Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions
Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt

- Store conditional: `sc rt, offset(rs)`

Example: atomic swap (to test/set lock variable)

```assembly
try: add $t0,$zero,$s4 ; copy exchange value
    ll $t1,0($s1) ; load linked
    sc $t0,0($s1) ; store conditional
    beq $t0,$zero,try ; branch store fails
    add $s4,$zero,$t1 ; put load value in $s4
```

Location offset(rs) is the lock
Many compilers produce object modules directly

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory

Static linking
Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination

\[
\text{move } \$t0, \$t1 \rightarrow \text{add } \$t0, \$\text{zero}, \$t1 \\
\text{blt } \$t0, \$t1, L \rightarrow \text{slt } \$\text{at}, \$t0, \$t1 \\
\text{bne } \$\text{at}, \$\text{zero}, L
\]

- \$\text{at} (register 1): assembler temporary
Produce 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
Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space
Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  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 syscall
Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

a. First call to DLL routine

b. Subsequent calls to DLL routine
Starting Java Applications

Chapter 2 — Instructions: Language of the Computer — 11
C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function

- Swap procedure (leaf)
  ```c
  void swap(int v[], int k)
  {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
  }
  ```

- v in $a0, k in $a1, temp in $t0

Swaps v[k] and v[k+1]
# The Procedure Swap

```
swap:  sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

v in $a0, k in $a1, temp in $t0
The Sort Procedure in C

- Non-leaf (calls swap)

