

CMU 18-447 INTRODUCTION TO COMPUTER ARCHITECTURE, SPRING 2012
HANDOUT 2 / HW 1: INSTRUCTION SET ARCHITECTURE (ISA)

Prof. Onur Mutlu, Instructor
Chris Fallin, Lavanya Subramanian, Abeer Agrawal, TAs

Given: Wednesday, Jan 18, 2012
Due: **Monday, Jan 30, 2012**

1 SPIM Simulator (5pt)

The SPIM simulator is described in the textbook and is available as spim447 or xspim447 on the ECE Linux workstations (/afs/ece/class/ece447/bin). Work through the Getting Started with xspim tutorial on the CD that comes with the Patterson&Hennessy textbook. (The tutorial description is also available at <http://pages.cs.wisc.edu/~larus/xspim.pdf>.) Once you are familiar with SPIM/XSPIM, you can use it to help test and debug your MIPS assembly programs below. You can also use it to potentially test and debug your lab 1 assignment.

2 ISA (25 pt)

Your task is to compare the memory efficiency of five different styles of instruction sets for the code sequence below. The architecture styles are:

1. A zero-address machine is a stack-based machine where all operations are done using values stored on the operand stack. For this problem, you may assume that its ISA allows the following operations:
 - PUSH M - pushes the value stored at memory location M onto the operand stack.
 - POP M - pops the operand stack and stores the value into memory location M.
 - OP - Pops two values off the operand stack, performs the binary operation OP on the two values, and pushes the result back onto the operand stack.

Note: To compute $A - B$ with a stack machine, the following sequence of operations are necessary: PUSH A, PUSH B, SUB. After execution of SUB, A and B would no longer be on the stack, but the value $A-B$ would be at the top of the stack.

2. A one-address machine uses an accumulator in order to perform computations. For this problem, you may assume that its ISA allows the following operations:
 - LOAD M - Loads the value stored at memory location M into the accumulator.
 - STORE M - Stores the value in the accumulator into memory location M.
 - OP M - Performs the binary operation OP on the value stored at memory location M and the value present in the accumulator. The result is stored into the accumulator ($ACCUM = ACCUM \text{ OP } M$).
3. A two-address machine takes two sources, performs an operation on these sources and stores the result back into one of the sources. For this problem, you may assume that its ISA allows the following operation:
 - OP M1, M2 - Performs a binary operation OP on the values stored at memory locations M1 and M2 and stores the result back into memory location M1 ($M1 = M1 \text{ OP } M2$).

4. A three-address machine, in general takes two sources, performs an operation and stores the result back into a destination different from either of the sources.
Consider

- (a) A three-address memory-memory machine whose sources and destination are memory locations. For this problem, you may assume that its ISA allows the following operation:
- OP M3, M1, M2 - Performs a binary operation OP on the values stored at memory locations M1 and M2 and stores the result back into memory location M3 ($M3 = M1 \text{ OP } M2$).
- (b) A three-address load-store machine whose sources and destination are registers. Values are loaded into registers using memory operations (The MIPS is an example of a three-address load-store machine). For this problem, you may assume that its ISA allows the following operations:
- OP R3, R1, R2 - Performs a binary operation OP on the values stored at registers R1 and R2 and stores the result back into register R3 ($R3 = R1 \text{ OP } R2$).
 - LD R1, M - Loads the value at memory location M into register R1.
 - ST R2, M - Stores the value in register R2 into memory location M.

To measure memory efficiency, make the following assumptions about all five instruction sets:

- The opcode is always 1 byte (8 bits).
- All register operands are 1 byte (8 bits).
- All memory addresses are 2 bytes (16 bits).
- All data operands are 4 bytes (32 bits).
- All instructions are an integral number of bytes in length.

There are no other optimizations to reduce memory traffic, and the variables A, B, C, and D are initially in memory.

- (a) Write the code sequences for the following high-level language fragment for each of the five architecture styles.

```
A = B + C;
B = A + C;
D = A - B;
```

- (b) Calculate the instruction bytes fetched and the memory-data bytes transferred (read or written) for each of the five architecture styles.
- (c) Which architecture is most efficient as measured by code size?
- (d) Which architecture is most efficient as measured by total memory bandwidth required (code+data)?

3 MIPS Assembly (50 pt)

3.1 MIPS assembly function foo() (25 pt)

Write the MIPS assembly function foo().

```
int foo(int *A, int n) {
    int s;
    if (n>=2) {
        s=foo(A, n-1);
        s=s+A[n-2];
    }
    else {
        s=1;
    }
    A[n]=s+1;
    return A[n];
}
```

Following MIPS linkage convention, the arguments A , and n are passed in registers $r4$ and $r5$; the result should be returned in $r2$; $r31$ stores the return address; $r29$ is the stack pointer register; $r8$ – $r15$ are caller-saved temporary registers; and $r16$ – $r23$ are callee-saved. A summary of the MIPS ISA is given at the end of this handout. The MIPS architecture reference manual is present at `/afs/ece/class/ece447/docs/r4000man.pdf`.

