CS152 Computer Architecture and Engineering

February 5, 2003

ISAs, Microprogramming and Pipelining

*Assigned February 5* Problem Set #1 *Due February 12*

**http://inst.eecs.berkeley.edu/~cs152/sp08**

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in their own solutions to the problems.

The problem sets also provide essential background material for the quizzes. The problem sets will be graded primarily on an effort basis, but if you do not work through the problem sets you are unlikely to succeed at the quizzes! We will distribute solutions to the problem sets on the day the problem sets are due to give you feedback. Homework assignments are due at the beginning of class on the due date. Homework will not be accepted once solutions are handed out.

## Problem 1: CISC, RISC, and Stack: Comparing ISAs

In this problem, your task is to compare three different ISAs. x86 is an extended accumulator, CISC architecture with variable length instructions. MIPS64 is a load-store, RISC architecture with fixed length instructions. We will also look at a simple stack-based ISA.

### Problem 1.A CISC

Let us begin by considering the following C code:

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i = 0; i<b; i++){
    result=result+a;
  }
}
```

Using gcc and objdump on a Pentium III, we see that the above loop compiles to the following x86 instruction sequence. (On entry to this code, register `%ecx` contains `i`, and register `%edx` contains `result`, and register `%eax` contains `a`. `b` is stored in memory at location `0x8049580`)

