New-School Machine Structures
(It’s a bit more complicated!)

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Big Idea #1: Levels of Representation/Interpretation

<table>
<thead>
<tr>
<th>High-Level Language Program (e.g., C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
</tr>
<tr>
<td>Assembly Language Program (e.g., MIPS)</td>
</tr>
<tr>
<td>Assembler</td>
</tr>
<tr>
<td>Machine Language Program (MIPS)</td>
</tr>
</tbody>
</table>

- Review
  - Program can interpret binary number as unsigned integer, two’s complement signed integer, floating point number, ASCII characters, Unicode characters, ...
  - Integers have largest positive and largest negative numbers, but represent all in between
  - Two’s comp. weirdness is one extra negative number and floating point operations can lead to results too big to store within their representations: overflow/underflow
  - Floating point is an approximation of reals
  - $2^a$ patterns to represent reals from $-\infty$ to $+\infty$
  - Everything is a (binary) number in a computer

- Instructions are also kept as binary numbers in memory
  - Stored program concept
  - As easy to change programs as it is to change data
  - Register names mapped to numbers
  - Need to map instruction operation to a part of number
Names of MIPS fields

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
- **op**: Basic operation of instruction, or *opcode*
- **rs**: 1st register source operand
- **rt**: 2nd register source operand
- **rd**: Register destination operand (result of operation)
- **shamt**: Shift amount.
- **funct**: Function. This field, often called *function code*, selects the specific variant of the operation in the op field

Names of MIPS Fields in I-type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>address or constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
- **op**: Basic operation of instruction, or *opcode*
- **rs**: 1st register source operand
- **rt**: 2nd register source operand for branches but register destination operand for lw, sw, and immediate operations
- **Address/constant**: 16-bit two’s complement number
  - Note: equal in size of rd, shamt, funct fields

What about Load, Store, Immediate, Branches, Jumps?

- Fields for constants only 5 bits (-16 to +15)
  - Too small for many common cases
- #1 Simplicity favors regularity (all instructions use one format) vs. #3 Make common case fast (multiple instruction formats)?
- 4th Design Principle: *Good design demands good compromises*
- Better to have multiple instruction formats and keep all MIPS instructions same size
  - All MIPS instructions are 32 bits or 4 bytes

Register (R), Immediate (I), Jump (J) Instruction Formats

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td>op</td>
<td>rt</td>
<td>address or constant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>----</td>
<td>---------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now loads, stores, branches, and immediates can have 16-bit two’s complement address or constant: -32,768 (-2**15) to +32,767 (2**15-1)

What about jump, jump and link?

<table>
<thead>
<tr>
<th>J-type</th>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
</tr>
</tbody>
</table>

Addressing in Branches

<table>
<thead>
<tr>
<th>I-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address or constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
- Programs much bigger than 2**16 bytes, but branch address must fit in 16-bit field
  - Must specify a register for branch addresses for big programs: PC = Register + Branch address
  - Which register?
- Conditional branching for IF-statement, loops
  - Tend to be near branches; ½ within 16 instructions
- Idea: *PC-relative branching*

Addressing in Branches

<table>
<thead>
<tr>
<th>I-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address or constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
- Hardware increments PC early, so relative address is PC = (PC + 4) + Branch address
- Another optimization since all MIPS instructions 4 bytes long?
- Multiply value in branch address field by 4!
- MIPS PC-relative branching
  PC = (PC + 4) + (Branch address * 4)
Addressing in Jumps

J-type

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
- Same trick for Jumps, Jump and Link
- PC = Jump address * 4
- Since PC = 32 bits, and Jump address * 4 = 28 bits, what about other 4 bits?
- Jump and Jump and Link only changes bottom 28 bits of PC

Assembler

- Input: Assembly Language Code (e.g., foo.s for MIPS)
- Output: Object Code, Information tables (e.g., foo.o for MIPS)
- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

Converting C to MIPS Machine code

&A=ST1 (reg 9), ST0 (reg 8), h=SS2 (reg 18)

A[300] = h + A[300];

Lw $t0,1200($t1)
addu $t0,$s2,$t0
sw $t0,1200($t1)

R-type

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

I-type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>address or constant</th>
</tr>
</thead>
</table>

