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)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)
  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
Many compilers produce object modules directly.

Static linking
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
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
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

Chapter 2 — Instructions: Language of the Computer — 65
Starting Java Applications

Java program

Compiler

Class files (Java bytecodes)

Java Virtual Machine

Java library routines (machine language)

Compiled Java methods (machine language)

Interprets bytecodes

Simple portable instruction set for the JVM

Compiles bytecodes of “hot” methods into native code for host machine
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
The Procedure Swap

```
swap: sll $t1, $a1, 2       # $t1 = k * 4
    add $t1, $a0, $t1     # $t1 = v+(k*4)
    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
```
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

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</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><strong>for1tst:</strong> 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><strong>for2tst:</strong> 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><strong>exit2:</strong> addi $s0, $s0, 1</td>
<td># i += 1</td>
</tr>
<tr>
<td>j for1tst</td>
<td># jump to test of outer loop</td>
</tr>
</tbody>
</table>
The Full Procedure

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

Compiled with gcc for Pentium 4 under Linux

- **Relative Performance**
  - None: 1.5
  - O1: 2.0
  - O2: 2.5
  - O3: 3.0

- **Instruction count**
  - None: 120,000
  - O1: 100,000
  - O2: 80,000
  - O3: 60,000

- **Clock Cycles**
  - None: 160,000
  - O1: 140,000
  - O2: 120,000
  - O3: 100,000

- **CPI**
  - None: 1.5
  - O1: 2.0
  - O2: 1.5
  - O3: 1.0
Effect of Language and Algorithm

- **Bubblesort Relative Performance**
- **Quicksort Relative Performance**
- **Quicksort vs. Bubblesort Speedup**

For the graphs:
- Y-axis represents relative performance or speedup.
- X-axis lists different language and optimization levels (C/none, C/O1, C/O2, C/O3, Java/int, Java/JIT).

Chapter 2 — Instructions: Language of the Computer — 73
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 an Array

clear1(int array[], int size) {
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}
clear2(int *array, int size) {
    int *p;
    for (p = &array[0]; p < &array[size];
        p = p + 1)
        *p = 0;
}

move $t0,$zero     # i = 0
loop1:  sll $t1,$t0,2    # $t1 = i * 4
        add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
        sw $zero, 0($t2) # array[i] = 0
        addi $t0,$t0,1   # i = i + 1
        slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
        bne $t3,$zero,loop1 # if (...)
                        # goto loop1
move $t0,$a0       # p = & array[0]
loop2:  sll $t1,$a1,2    # $t1 = size * 4
        add $t2,$a0,$t1  # $t2 =
                        #   &array[size]
        sw $zero,0($t0) # Memory[p] = 0
        addi $t0,$t0,4   # p = p + 4
        slt $t3,$t0,$t2  # $t3 =
                        #   (p<&array[size])
        bne $t3,$zero,loop2 # if (...)
                        # goto loop2
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
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
  - **Pentium Pro (1995), Pentium II (1997)**
    - 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>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
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 = \( R_{base} + 2^{scale} \times R_{index} \) (scale = 0, 1, 2, or 3)
  - Address = \( R_{base} + 2^{scale} \times R_{index} + \text{displacement} \)
## x86 Instruction Encoding

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE EIP + displacement</td>
<td>4 4 8</td>
<td>Condition Displacement</td>
</tr>
<tr>
<td>CALL</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>MOV EBX, [EDI + 45]</td>
<td>6 1 1 8</td>
<td>d w imm Postbyte Displacement</td>
</tr>
<tr>
<td>PUSH ESI</td>
<td>5 3</td>
<td></td>
</tr>
<tr>
<td>ADD EAX, #6765</td>
<td>4 5 1</td>
<td>Reg w Immediate</td>
</tr>
<tr>
<td>TEST EDX, #42</td>
<td>7 1 8</td>
<td>w Postbyte Immediate</td>
</tr>
</tbody>
</table>
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
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

![Graph showing the number of instructions over years for the x86 instruction set.](image-url)
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>