```
            xor    %edx,%edx
            xor    %ecx,%ecx
 loop:      cmp    0x8049580,%ecx
            jl     L1
            jmp    done
 L1:        add    %eax,%edx
            inc    %ecx
            jmp    loop
 done:    ...
```

The meanings and instruction lengths of the instructions used above are given in the following table. Registers are denoted with $R_{SUBSCRIPT}$, register contents with $\langle R_{SUBSCRIPT} \rangle$.

| Instruction | Operation | Length |
|---|---|---|
| add $R_{DEST}$, $R_{SRC}$ | $R_{SRC} \leftarrow \langle R_{SRC} \rangle + \langle R_{DST} \rangle$ | 2 bytes |
| cmp imm32, $R_{SRC2}$ | Temp $\leftarrow \langle R_{SRC2} \rangle$ - MEM[imm32] | 6 bytes |
| inc $R_{DEST}$ | $R_{DEST} \leftarrow \langle R_{DEST} \rangle + 1$ | 1 byte |
| jmp label | jump to the address specified by `label` | 2 bytes |
| jl label | if (SF≠OF)<br>jump to the address specified by `label` | 2 bytes |
| xor $R_{DEST}$, $R_{SRC}$ | $R_{DEST} \leftarrow R_{DEST} \otimes R_{SRC}$ | 2 bytes |

Notice that the jump instruction `jl` (jump if less than) depends on SF and OF, which are status flags. Status flags, also known as condition codes, are analogous to the condition register used in the MIPS architecture. Status flags are set by the instruction preceding the jump, based on the result of the computation. Some instructions, like the `cmp` instruction, perform a computation and

set status flags, but do not return any result. The meanings of the status flags are given in the following table:

| Name | Purpose | Condition Reported |
|---|---|---|
| **OF** | Overflow | Result exceeds positive or negative limit of number range |
| **SF** | Sign | Result is negative (less than zero) |

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

## Problem 1.B RISC

Translate each of the x86 instructions in the following table into one or more MIPS64 instructions. Place the L1 and loop labels where appropriate. You should use the minimum number of instructions needed to translate each x86 instruction. Assume that upon entry, R1 contains `b`, R2 contains `a`, R3 contains `i`. R4 should receive `result`. If needed, use R5 as a condition register, and R6, R7, etc., for temporaries. You should not need to use any floating point registers or instructions in your code. A description of the MIPS64 instruction set architecture can be found in Appendix B of Hennessy & Patterson. The authoritative source would be: http://www.mips.com/products/resource-library/product-materials/mips-architecture/

| x86 instruction | label | MIPS64 instruction sequence |
|---|---|---|
| `xor %edx,%edx` | | |
| `xor %ecx,%ecx` | | |
| `cmp 0x8049580,%ecx` | | |
| `jl L1` | | |
| `jmp done` | | |
| `add %eax,%edx` | | |
| `inc %ecx`<br>`jmp loop` | | |
| `...` | *done:* | `...` |

How many bytes is the MIPS64 program using your direct translation? How many bytes of MIPS64 instructions need to be fetched for b = 10 using your direct translation? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

**Problem 1.C Stack**

In a stack architecture, all operations occur on top of the stack. Only push and pop access memory, and all other instructions remove their operands from the stack and replace them with the result. The hardware implementation we will assume for this problem set uses stack registers for the top two entries; accesses that involve other stack positions (e.g., pushing or popping something when the stack has more than two entries) use an extra memory reference. The table below gives a subset of a simple stack-style instruction set. Assume each opcode is a single byte. Labels, constants, and addresses require two bytes.

| Example instruction | Meaning |
|---|---|
| PUSH A | push M[A] onto stack |
| POP A | pop stack and place popped value in M[A] |
| ADD | pop two values from the stack; ADD them; push result onto stack |
| SUB | pop two values from the stack; SUBtract top value from the 2nd; push result onto stack |
| ZERO | zeroes out the value at top of stack |
| INC | pop value from top of stack; increments value by one push new value back on the stack |
| BEQZ *label* | pop value from stack; if it's zero, continue at *label*; else, continue with next instruction |
| BNEZ *label* | pop value from stack; if it's not zero, continue at *label*; else, continue with next instruction |
| GOTO *label* | continue execution at location *label* |

Translate the `multiplyByB` loop to the stack ISA. For uniformity, please use the same control flow as in parts a and b. Assume that when we reach the loop, `a` is the only thing on the stack. Assume `b` is still at address 0x8000 (to fit within a 2 byte address specifier).

