CIS 314 : Computer Organization
Midterm Review
What is Computer Organization?
Where is the HW/SW Interface?

*Coordination of many levels (layers) of abstraction
Levels of Representation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g., Verilog Language)

Architecture Implementation

Logic Circuit Description (Verilog Language)

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

wire [31:0] dataBus;
regFile registers (databus);
ALU ALUBlock (inA, inB, databus);

wire w0;
XOR (w0, a, b);
AND (s, w0, a);
Anatomy:
5 components of any Computer

- Processor
  - Control ("brain")
  - Datapath ("brawn")
- Memory
  (where programs, data live when running)
- Devices
  - Input
  - Output

Keyboard, Mouse
Disk (where programs, data live when not running)
Display, Printer
Computer Technology - Dramatic Change!

- **Memory**
  - DRAM capacity: 2x / 2 years (since ‘96); 64x size improvement in last decade.

- **Processor**
  - Speed 2x / 1.5 years (since ‘85); 100X performance in last decade.

- **Disk**
  - Capacity: 2x / 1 year (since ‘97); 250X size in last decade.
Assembly Language

- Basic job of a CPU: execute lots of instructions.

- Instructions are the primitive operations that the CPU may execute.

- Von Neumann architecture model of program execution:
  - Stored program model: instructions and data are stored in memory
  - Fetch/Increment/Execute cycle
Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA).

- Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...
Instruction Set Architectures

- General Purpose Register ISA
- Three types based on where operands for arithmetic operations can come from.
  - Memory-memory
  - Register-memory
  - Register-register (Load/Store)

MIPS has a Load/Store ISA
RISC philosophy – Reduced Instruction Set Computing (Cocke IBM, Patterson, Hennessy, 1980s)

- Keep the instruction set small and simple, makes it easier to build fast hardware.
- Let software do complicated operations by composing simpler ones.

- Fixed instruction lengths
- Load/store instruction sets
- Limited addressing modes (ways to access variables in memory)
- Limited operations
Assembly Variables: Registers

- **Assembly Operands are registers**
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these!
  - Fast!
  - 32 registers in MIPS (Smaller is faster)
  - Each MIPS register is 32 bits wide (word)
Assembly Variables: Registers

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- For now:
  - $16 - $23 → $s0 - $s7  Saved
  - $8 - $15 → $t0 - $t7  Temporary
  - $0 → $zero  Always zero
- Registers have no type
Assembly Instructions for memory access

- **LW** $s3, X     # Load Word
  
  Loads one word of data from memory location X into register $s3

- **SW** $s4, Y     # Store Word
  
  Stores one word of data from register $s4 into memory location Y
MIPS Addition and Subtraction

- Syntax of Instructions:
  \[
  \text{OP Dest, Src1, Src2}
  \]

  where:

  \[
  \begin{align*}
  \text{OP} & \quad \text{operation name} \\
  \text{Dest} & \quad \text{operand getting result ("destination")} \\
  \text{Src1} & \quad \text{1st operand for operation ("source1")} \\
  \text{Src2} & \quad \text{2nd operand for operation ("source2")}
  \end{align*}
  \]

- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity
Addition and Subtraction of Integers

- **Addition in Assembly**
  - Example: `add $s0,$s1,$s2` (in MIPS)
    
    Equivalent to: `s0=s1+s2` (in C)

- **Subtraction in Assembly**
  - Example: `sub $s3,$s4,$s5` (in MIPS)
    
    Equivalent to: `s3=s4-s5` (in C)

- `add (add), add immediate (addi) and subtract (sub)` *cause overflow to be detected*

- `add unsigned (addu), add immediate unsigned (addiu) and subtract unsigned (subu)` do *not* cause overflow detection
Immediates

- **Immediates are numerical constants.**

- **Add Immediate:**
  
  \[
  \text{addi } $s0,\$s1,10 \text{ (in MIPS)}
  \]

- **There is no Subtract Immediate in MIPS: Why?**
Data Transfer: Memory to Register

- To transfer a word of data, we need to specify two things:
  - Register
    - # ($0 - $31) or name ($s0, $t0, etc.)
  - Memory address

- Example: `lw $t0, 8($t0)`
  - Specifies the memory address pointed to by the value in $t0, plus 8 bytes

- `lw` (Load Word - 32 bits)

- Load byte: `lb`

- Store byte: `sb`

- `lb`: sign extends to fill upper 24 bits

- `lbu`: loads byte without sign extension
Data Transfer: Reg to Memory

- `sw (store word)` is the inverse of `lw`
- **Example:** `sw $t0,12($s0)`
  
  This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into that memory address.