J-type

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
</table>
• Lab #4 posted
• Project #1 Due Sunday @ 11:59:59
• This week should be easier – given jar file of MR
  – Run at scale, and compare to your code
• Midterm is now on the horizon:
  – No discussion during exam week
  – TA Review: Su, Mar 4, starting 2 PM, 2050 VLSB
  – Exam: Tu, Mar 6, 6:40-9:40 PM, 2050 VLSB (room change)
  – Small number of special consideration cases, due to class conflicts, etc.—contact me

My Life Beyond Computing

• 1938 Ford Cabriolet
• 454 cubic inch (7500 cc) “Big Block” Chevy V8 Engine

Converting C to MIPS Machine code

&A=st1 (reg 9), st0 (reg 8), h=st2 (reg 18)
A[300] = h + A[300];

<table>
<thead>
<tr>
<th>Instruction</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0,1200($t1)</td>
<td>9</td>
<td>8</td>
<td>1200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addu $t0,$s2,$t0</td>
<td>18</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $t0,1200($t1)</td>
<td>9</td>
<td>8</td>
<td>1200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

My Life Beyond Computing

• 1935 Ford Station Wagon
• 221 cubic inch (85 HP) original Ford Flathead V8 Engine

Midterm is now on the horizon:

Project #1 Due Sunday @ 11:59:59
Lab #4 posted
Converting to MIPS Machine code

<table>
<thead>
<tr>
<th>Add</th>
<th>Loop:</th>
<th>Format?</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>sll $t1,$s3,2</td>
<td></td>
</tr>
<tr>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
<td></td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</td>
<td></td>
</tr>
<tr>
<td>812</td>
<td>bne $t0,$s5,Exit</td>
<td></td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td></td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td></td>
</tr>
</tbody>
</table>

Exit:

R-type

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

I-type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address or constant</th>
</tr>
</thead>
</table>

J-type

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
</table>

32 bit constants in MIPS

- Can create a 32-bit constant from two 32-bit MIPS instructions
- **Load Upper Immediate** (lui or “Louie”) puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register $s0:
  0000 0000 0011 1101 1001 0000 0000 two
  lui $s0, 61 # 61 = 0000 0000 0011 1101 two
  ori $s0, $s0, 2304 # 2304 = 0000 1001 0000 0000 two

Assembly and Pseudo-instructions

- Turning textual MIPS instructions into machine code called **assembly**, program called **assembler**
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called **assembly language**
- Can also accept instructions convenient for programmer but not in hardware
  - **Load immediate** (l) allows 32-bit constants, assembler turns into lui + ori (if needed)
  - **Load double** (ld) uses two lw instructions to load a pair of 32-bit floating point registers
  - Called **Pseudo-Instructions**

Assembler Directives (p. B-5 to B-7)

- 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
  - `.word w1…wn`: Store the n 32-bit quantities in successive memory words

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination
  - move $t0, $t1 → add $t0, $zero, $t1
  - blt $t0, $t1, L → slt $at, $t0, $t1, bne $at, $zero, L
  - $at (register 1): assembler temporary
More Pseudoinstructions

• Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:

- addu $t0,$t6,1
- subu $sp,$sp,32
- sd $a0, 32($sp)
- la $a0, str

Real:

- ________________

Producing an Object Module

• Assembler (or compiler) translates program into machine instructions

• Provides information for building a complete program from the pieces

  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Separate Compilation and Assembly

• No need to compile all code at once
• How put pieces together?

Translation and Startup

Many compilers produce object modules directly

Static linking

Linker Stitches Files Together

• Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
• Often a slower than compiling
  — all the machine code files must be read into memory and linked together
Loading a Program

• Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space (cover later in semester)
  3. Copy text and initialized data into memory
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     • Copies arguments to $a0, ... and calls main
     • When main returns, do "exit" systems call

Unoptimized MIPS Code

```
    .text
    .globl main
    .type main, @function

#define ARRAY_SIZE 20000

int main() {  
    int iarray[ARRAY_SIZE], x, y, holder;
    for(x = 0; x < ARRAY_SIZE; x++)
        for(y = 0; y < ARRAY_SIZE-1; y++)
            if((iarray[y] > iarray[y+1]) {  
                holder = iarray[y+1];
                iarray[y+1] = iarray[y];
                iarray[y] = holder;
            }
}
```

What’s a Compiler?