How many bytes is your program? Using your stack translations from part (c), how many bytes of stack instructions need to be fetched for b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored? If you could push and pop to/from a four-entry register file rather than memory (the Java virtual machine does this), what would be the resulting number of bytes fetched and stored?

## Problem 1.D Conclusions

In just a few sentences, compare the three ISAs you have studied with respect to code size, number of instructions fetched, and data memory traffic.

## Problem 1.E Optimization

To get more practice with MIPS64, optimize the code from part B so that it can be expressed in fewer instructions. There are solutions more efficient than simply translating each individual x86 instruction as you did in part B. Your solution should contain commented assembly code, a paragraph which explains your optimizations, and a short analysis of the savings you obtained.

## Problem 2: Microprogramming and Bus-Based Architectures

In this problem, we explore microprogramming by writing microcode for the bus-based implementation of the MIPS machine described in Handout #1 (Bus-Based MIPS Implementation). Read the instruction fetch microcode in Table H1-3 which was reproduced at the end of this problem (Worksheet M1-1) for readers' convenience. Make sure that you understand how different types of data and control transfers are achieved by setting the appropriate control signals before attempting this problem.

In order to further simplify this problem, *ignore* the busy signal, and assume that the memory is as fast as the register file.

The final solution should be elegant and efficient (e.g. number of new states needed, amount of new hardware added).

### Problem 2.A — Implementing Memory-to-Memory Add

For this problem, you are to implement a new memory-memory add operation. The new instruction has the following format:

**ADDm $r_d$, $r_s$, $r_t$**

ADDm performs the following operation:

$$M[r_d] \leftarrow M[r_s] + M[r_t]$$

Fill in Worksheet M1-1 with the microcode for ADDm. Use *don't cares* (*) for fields where it is safe to use don't cares. Study the hardware description well, and make sure all your microinstructions are legal.

Please comment your code clearly. If the pseudo-code for a line does not fit in the space provided, or if you have additional comments, you may write in the margins as long as you do it neatly. Your code should exhibit "clean" behavior and not modify any registers (except $r_d$) in the course of executing the instruction.

Finally, make sure that the instruction fetches the next instruction (i.e., by doing a microbranch to FETCH0 as discussed above).

## Problem 2.B — Implementing DBNEZ Instruction

DBNEZ stands for Decrease Branch Not Equal Zero. This instruction uses the same encoding as conditional branch instructions on MIPS:

| 6 | 5 | 5 | 16 |
|---|---|---|---|
| **opcode** | **rs** | | **Offset** |

DBNEZ decrements register **rs** by 1, writes the result back to **rs**, and branches to (**PC**+4)+**offset**, if result in **rs** is not equal to 0. Offset is sign extended to allow for backward branches. This instruction can be used for efficiently implementing loops.

Your task is to fill out Worksheet M1-2 for DBNEZ instruction. You should try to optimize your implementation for the minimal number of cycles necessary and for which signals can be set to don't-cares. You do not have to worry about the busy signal.

(Note that the microcode for the fetch stage has changed slightly from the one in the Problem M1.4.A, to allow for more efficient implementation of some instructions.)

## Problem 2.C — Instruction Execution Times

How many cycles does it take to execute the following instructions in the microcoded MIPS machine? Use the states and control points from MIPS-Controller-2 in Lecture 4 and assume Memory will not assert its busy signal.

| Instruction | Cycles |
|---|---|
| `SUB  R3,R2,R1` | |
| `SUBI R2,R1,#4` | |
| `SW   R1,0(R2)` | |
| `BEQZ R1,label  # (R1 == 0)` | |
| `BNEZ R1,label  # (R1 != 0)` | |
| `J    label` | |
| `JR   R1` | |
| `JAL  label` | |
| `JALR R1` | |

Which instruction takes the most cycles to execute? Which instruction takes the fewest cycles to execute?

**Problem 2.D** **Exponentiation**

Ben Bitdiddle needs to compute the power function for small numbers. Realizing there is no multiply instruction in the microcoded MIPS machine, he uses the following code to calculate the result when an unsigned number `m` is raised to the `n`th power, where `n` is another unsigned number.

