Shift vs. Multiply **[20 pts]**

Design a 16-bit *left* shifter that shifts 0 to 15 bits using only 4:1 multiplexors.

a) How many levels of 4:1 multiplexors are needed? Show your work. **[4 pts]**

This question only makes sense if you read it as asking for the *minimum* number of multiplexors needed for the operation. It can clearly be done with arbitrarily more. The minimum number of levels is $\log_4 16 = 2$

Grading: For an answer of 2 levels: 2 pts. For "showing your work," either by writing the formula or answering #3 correctly : 2pts. An answer of 3 levels, if supported by reasonable logic here or in #3 : 2pts.

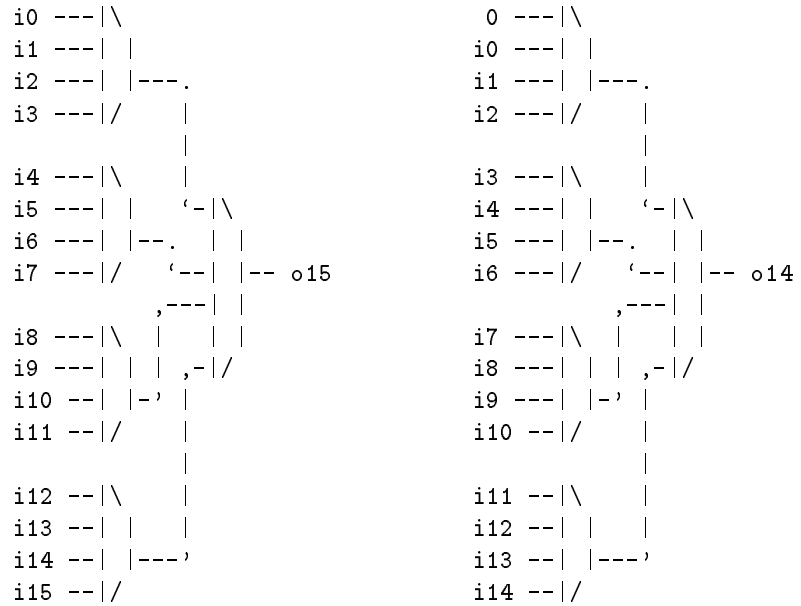
b) If the delay per multiplexor is 2ns, what is the speed of this shifter? (Assume zero delay for the wires.) **[3 pts]**

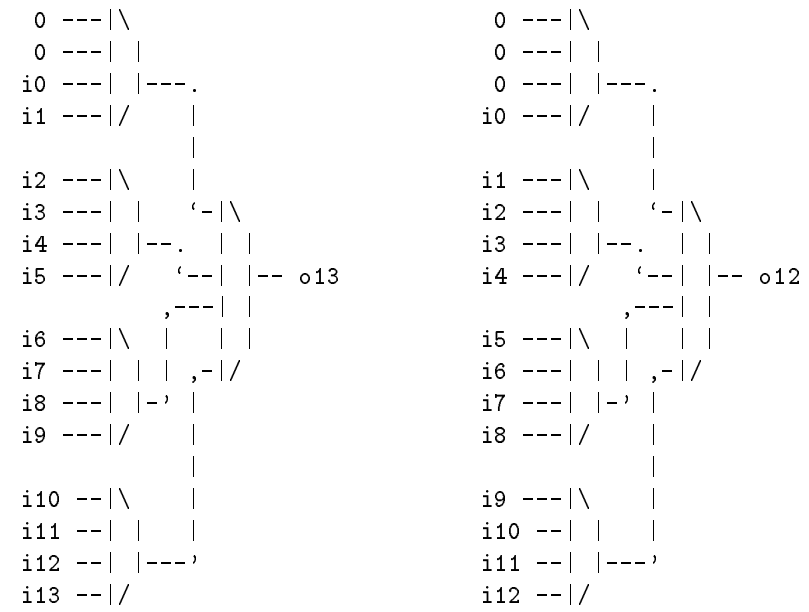
2 levels $\times$ 2ns/level = 4ns.

Grading: 1 pt if you can multiply correctly. 2 pts for the correct answer.

c) Draw the four leftmost bits of the multiplexors with the proper connections. (You might want to practice drawing it on the back of a page, and then transfer the final version here.) **[6 pts]**

The simplest solution here was to draw each output O_{15} ... O_{12} separately, each using the shift amount bit pattern as the control, ie:





The control signals to the first level muxes are $s_3 s_2$, and the control signals to the second level muxes are $s_1 s_0$, where s is the shift amount bit pattern.