- **Remember:** "Store INTO memory"
- **Pitfall:** Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
**Pointers vs. Values**

- **Key Concept:** A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), and so on.

- If you write `add $t2,$t1,$t0` then `$t0` and `$t1` better contain values.

- If you write `lw $t2,0($t0)` then `$t0` better contain a pointer.

- Don’t mix these up!
More Notes about Memory: Alignment

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes.

- **Alignment**: objects must fall on address that is multiple of word size.

![Diagram showing alignment](image)

- Last hex digit of address is:
  - 0, 4, 8, or \(C_{\text{hex}}\)
  - 1, 5, 9, or \(D_{\text{hex}}\)
  - 2, 6, A, or \(E_{\text{hex}}\)
  - 3, 7, B, or \(F_{\text{hex}}\)
MIPS Decision Instructions

- Decision instruction in MIPS:
  
  `.beq   register1, register2, L1`

  `.beq` is

  “Branch if (contents of registers are) equal”

- Complementary MIPS decision instruction
  
  `.bne   register1, register2, L1`

- Pseudo instructions `blt, ble, bgt, bge`. 
In addition to conditional branches, MIPS has an **unconditional branch**:

```assembly
j label
```

Technically, it’s the same as:

```assembly
beq $0,$0,label
```
Compiling C if into MIPS

• Compile by hand

```c
if (i == j) f = g + h;
else f = g - h;
```

• Final compiled MIPS code:

```
beq $s3,$s4,True   # branch i==j
sub $s0,$s1,$s2     # f=g-h(false)
j   Fin            # goto Fin
True: add $s0,$s1,$s2 # f=g+h (true)
Fin:
```
Two Logic Instructions

○ Shift Left: \texttt{sll $s1,$s2,2} \#s1=s2<<2
  
  \begin{itemize}
  \item Before:
  \begin{verbatim}
  0000 0000 0000 0000 0000 0000 0000 0010
  \end{verbatim}
  \texttt{two}
  \end{itemize}

  \begin{itemize}
  \item After:
  \begin{verbatim}
  0000 0000 \textcolor{green}{0008} \text{\texttt{hex}}
  0000 0000 0000 0000 0000 0000 0000 1000
  \end{verbatim}
  \texttt{two}
  \end{itemize}

○ Shift Right: \texttt{srl} is opposite shift; \texttt{>>}
Loops in C/Assembly

do {  
g = g + A[i];
i = i + j;
} while (i != h);

Loop:  
g = g + A[i];
i = i + j;
if (i != h)
goto Loop;

Loop:
sll $t1,$s3,2  #$t1= 4*I
add $t1,$t1,$s5 #$t1=addr A
lw  $t1,0($t1)  #$t1=A[i]
add $s1,$s1,$t1 #g=g+A[i]
add $s3,$s3,$s4 #i=i+j
bne $s3,$s2,Loop#  goto Loop
#  if  i!=h
Inequalities in MIPS

- **Pseudo MIPS inequality instructions:**
  - `blt, bgt, ble, bge`

- **Native MIPS inequality instruction:**
  1. “Set on Less Than”
  2. **Syntax:** `slt reg1, reg2, reg3`

- **Example:**
  ```
  if (g < h) goto Less;
  slt $t0,$s0,$s1 # $t0 = 1 if g<h
  bne $t0,$0,Less
  ```
Instructions as Numbers

- Instructions are stored just like data.
- One word is 32 bits, so divide instruction word into "fields".
- Each field tells computer something about instruction.
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - I-format (load, store, branches, etc.)
  - J-format (j, jal)
  - R-format (all other instructions)
**R-Format Instructions**

- Define “fields” of the following number of bits each: \( 6 + 5 + 5 + 5 + 5 + 6 = 32 \)

<table>
<thead>
<tr>
<th>(32)</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

CIS 314 Introduction (28) Fall 2005
R-Format Example

- **MIPS Instruction:**
  
  ```
  add $8, $9, $10
  ```

  **Decimal number per field representation:**
  
<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>10</th>
<th>8</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
</table>

  **Binary number per field representation:**
  
<table>
<thead>
<tr>
<th>0000000</th>
<th>01001</th>
<th>01010</th>
<th>01000</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
</table>

  **Hex representation:**
  
  $012A \text{ } 4020_{\text{hex}}$

  - Called a **Machine Language Instruction**
I-Format Instructions

- Define “fields” of the following number of bits each: 6 + 5 + 5 + 16 = 32 bits

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

- Again, each field has a name:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
</tbody>
</table>
**I-Format Example (2/2)**