```
if (m == 0) {
    result = 0;
}
else {
    result = 1;
    i = 0;

    while (i < n) {
        temp = result;
        j = 1;
        while (j < m) {
            result += temp;
            j++;
        }
        i++;
    }
}
```

The variables `i`, `j`, `m`, `n`, `temp`, and `result` are unsigned 32-bit values.

Write the MIPS assembly that implements Ben's code. Use only the MIPS instructions that can be executed on the microcoded MIPS machine (ALU, ALUi, LW, SW, J, JAL, JR, JALR, BEQZ, and BNEZ). The microcoded MIPS machine does not have branch delay slots. Use `R1` for `m`, `R2` for `n`, and `R3` for `result`. At the end of your code, only `R3` must have the correct value. The values of all other registers do not have to be preserved.

How many MIPS instructions are executed to calculate the power function? How many cycles does it take to calculate the power function? Again, use the states and control points from MIPS-Controller-2 and assume Memory will not assert its busy signal.

| m, n | Instructions | Cycles |
|---|---|---|
| 0, 1 | | |
| 1, 0 | | |
| 2, 2 | | |
| 3, 4 | | |
| M, N | | |

| State | PseudoCode | ld IR | Reg Sel | Reg W | en Reg | ld A | ld B | ALUOp | en ALU | ld MA | Mem W | en Mem | Ex Sel | en Imm | μB r | Next State |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FETCH0: | MA <- PC;<br>A <- PC | 0 | PC | 0 | 1 | 1 | * | * | 0 | 1 | * | 0 | * | 0 | N | * |
| | IR <- Mem | 1 | * | * | 0 | 0 | * | * | 0 | 0 | 0 | 1 | * | 0 | N | * |
| | PC <- A+4 | 0 | PC | 1 | 1 | 0 | * | INC_A_4 | 1 | * | * | 0 | * | 0 | D | * |
| . . . | | | | | | | | | | | | | | | | |
| NOP0: | microbranch<br>back to FETCH0 | 0 | * | * | 0 | * | * | * | 0 | * | * | 0 | * | 0 | J | FETCH0 |
| ADDM0: | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |

Worksheet M1-2

| State | PseudoCode | ld IR | Reg Sel | Reg W | en Reg | ld A | ld B | ALUOp | en ALU | Ld MA | Mem W | en Mem | Ex Sel | en Imm | μB r | Next State |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FETCH0: | MA <- PC;<br>A <- PC | * | PC | 0 | 1 | 1 | * | * | 0 | 1 | * | 0 | * | 0 | N | * |
| | IR <- Mem | 1 | * | * | 0 | 0 | * | * | 0 | * | 0 | 1 | * | 0 | N | * |
| | PC <- A+4;<br>B <- A+4 | 0 | PC | 1 | 1 | * | 1 | INC_A_4 | 1 | * | * | 0 | * | 0 | D | * |
| . . . | | | | | | | | | | | | | | | | |
| NOP0: | microbranch<br>back to FETCH0 | * | * | * | 0 | * | * | * | 0 | * | * | 0 | * | 0 | J | FETCH0 |
| DBNEZ: | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |

Worksheet M1-2