• Compiler: a program that accepts as input a program text in certain language and produces as output a program text in another language, while preserving the meaning of that text.
  • The text must comply with the syntax rules of whichever programming language it is written in.
  • A compiler’s complexity depends on the syntax of the language and how much abstraction that programming language provides.
    -- A C compiler is much simpler than C++ Compiler
  • Compiler executes before compiled program runs

Compiled Languages: Edit-Compile-Link-Run

Editor  
Source code  
Compiler  
Object code  
Linker  
Executable program

What is Typical Benefit of Compiler Optimization?

• What is a typical program?
• For now, try a toy program: BubbleSort.c
-O2 optimized MIPS Code

```
li $13,65536
ori $13,$13,0x3890
addu $13,$13,$sp
sw $28,0($13)
move $4,$0
addu $8,$sp,16
$L6:
move $3,$0
addu $9,$4,1
.p2align 3
$L10:
sll $2,$3,2
addu $6,$8,$2
addu $7,$3,1
sll $2,$7,2
addu $5,$8,$2
lw $3,0($6)
lw $4,0($5)
slt $2,$4,$3
beq $2,$0,$L9
sw $3,0($5)
sw $4,0($6)
$L9:
move $3,$7
slt $2,$3,19999
bne $2,$0,$L10
move $4,$9
slt $2,$4,20000
bne $2,$0,$L6
li $12,65536
ori $12,$12,0x38a0
addu $13,$12,$sp
addu $sp,$sp,$12
j $31
```

What’s an Interpreter?

- It reads and executes source statements executed one at a time
  - No linking
  - No machine code generation, so more portable
- Start executing quicker, but run much more slowly than compiled code
- Performing the actions straight from the text allows better error checking and reporting to be done
- The interpreter stays around during execution
  - Unlike compiler, some work is done after program starts
- Writing an interpreter is much less work than writing a compiler

Interpreted Languages:

**Edit-Run**

<table>
<thead>
<tr>
<th>Editor</th>
<th>Source code</th>
<th>Interpreter</th>
</tr>
</thead>
</table>

Compiler vs. Interpreter

**Advantages**

**Compilation:**
- Faster execution
- Single file to execute
- Compiler can do better diagnosis of syntax and semantic errors, since it has more info than an interpreter (Interpreter only sees one line at a time)
- Can find syntax errors before run program
- Compiler can optimize code

**Interpreter:**
- Easier to debug program
- Faster development time

**Disadvantages**

**Compilation:**
- Harder to debug program
- Takes longer to change source code, recompile, and relink

**Interpreter:**
- Slower execution times
- No optimization
- Need all of source code available
- Source code larger than executable for large systems
- Interpreter must remain installed while the program is interpreted

Java’s Hybrid Approach: Compiler + Interpreter

- A Java compiler converts Java source code into instructions for the **Java Virtual Machine (JVM)**
- These instructions, called **bytecodes**, are same for any computer / OS
- A CPU-specific Java interpreter interprets bytecodes on a particular computer
Java’s Compiler + Interpreter

Why Bytecodes?

- Platform-independent
- Load from the Internet faster than source code
- Interpreter is faster and smaller than it would be for Java source
- Source code is not revealed to end users
- Interpreter performs additional security checks, screens out malicious code

JVM uses Stack vs. Registers

```
a = b + c;
=>
iload b ; push b onto Top Of Stack (TOS)
iload c ; push c onto Top Of Stack (TOS)
iadd ; Next to top Of Stack (NOS) = Top Of Stack (TOS) + NOS
istore a ; store TOS into a and pop stack
```

Java Bytecodes (Stack) vs. MIPS (Reg.)

Review

- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept
- Assemblers can enhance machine instruction set to help assembly-language programmer
- Translate from text that easy for programmers to understand into code that machine executes efficiently: Compilers, Assemblers
- Linkers allow separate translation of modules
- Interpreters for debugging, but slow execution
- Hybrid (Java): Compiler + Interpreter to try to get best of both
- Compiler Optimization to relieve programmer

Starting Java Applications

```
Compiles bytecodes of "hot" methods into native code for host machine
```

```
Just In Time (JIT) compiler translates bytecode into machine language just before execution
```