- **MIPS Instruction:**
  
  ```
  addi $21, $22, -50
  ```

<table>
<thead>
<tr>
<th>Decimal/field representation:</th>
<th>8</th>
<th>22</th>
<th>21</th>
<th>-50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary/field representation:</td>
<td>001000</td>
<td>10110</td>
<td>10101</td>
<td>11111111110011110</td>
</tr>
<tr>
<td>Hexadecimal representation:</td>
<td>22D5</td>
<td>FFCE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: *hex refers to hexadecimal.
Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.

- **Register conventions:**
  - Return address $ra$
  - Arguments $a0, a1, a2, a3$
  - Return value $v0, v1$
  - Local variables $s0, s1, \ldots, s7$
Instruction Support for Functions

- **jump and link**
  
  \textit{jal label}
  
  - Step 1 (link): Save address of next instruction into $ra$ (Why next instruction? Why not current one?)
  
  - Step 2 (jump): Jump to the given label

- **jump register**
  
  \textit{jr register}

- **In function calls:**
  
  - \textit{jal} stores return address in register ($ra$)
  
  - \textit{jr $ra} jumps back to that address
Using the Stack

Hand-compile

```c
int sumSquare(int x, int y) {
    return mult(x,x) + y;
}
```

```
sumSquare:  

```
push```

```asm
addi $sp,$sp,-8    # space on stack
lw $a1, 0($sp)    # save y
jal mult          # call mult

```

```
pop```

```asm
addi $sp,$sp,8     # restore stack
jr $ra             # return
```

mult: ...
Register Conventions

- **Caller:** the calling function
- **Callee:** the function being called

**Register Conventions:** A set of generally accepted rules as to which registers will be unchanged after a procedure call (jal) and which may be changed.
Register Conventions

- $0$: No Change. Always 0.

- $s0$-$s7$: Save and restore if you change.

- $sp$: Save and restore if you change. The stack pointer must point to the same place before and after the jal call, or else the caller won’t be able to restore values from the stack.

- $ra$: Can Change.

- $v0$-$v1$: Can Change.
Structure of a Procedure

Alfa:

1. Save Saved Registers
2. Retrieve Parameters
3. ... (omitted for brevity)
4. Save Unsaved Registers
5. Send Parameters
6. Call Procedure
7. Receive Results
8. Restore Unsaved Regs.
9. Return Results
10. Restore Saved Registers
11. Return

Beta:

1. Save Saved Registers
2. Retrieve Parameters
3. ... (omitted for brevity)
4. Save Unsaved Registers
5. Send Parameters
6. Call Procedure
7. Receive Results
8. Restore Unsaved Regs.
9. Return Results
10. Restore Saved Registers
11. Return
# MIPS Registers

| Use names for registers -- code is clearer! |
|-------------------------------------------|---|
| The constant 0                            | $0 | $zero |
| Reserved for Assembler                    | $1 | $at  |
| Return Values                             | $2-$3 | $v0-$v1 |
| Arguments                                 | $4-$7 | $a0-$a3 |
| Temporary                                 | $8-$15 | $t0-$t7 |
| Saved                                     | $16-$23 | $s0-$s7 |
| More Temporary                            | $24-$25 | $t8-$t9 |
| Used by Kernel                            | $26-27 | $k0-$k1 |
| Global Pointer                            | $28 | $gp  |
| Stack Pointer                             | $29 | $sp  |
| Frame Pointer                             | $30 | $fp  |
| Return Address                            | $31 | $ra  |
Other Registers

- $at$: may be used by the assembler at any time; unsafe to use

- $k0$–$k1$: may be used by the OS at any time; unsafe to use

- $gp$, $fp$: don’t worry about them

Note: Feel free to read up on $gp$ and $fp$ in Appendix A, but you can write perfectly good MIPS code without them.
Logical Operators

- **and, or**: Both of these expect the third argument to be a register

- **andi, ori**: Both of these expect the third argument to be an immediate

- **MIPS Logical Operators** are all *bitwise*

- **MIPS shift instructions**:
  1. **sll** (shift left logical): shifts left and **fills emptied bits with 0s**
  2. **srl** (shift right logical): shifts right and **fills emptied bits with 0s**
  3. **sra** (shift right arithmetic): shifts right and **fills emptied bits by sign extending**
Steps to Starting a Program

C program: foo.c

Assembly program: foo.s

Object (mach lang module): foo.o

Executable (mach lang pgm): a.out

Compiler

Assembler

Linker

Loader

Memory
Assembler Directives (p. A-51 to A-53)

- Give directions to assembler, but do not produce machine instructions

  - `.text`: Subsequent items put in user text segment
  - `.data`: Subsequent items put in user data segment
  - `.globl sym`: declares `sym` global and can be referenced from other files
  - `.asciiz str`: Store the string `str` in memory and null-terminate it
# Pseudoinstruction Replacement

**Pseudo:**

- `subu $sp,$sp,32`
- `sd $a0, 32($sp)`
- `mul $t7,$t6,$t5`
- `addu $t0,$t6,1`
- `ble $t0,100,loop`
- `la $a0, str`

**Real:**

- `addiu $sp,$sp,-32`
- `sw $a0, 32($sp)`
- `sw $a1, 36($sp)`
- `mul $t6,$t5`
- `mflo $t7`
- `addiu $t0,$t6,1`
- `slti $at,$t0,101`
- `bne $at,$0,loop`
- `lui $at,left(str)`
- `ori $a0,$at,right(str)`
Producing Machine Language

- **Simple Case**
  - Arithmetic, Logical, Shifts, and so on.
  - All necessary info is within the instruction already.

- **What about Branches?**
  - PC-Relative
    - pseudo-instructions -> real ones
    - we know by how many instructions to branch.
What about jumps (j and jal)?

- Jumps require **absolute address**.

What about references to data?

- `la` gets broken up into `lui` and `ori`
- These will require the full 32-bit address of the data.

These can’t be determined yet, so we create two tables...
Symbol Table & Relocation Table

- Symbol Table:
  List of “labels” in this file.

- Relocation Table
  List of “instructions” for which this file needs the address.

- .j or jal

- .la
Object File Format

- **object file header**: size and position of the other pieces of the object file
- **text segment**: the machine code
- **data segment**: binary representation of the data in the source file
- **relocation information**: identifies lines of code that need to be “handled”
- **symbol table**: list of this file’s labels and data that can be referenced
- **debugging information**
Four Types of Addresses

- **PC-Relative Addressing (beq, bne):** never relocate
- **Absolute Address (j, jal):** always relocate
- **External Reference (usually jal):** always relocate
- **Data Reference (often lui and ori):** always relocate
I-Format Problems

Solution (continued):

So how does \texttt{lui} help us?

Example:

\begin{verbatim}
addi $t0,$t0, 0xABABCDCD
becomes:
\end{verbatim}

\begin{verbatim}
lui $at, 0xABAB
ori $at, $at, 0xCDCD
add $t0,$t0,$at
\end{verbatim}
Branching

- **MIPS Code:**

