Adrian Stoll

# EECS 370 Midterm Review

**TODO: add examples for everything**

## Instructions

ISA - instruction set architecture - instructions implemented in hardware that software can use
    ex: x86, MIPS, ARM
processor microarchitecture - hardware implementation of ISA

**R-type**
opcode | src operand | src operand | dest
ex: add

**I-Type**
opcode | src operand | dest | constant
ex: beq

**J-type**
opcode | addr
ex: jump

-more registers => larger instructions
-pseudo instructions - instructions that assembler expresses in terms of another instruction that hardware implements
ex:
**`not $src, $dest`** => **`nor $src, $src, $dest`**
**`mov $src, $dest`** => **`addi $src, $dest, 0`**

CISC - complex instruction set - variable length instructions
    -can implement more instructions at cost of complexity
    -ex: x86
RISC - reduced instruction set - fixed length instructions

**immediate value** - small constant embedded in instruction
    -pros: do not have to fetch extra values from memory or use up additional registers
    -cons: value fixed throughout execution, size limited
    ex: addi, branch offset
    ++ptr => increment by small fixed amount => use addi
    if(a < b) min = a
    else min = b
    ///jump over 2 instructions if a >= b => small offset => immediate value

Problem: what if constant too large to be immediate value?
Solution:

load upper immediate
**`lui $s0, upperbits`** => $s0 = upper bits || 0 … 0
**`addi $s0, $s0, lowerbits`** => $s0 = upper bits || 0 … 0 + lowerbits = upperbits || lowerbits

j <addr> - jump to address
PC - relative addressing
    -most branches are to nearby instructions => offsets are small
beq r1, r2, <offset>
    if r1 == r2 then go to pc + 1 + offset

**Addressing Modes:**
-Direct/immediate - operand in instruction (good for code like ++ptr)
    -address is constant embedded in instruction
    -good for accessing global + static vars
    load r1, M[1500] //address is constant
    jump 3000
-Indirect
    load r1, M[M[3000]]
-register - address stored in register
    load r1, M[r2]
-base + displacement (good for indexing into an array)
    load r2, M[r1 + offset]
-pc-relative - specify offset from next instruction - good for branches
    address = PC + 4 + offset
-pseudo-direct - jump addr - upper bits from PC, lower bits from immediate value

byte-addressable - every byte gets its own address ex: MIPS
word-addressable - smallest unit of addressable memory is word

**Special registers:**
    -stack pointer - points to top of stack
    -global pointer - points to start of data segment
    -status register - holds results of comparisons, overflow bit, misc data

**Comparisons**
slti $dest, $src, constant
    -set dest = 1 if scr < constant
    -constant is immediate
slt $dest, $s1, $s2
    -dest = 1 if $s1 < $s2
unsigned versions are sltu, sltui
then use beq, or bne $dest, 0, offset

**Jump Table**
-for switch statements
-value is index into table of addresses
-n cases => if else if …. else is O(n) comparisons, vs O(1) for switch with jump table

**Endianess**
Given 32-bit value: b3 * 256^3 + b2 * 256^2 + b1*256^1 + b0 * 256^0 made of bytes b3, .. b0 there are multiple possible ways to store the bytes.
Big endian - highest order byte is stored first ie
  -order of bytes is: b3, b2, b1, b0
  -used by TCP/IP
Little endian - lower order bytes come first
  -order of bytes is: b0, b1, b2, b3
  -used by x86

**Two's complement - how to store signed numbers**
The bits: $x = x_{31}x_{30} \ldots x_1x_0$ represent $X = -x_{31}*2^{31} + x_{30}2^{30} + \ldots + x_02^0$
Properties:
-First bit is sign bit ie: $x_{31} = 1$ => negative, $x_{31} = 0$ => positive
-positive numbers are represented like normal (except they cannot use the first bit)
-range of values $-2^{31}$ to $2^{31} - 1$ (note: one more negative value than positive)
-to negate a number take the bitwise complement of the number and add 1
$x + \overline{x} = -x_{31}2^{31} + x_{30}2^{30} + \ldots + x_0 - \overline{x}_{31}2^{31} + \overline{x}2^{30} + \ldots + \overline{x}_0 = \{1\}^{32}$
So $x + \overline{x} + 1 = \{1\}^{32} + 1 = \{0\}^{32\}} = 0$
=> $-x = \overline{x} + 1$

add, addi, … for signed addition
addu, addu … for unsigned addition

**Overflow** - when result of operation cannot fit into a word
if positive + negative => overflow impossible
positive + positive = negative => overflow
negative + negative = positive => underflow
-can occur with multiplication TODO: how to detect for multiplication

**Sign Extension**
Q: How to copy 2's complement number into larger block of memory while preserving its sign?
A: fill in extra leading bits with the sign bit. (called sign extension)
TODO: add proof of correctness

**Load Instructions**
lw - load word

lh - loads half word
lb - load byte

    ex: load character form char*

lh, lb are signed loads ie: they sign extend the value they load ensure that the same 2's complement number is represented when the value is stored in a full word
lhu, lbu - unsigned loads

sh, sb - store halfword, store byte (signed and unsigned versions are the same)

**Data Layout**
-Processors access memory in blocks whose sizes are powers of 2. If multibyte value is split across two blocks, then multiple block have to be fetched to read that value => slower loads and more complex load implementation
-solution: make sure primitive data-types are fully contained in a single block by enforcing address alignment
**Alignement**
-each primitive data-type has address that is a multiple of its size
ex: int's and pointers have addresses that are multiples of 4 (for 32-bit systems), doubles have address that is multiple of 8
-padding is added to previous variable to ensure that next is aligned