## Problem 3: A 5-Stage Pipeline with an Additional Adder

In this problem we consider a new datapath to improve the performance of the fully-bypassed 5-stage 32-bit MIPS processor datapath given in Lecture 4. In the new datapath the ALU the Execute stage is replaced by a simple adder and the original ALU is moved from the Execute stage to the Memory stage (See Figure 3-A). The adder in the 3rd stage (formerly Execute) is used only for address calculations involving load/store instructions. For all other instructions, the data is simply forwarded to the 4th stage.

The ALU will now run in parallel with the data memory in the 4th stage of the pipeline (formerly Mem). During a load/store instruction, the ALU is inactive, while the data memory is inactive during the ALU instructions. *In this problem we will ignore jump and branch instructions.*

**Problem 3.A** **Elimination of a hazard**

Give an example sequence of MIPS instructions (five or fewer instructions) that would cause a pipeline bubble in the original datapath, but not in the new datapath.

**Problem 3.B** **New hazard**

Give an example sequence of MIPS instructions (five or fewer instructions) that would cause a pipeline bubble in the new datapath, but not in the original datapath.

**Problem 3.C** **Comparison**

Compare the advantages and disadvantages of the new datapath. Which one would you recommend? Justify your choice.

| IF | ID | AC | EX/MEM | WB |
|---|---|---|---|---|
| Instruction fetch | Instruction decode and register read | Address calculation | ALU execution and memory access | Writeback to register file |

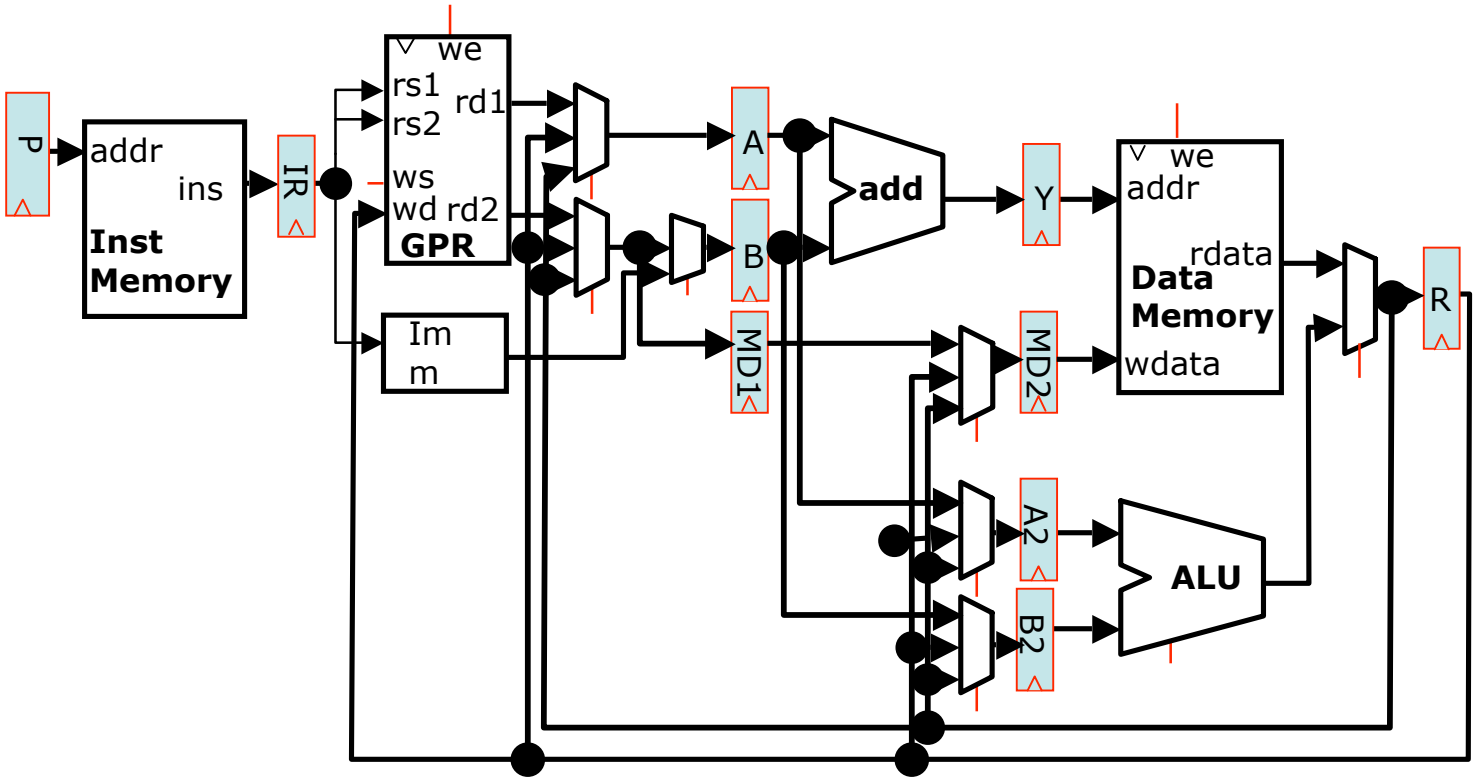


Figure 3-A. 5-Stage Pipeline with an Additional Adder

**Problem 3.D** **Datapath Improvement**

Consider a MIPS ISA that only supports register indirect addressing, i.e. has no displacement (base+offset) addressing mode. Assuming the new machine only had to support this ISA, how could the datapath be improved? Draw the new datapath showing your design. (You do not have to show everything -- just the important features like pipeline registers, major components, major connections, etc.) Compare the hazards in this new datapath with the hazards in datapaths shown in Figure 3-A and the original datapath in 4. Justify the new datapath.

**Problem 3.E** **Displacement Addressing Synthesizing**

If the MIPS ISA did not have displacement addressing, what would programmers do? Could you still write the same programs as before? Explain.

**Problem 3.F** **Jumps and Branches**

Now we will consider jumps and branches for the pipeline shown in part A of this problem. Assume that the branch target calculation is performed in the Instruction Decode stage. In what pipeline stages can you put the logic to determine whether a conditional branch is taken? (don't worry about duplicating logic) What are the advantages and disadvantages between the different choices? For each choice, consider the number of cycles for the branch delay, any additional stall conditions, and any potential changes in the clock period.