  ```
  Loop: beq $9, $0, End
         addi $8, $8, $10
         addi $9, $9, -1
         j Loop
  End:
  ```

  **decimal representation:**

<table>
<thead>
<tr>
<th>4</th>
<th>9</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
</table>

  **binary representation:**

  | 000100 | 01001 | 00000 | 0000000000000000011 |
J-Format Problems

- Large addresses
  - New PC = \{ PC[31..28], target address, 00 \}

- Understand where each part came from!

- Note: \{ , , \} means concatenation
  \{ 4 bits , 26 bits , 2 bits \} = 32 bit address
  - \{ 1010, 11111111111111111111111111, 00 \} =
    10101111111111111111111111111100
2s Complement

- 2s complement of an unsigned number is obtained by inverting all the bits and adding 1.

Example:

\[ \text{2s complement of } (01111110)_2 = (10000001)_1s \quad \text{(invert)} \]
\[ = (10000100)_2s \quad \text{(add 1)} \]

- Given a number \( x \) which can be expressed as an \( n \)-bit binary number, its negative number can be obtained in 2s-complement representation using:

\[ -x = 2^n - x \]
2’s Complement Number “line”: \( N = 5 \)

- \( 2^{N-1} \) non-negatives
- \( 2^{N-1} \) negatives
- one zero
- how many positives?

10000 ... 11110 11111

00000 00001 ... 01111
Two’s Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:
  \[ d_{31} \times -(2^{31}) + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0 \]
2s complement arithmetic

Addition:

2. Perform binary addition on the two numbers.
3. Ignore the carry out of the MSB.
4. Check for overflow: Overflow occurs if the carrier into and out of the MSB are different.

Subtraction:

2. Take 2s complement of B
3. Add the 2s complement of B to A