**Structures** - composite data structure
ex:

```
struct Node {
      char letter;
      int count;
      Node* left;
      Node* right;
};
```

| Variable | Start | End |
|---|---|---|
| letter | 0 | 1 |
| count | 4 | 7 |
| left | 8 | 11 |
| right | 12 | 15 |

Want to be able to make arrays to structures while ensuring all values within structure are aligned.
Could do:

```
struct S{
      char c1;
      int i
      char c2;
};
```

0x00 s1.c1 s1 at 0x00 with size 9
0x01 3 bytes padding
0x04 s1.i
0x08 s1.c2
0x09 s2.c1 s2 at 0x9 with size 8
0x0A 2 bytes of padding
0x0C s2.i
0x10 s2.c2
0x11 s3.c1 s3 at 0x11 with size 8
0x12 2 bytes padding
0x14 s3.i
0x15 s3.c2
Problem: this messes up array indexing because elements are not all the same size.

Solution: align structures based on the largest primitive element they contain and pad structure to make size a multiple of the size of the largest primitive element.
-order elements in struct by size to minimize padding

**Function Calls**
MIPS
$a0 - $a4 - registers to pass function arguments in (if there are not enough registers, then put extra arguments on the stack)
$v0, $v0 - registers to store return value
$ra - register that stores return address

jal <target> - stores PC + 4 (the return address) in special register and jump to address <target>
jr <reg> - jump to address in register
    ex: at end of function, jr $ra to jump to return address

**Stack - stores local variables and function call information**
-stores return addresses
-if there are not enough registers for all local variables, use stack memory
-if there are not enough registers for all function arguments, use stack for rest
- "" return value, use the stack
-$sp - stack pointer points to top of stack

**Memory Layout**
stack (grows downward) high addresses

heap (grows upward)
data
text  low addresses

-decrement sp to add values to stack
-increment sp to pop values off of stack

**Stack Frame**

function parameters
return address
spilled registers //saved register values
local vars

-frame pointer points to bottom of frame - easier to refer to variables based on offset from frame pointer than offset from frame pointer

**Caller vs Callee-Save**

-functions share the same set of registers

Problem: a function might modify registers that its caller is still using
Solution: same values of registers on stack

-caller save - the calling function saves register to stack before executing a function call, after function call, caller restores previous register values
-callee save - each function saves values of registers to stack before it uses, at end of function, function restores previous values
Q: which registers are caller saved?
A: Specified by ISA
MIPS
\$t0 - \$t9 - temporary registers, caller-save
\$s0 - \$s7 - saved registers - callee-save

-functions often are in different files and are compiled at different times, one function might be called by many other functions => inter (between) function optimizations are too complicated for current compilers => only intra (within) function optimizations are made

**Which is better?**

Leaf function - a function the calls no other functions

-caller save is better because no saves and restores are needed

Liveness - a variable is live across a function call if its values is read after the call
ex:

```
f() {
    a = 1
    c = 9
    b = g(a, c)
    c = b + a    //a is live because its value is used after the function call
                 //c is dead because its previous value is not used after the function call
}
```

Caller save is better for c because it is dead across function call.

```
double integrate(double a, double b, double dx, double (*f)(double)) {
	double sum = 0;
	for(double x = a; x < b; x += dx) {
		sum += f(x) * dx;
	}
	return sum;
}
```

For loop variables calle-save is better. (ex: x is saved and restored once for callee-save vs once per iteration for caller-save)

**Object Files**

-output of the assembler

header - specifies sizes of following parts

text - machine code for instructions

data -contains values for initialized global variables and statics

-two parts: initialized and uninitialized

symbol table - lists globally accessible symbols

-symbol, type

-for globals, functions, externs

relocation table - locations of instructions that depend on variables/functions in other object files

-stores locations of instructions that use absolute addresses

-in table: jump, lw <global/static>, sw <global/static>

-not in table: PC-relative instructions (beq), add, addi, …, lw <stack or heap variable>

debug info

**Linker - mergers multiple object files and resolves dependencies**

Object file 1, … Object file n

=>

text 1

text 2

…

text n

data 1

data 2

…

data n

=> resolves references, checks for undefined labels, updates instructions in relocation tables with finalized addresses of symbols they refer to

**Floating Point**

IEEE 754 Floating point
-idea: use scientific notation
Format:
-|sign bit | 8 bit exponent | 23 bit mantissa|
-base 2 => base is fixed => no need to store it
-all base 2 numbers (except zero) begin with zero => implicit 1 before the mantissa
    -use exponent of -127 to represent 0
-has +/- Infinity and not a number (NaN) values
-floating point addition compares exponents => want exponent comparison to be fast
    -want to use fast unsigned comparison instead of slower 2's complement comparison
    -solution: represent exponent in biased base 127
        -represent x as x + 127
        => exponent range: -127 to 128
-Stored value represents: $(-1)^{sign} * (1 + 0.\text{mantissa bits}) * 2^{exponent - 127}$

Multiplication:
-product sign is xor of sign bits
-add exponents
-multiply mantissa (remembering implicit leading 1)
-adjust exponent if needed and normalize mantissa

Addition:
-shift mantissa of number with smaller exponent right while increasing exponent until exponents are the same (so bits with same order are aligned)
-add resulting mantissa bits
-if addition overflows, renormalize mantissa and update exponent
(More complicated that this)

-value overflows when resulting exponent is too large

**Finite State Machines**
-set of states
-determines next state based on input and current state
Moore Machine - output determined by only current state
Mealy Machine - output determined by input and current state

-can implement with two tables (details are depend on type of machine)
Table 1: |current state | input 1 | … | input n| address| - address in index in table 2
Table 2: |next state | output 1 | ... | output m|