```c
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
}
```

- v in $a0, k in $a1, i in $s0, j in $s1
The Procedure Body for sort
(see next slide for save/restore registers)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $s2, $a0</td>
<td># save $a0 into $s2</td>
</tr>
<tr>
<td>move $s3, $a1</td>
<td># save $a1 into $s3</td>
</tr>
<tr>
<td>move $s0, $zero</td>
<td># $i = 0</td>
</tr>
<tr>
<td>for1tst: slt $t0, $s0, $s3</td>
<td># $t0 = 0 if $s0 $s3 (i $n)</td>
</tr>
<tr>
<td>beq $t0, $zero, exit1</td>
<td># go to exit1 if $s0 $s3 (i $n)</td>
</tr>
<tr>
<td>addi $s1, $s0, -1</td>
<td># $j = i - 1</td>
</tr>
<tr>
<td>for2tst: slti $t0, $s1, 0</td>
<td># $t0 = 1 if $s1 &lt; 0 (j &lt; 0)</td>
</tr>
<tr>
<td>bne $t0, $zero, exit2</td>
<td># go to exit2 if $s1 &lt; 0 (j &lt; 0)</td>
</tr>
<tr>
<td>sll $t1, $s1, 2</td>
<td># $t1 = j * 4</td>
</tr>
<tr>
<td>add $t2, $s2, $t1</td>
<td># $t2 = v + (j * 4)</td>
</tr>
<tr>
<td>lw $t3, 0($t2)</td>
<td># $t3 = v[j]</td>
</tr>
<tr>
<td>lw $t4, 4($t2)</td>
<td># $t4 = v[j + 1]</td>
</tr>
<tr>
<td>slt $t0, $t4, $t3</td>
<td># $t0 = 0 if $t4 $t3</td>
</tr>
<tr>
<td>beq $t0, $zero, exit2</td>
<td># go to exit2 if $t4 $t3</td>
</tr>
<tr>
<td>move $a0, $s2</td>
<td># 1st param of swap is v (old $a0)</td>
</tr>
<tr>
<td>move $a1, $s1</td>
<td># 2nd param of swap is j</td>
</tr>
<tr>
<td>jal swap</td>
<td># call swap procedure</td>
</tr>
<tr>
<td>addi $s1, $s1, -1</td>
<td># $j = $j - 1</td>
</tr>
<tr>
<td>j for2tst</td>
<td># jump to test of inner loop</td>
</tr>
<tr>
<td>exit2: addi $s0, $s0, 1</td>
<td># $i = $i + 1</td>
</tr>
<tr>
<td>j for1tst</td>
<td># jump to test of outer loop</td>
</tr>
</tbody>
</table>

Chapter 2 — Instructions: Language of the Computer — 15
The Full Procedure

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort: addi $sp,$sp, −20</td>
<td># make room on stack for 5 registers</td>
</tr>
<tr>
<td>sw $ra, 16($sp)</td>
<td># save $ra on stack</td>
</tr>
<tr>
<td>sw $s3,12($sp)</td>
<td># save $s3 on stack</td>
</tr>
<tr>
<td>sw $s2, 8($sp)</td>
<td># save $s2 on stack</td>
</tr>
<tr>
<td>sw $s1, 4($sp)</td>
<td># save $s1 on stack</td>
</tr>
<tr>
<td>sw $s0, 0($sp)</td>
<td># save $s0 on stack</td>
</tr>
<tr>
<td>...</td>
<td># procedure body</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>exit1: lw $s0, 0($sp)</td>
<td># restore $s0 from stack</td>
</tr>
<tr>
<td>lw $s1, 4($sp)</td>
<td># restore $s1 from stack</td>
</tr>
<tr>
<td>lw $s2, 8($sp)</td>
<td># restore $s2 from stack</td>
</tr>
<tr>
<td>lw $s3,12($sp)</td>
<td># restore $s3 from stack</td>
</tr>
<tr>
<td>lw $ra,16($sp)</td>
<td># restore $ra from stack</td>
</tr>
<tr>
<td>addi $sp,$sp, 20</td>
<td># restore stack pointer</td>
</tr>
<tr>
<td>jr $ra</td>
<td># return to calling routine</td>
</tr>
</tbody>
</table>
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

Perf = inv of cycles

ICxCPI=cycles
Effect of Language and Algorithm

- **Bubblesort Relative Performance**
  - C/none: 1
  - C/O1: 2.5
  - C/O2: 2.5
  - C/O3: 2
  - Java/int: 1.5
  - Java/JIT: 2

- **Quicksort Relative Performance**
  - C/none: 1
  - C/O1: 2.5
  - C/O2: 2.5
  - C/O3: 3
  - Java/int: 1
  - Java/JIT: 1

- **Quicksort vs. Bubblesort Speedup**
  - C/none: 2500
  - C/O1: 1500
  - C/O2: 1000
  - C/O3: 2000
  - Java/int: 500
  - Java/JIT: 100

Java/int is the slowest.
Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!
Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity
**Example: Clearing and Array**

<table>
<thead>
<tr>
<th>clear1(int array[], int size) {</th>
<th>clear2(int *array, int size) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i;</td>
<td>int *p;</td>
</tr>
<tr>
<td>for (i = 0; i &lt; size; i += 1)</td>
<td>for (p = &amp;array[0]; p &lt; &amp;array[size]; p = p + 1)</td>
</tr>
<tr>
<td>array[i] = 0;</td>
<td>*p = 0;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>move $t0,$zero</th>
<th>move $t0,$a0</th>
</tr>
</thead>
<tbody>
<tr>
<td># i = 0</td>
<td># p = &amp; array[0]</td>
</tr>
</tbody>
</table>

**loop1:**

<table>
<thead>
<tr>
<th>sll $t1,$t0,2</th>
<th>sll $t1,$a1,2</th>
</tr>
</thead>
<tbody>
<tr>
<td># $t1 = i * 4</td>
<td># $t1 = size * 4</td>
</tr>
<tr>
<td>add $t2,$a0,$t1</td>
<td>add $t2,$a0,$t1</td>
</tr>
<tr>
<td># $t2 = &amp;array[i]</td>
<td># $t2 = &amp;array[size]</td>
</tr>
<tr>
<td>sw $zero, 0($t2)</td>
<td>sw $zero, 0($t0)</td>
</tr>
<tr>
<td># array[i] = 0</td>
<td># Memory[p] = 0</td>
</tr>
<tr>
<td>addi $t0,$t0,1</td>
<td>add $t0,$t0,4</td>
</tr>
<tr>
<td># i = i + 1</td>
<td># p = p + 4</td>
</tr>
<tr>
<td>slt $t3,$t0,$a1</td>
<td>slt $t3,$t0,$t2</td>
</tr>
<tr>
<td># $t3 = (i &lt; size)</td>
<td># (p &lt; &amp;array[size])</td>
</tr>
<tr>
<td>bne $t3,$zero,loop1</td>
<td>bne $t3,$zero,loop2</td>
</tr>
<tr>
<td># if (...)</td>
<td># if (...)</td>
</tr>
<tr>
<td># goto loop1</td>
<td># goto loop2</td>
</tr>
</tbody>
</table>

Chapter 2 — Instructions: Language of the Computer — 21
Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer
ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>15 × 32-bit</td>
<td>31 × 32-bit</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result

- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions
Instruction Encoding

Chapter 2 — Instructions: Language of the Computer — 25
The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments
The Intel x86 ISA

Further evolution…

- i486 (1989): pipelined, on-chip caches and FPU
  - Compatible competitors: AMD, Cyrix, …
- Pentium (1993): superscalar, 64-bit datapath
  - Later versions added MMX (Multi-Media eXtension) instructions
  - The infamous FDIV bug
  - New microarchitecture (see Colwell, *The Pentium Chronicles*)
- Pentium III (1999)
  - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions
The Intel x86 ISA

- And further...
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead...
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions

- If Intel didn’t extend with compatibility, its competitors would!
  - Technical elegance ≠ market success
## Basic x86 Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPR 0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPR 1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPR 2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPR 3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPR 4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPR 5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPR 6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPR 7</td>
</tr>
</tbody>
</table>

- **CS**: Code segment pointer
- **SS**: Stack segment pointer (top of stack)
- **DS**: Data segment pointer 0
- **ES**: Data segment pointer 1
- **FS**: Data segment pointer 2
- **GS**: Data segment pointer 3
- **EIP**: Instruction pointer (PC)
- **EFLAGS**: Condition codes
Basic x86 Addressing Modes

- Two operands per instruction

<table>
<thead>
<tr>
<th>Source/dest operand</th>
<th>Second source operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

- Memory addressing modes

- Address in register

- Address = \( R_{base} + \text{displacement} \)

- Address = \( R_{base} + 2^{\text{scale}} \times R_{index} \) (scale = 0, 1, 2, or 3)

- Address = \( R_{base} + 2^{\text{scale}} \times R_{index} + \text{displacement} \)
x86 Instruction Encoding

- Variable length encoding
- Postfix bytes specify addressing mode
- Prefix bytes modify operation
  - Operand length, repetition, locking, …
Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions
ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions
Fallacies

- Powerful instruction $\Rightarrow$ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity
Fallacies

- Backward compatibility $\Rightarrow$ instruction set doesn’t change
  - But they do accrete more instructions

x86 instruction set
Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped
Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

- Layers of software/hardware
  - Compiler, assembler, hardware

- MIPS: typical of RISC ISAs
  - c.f. x86
Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>