3.2 MIPS assembly for REP MOVSB (25 pt)

Here you will implement the MIPS equivalent for a single Intel x86 instruction, REP MOVSB, which we will specify here.¹

The REP MOVSB instruction uses three fixed x86 registers: ECX (count), ESI (source), and EDI (destination). The “repeat” (REP) prefix on the instruction indicates that it will repeat ECX times. Each iteration, it moves one byte from memory at address ESI to memory at address EDI, and then increments both pointers by one. Thus, the instruction copies ECX bytes from address ESI to address EDI.

- Write the corresponding assembly code in MIPS ISA that accomplishes the same function as this instruction. You can use any general purpose register. Indicate which MIPS registers you have chosen to correspond to the x86 registers used by REP MOVSB. Minimize code size.
- What is the size of the MIPS assembly code you wrote in (a), in bytes? How does it compare to REP MOVSB in x86 (Note: REP MOVSB occupies 2 bytes.)
- Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0x00004444
EBP: 0xaaaaffff
ECX: 0x01000000
EDX: 0x0baddeed
ESI: 0x000decaf
EDI: 0x0deadfad
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, consider the MIPS assembly code you wrote in (a). How many total instructions will be executed by your code to accomplish the same function as the single REP MOVSB in x86 accomplishes for the given register state?

- Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0x00004444
EBP: 0xaaaaffff
ECX: 0x00000000
EDX: 0x0baddeed
ESI: 0x000decaf
EDI: 0x0deadfad
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, answer the same question in (c) for the above register values.

¹The REP MOVSB instruction is actually more complex than what we describe. For those who are interested, the Intel architecture manual at `/afs/ece/class/ece447/docs/intel-manual.pdf` describes the MOVSB instruction on page 1327 and the REP prefix on page 1682. We are assuming a 32-bit protected mode environment with flat addressing (no segmentation), and a direction flag set to zero. You do not need to worry about these details to complete the homework problem.

4 Performance (15 pt)

Patterson&Hennessy 1.5.4, 1.5.5, 1.5.6

Patterson&Hennessy 1.12.4, 1.12.5

5 Subroutine Call (15 pt)

Consider the following possibilities for saving the return address of a subroutine:

1. In a processor register.
2. In a memory location associated with the subroutine; i.e., a different memory location is used for each different subroutine.
3. On a stack.

Which of these possibilities supports subroutine nesting, and which supports subroutine recursion (that is, a subroutine that calls itself)?

MIPS Instruction Summary

| Opcodes | Example Assembly | Semantics |
|----------------------------|---------------------|---|
| add | add \$1, \$2, \$3 | $\$1 = \$2 + \$3$ |
| sub | sub \$1, \$2, \$3 | $\$1 = \$2 - \$3$ |
| add immediate | addi \$1, \$2, 100 | $\$1 = \$2 + 100$ |
| add unsigned | addu \$1, \$2, \$3 | $\$1 = \$2 + \$3$ |
| subtract unsigned | subu \$1, \$2, \$3 | $\$1 = \$2 - \$3$ |
| add imm. Unsigned | addiu \$1, \$2, 100 | $\$1 = \$2 + 100$ |
| multiply | mult \$2, \$3 | hi, lo = $\$2 * \3 |
| multiply unsigned | multu \$2, \$3 | hi, lo = $\$2 * \3 |
| divide | div \$2, \$3 | lo = $\$2/\3 , hi = $\$2 \bmod \3 |
| divide unsigned | divu \$2, \$3 | lo = $\$2/\3 , hi = $\$2 \bmod \3 |
| move from hi | mfhi \$1 | $\$1 = \text{hi}$ |
| move from low | mflo \$1 | $\$1 = \text{lo}$ |
| and | and \$1, \$2, \$3 | $\$1 = \$2 \& \$3$ |
| or | or \$1, \$2, \$3 | $\$1 = \$2 \$3$ |
| and immediate | andi \$1, \$2, 100 | $\$1 = \$2 \& 100$ |
| or immediate | ori \$1, \$2, 100 | $\$1 = \$2 100$ |
| shift left logical | sll \$1, \$2, 10 | $\$1 = \$2 \ll 10$ |
| shift right logical | srl \$1, \$2, 10 | $\$1 = \$2 \gg 10$ |
| load word | lw \$1, 100(\$2) | $\$1 = \text{memory}[\$2 + 100]$ |
| store word | sw \$1, 100(\$2) | $\text{memory}[\$2 + 100] = \1 |
| load upper immediate | lui \$1, 100 | $\$1 = 100 \ll 16$ |
| branch on equal | beq \$1, \$2, Label | if ($\$1 == \2) goto Label |
| branch on not equal | bne \$1, \$2, Label | if ($\$1 != \2) goto Label |
| set on less than | slt \$1, \$2, \$3 | if ($\$2 < \3) $\$1 = 1$ else $\$1 = 0$ |
| set on less than immediate | slti \$1, \$2, 100 | if ($\$2 < 100$) $\$1 = 1$ else $\$1 = 0$ |
| set on less than unsigned | sltu \$1, \$2, \$3 | if ($\$2 < \3) $\$1 = 1$ else $\$1 = 0$ |
| set on less than immediate | sltui \$1, \$2, 100 | if ($\$2 < 100$) $\$1 = 1$ else $\$1 = 0$ |
| jump | j Label | goto Label |
| jump register | jr \$31 | goto \$31 |
| jump and link | jal Label | $\$31 = \text{PC} + 4$; goto Label |