Grading: We traced three arbitrary paths through the presented array. If all three worked, and all inputs/outputs were labeled: 6 pts. Leaving off the input/output labeling resulting in a loss of from 1-3 points, depending on the severity. From 1-2 points was taken off for improper or no labeling of the control inputs, again depending on their complexity and how severe the omissions were. Partial credit was awarded for partially working solutions.

d) Multiplies can be accomplished by left shifts if the multiplier is a power of two. Write the MIPS instruction(s) that performs multiply via left shifts for a multiplier that is positive and a power of two. Assume the multiplicand is in register \$4, the multiplier is in register \$5, and that the least significant 32 bits of the product should be left in \$6. You can ignore the potential for overflow in the result register. **[7 pts]**

A good solution has four instructions in the loop, a single branch instruction, and avoids a delayed branch stall and/or error:

```
       LI    $C, -1
       ANDI  $T, $5, 1
LOOP:  ADDI  $C, $C, 1
       SHR   $5, $5, 1
       BEQZ  $T, LOOP
       ANDI  $T, $5, 1    ; Delay slot, taken from top of target
EXIT:  SHL   $6, $4, $C
```

Grading: Start with full credit for nominally working code. Then, -1 pt for the delay slot being wasted or doing useless work (i.e. the SHL). -2 pts for delay slot causing improper execution. -2 pts for using `div` rather than `shr`. -2 pts for using `mult` rather than `shl` somewhere. -1 to -2 pts for failing on certain cases, such as when \$5 = 1. 2 pts were given to solutions that assumed that \$5 contained the *log* of the multiplier and implemented a single instruction. -1 pt for illegal MIPS instructions, -1 pt for wrong MIPS instructions. Just implementing the hardware multiply algorithm from class in software was worth 2 pts. No points were deducted for checking if the multiplier was zero, although zero is not a power of two.

extra credit) In class, we've seen three versions of the multiply operation (ignoring Booth encoding). Approximately how many clock cycles does a multiply take using the fastest algorithm? Approximately how much faster are multiplies that use the shifting technique (assuming the multiplier is a power of two)? **[+4 pts]**

Original multiply takes from $1 \times 32 = 32$ to $2 \times 32 = 64$ clock cycles (just answering one or the other was accepted). Most common mistake here was forgetting that it is an algorithm in *hardware*, not a MIPS program doing the same thing. The shifting technique takes a variable amount of time, based on the location of the 1 in the multiplier. Total clock cycles also depends on the number of instructions in the loop. For the code above, number of clock cycles $= 3 + 4\times$ (size of multiplier). Assuming the multiplier is 2^i and i ranges form 0 to 15, and that i is randomly distributed (not necessarily true!), the minimum, average, and maximum time to execute is $3 + 4 = 7$, $3 + 4 \times 16 = 67$, and $3 + 4 \times 32 = 131$ clock cycles. Thus, the shifting algorithm is faster when the size of the multipiler is less than or equal to 15 bits (32768) for the 64 clock original multiply, or less than or equal to 7 bits (128) in the 32 clock cycle original.

Grading: +1 pt for the original multiply cycle time. +1 pt for the shift method's cycle time. +2 pts for realizing the performance is variable based on the multiplier, and analyzing the performance in accordance. +3 pt max for an otherwise correct solution with a minor error.

Question #4: Enhancing the Single Cycle Datapath **[30 pts]**

Recall the 32-bit single-cycle control and datapath from class.

a) A MIPS instruction that would be useful to have for writing loops is Decrement and Branch if Not Zero (DBNZ).

What changes and additions would be needed to the single cycle data path to support DBNZ? Modify the datapath below to show that support. You do not need to write the control signals.

The only change to the datapath needed is to add a hardwired "1" to the ALU MUX. To execute the DBNZ instruction, the register file supplies the value of register Rd to the ALU on bus A, and the ALU MUX supplies "1" to the ALU. The ALU performs a subtraction, the output is written back into Rd, and the "Zero" output is used to select PC + 4 or PC+IMM as the next instruction to fetch. All of the other components needed to execute DBNZ are already in the datapath.

This instruction is a good example of execution concurrency. Every other instructions we've studied uses either the ALU data output *or* the ALU "Zero" output in its execution, but never both. DBNZ uses both.

Grading : Most people lost points for adding unnecessary or redundant components. Adding another ALU: -5 points. Adding unnecessary MUXes: -3 points. Unclear descriptions or annotations: -1 to -5 points.

b) The basic datapath supports only 32-bit loads. Imagine we wanted to augment the instruction set with new I-type load instructions Load from Lower to Upper Halfword (LLUH) and Load from Upper to Lower Halfword (LULH).

LLUH: Rd $\leftarrow$ $M[\text{offset} + \text{base}]_{15..0} \,||\, 0^{16}$

LULH: Rd $\leftarrow$ $0^{16} \,||\, M[\text{offset} + \text{base}]_{31..16}$

What *datapath* changes must be made to support this style of 16-bit loads? Modify the datapath reproduced below to show that support.

There are two good solutions to this problem:

1. Insert two MUXes in busW, one to select the correct value for the upper halfword, the other to select the correct value for the lower halfword.

2. Add a shifter that can shift the value on busW 16 bits left or right. *Or*, add two shifters, one to shift 16 bits left, the other to shift 16 bits right. A MUX selects whether the output of the left shifter, the right shifter, or the unshifted value gets written into the register file.

Grading : Most people either got this completely correct or made fundamental mistakes. Incorrect logic: -5 to -10 points. Overlooking the fact that all for instructions other than LLUH and LULH the data won't be shifted: -5 points. Unclear descriptions or annotations: -1 to -5 points.