Putting it All Together

Alexander Nelson
February 12, 2020

University of Arkansas - Department of Computer Science and Computer Engineering
Bit by bit, putting it together... Piece by piece, only way to make a work of art. Every moment makes a contribution, Every little detail plays a part. Having just the vision’s no solution, Everything depends on execution, Putting it together, that’s what counts.

-Stephen Sondheim
Sunday in the Park with George
Putting it All Together

What do we know so far?

- Digital circuits built from transistors
- Combinations of circuits with synchronous clock can perform general computation
- Instructions can be used to direct general purpose computation
- Instructions are provided to hardware in binary, but translated from assembly language to machine code
- A consistent, simple ISA enables programmers to abstract knowledge of the hardware
- **Assembly isn’t a lot of fun to write high level code in**
Translating and Starting a Program
Putting it All Together

Many compilers produce object modules directly

C program

Assembly language program

Object: Machine language module

Object: Library routine (machine language)

Executable: Machine language program

Loader

Memory

Static linking
Compiler – program that takes a high level language and translates to assembly

In 1975, OS and assemblers written in assembly language
Why?
Compilers were inefficient & memory expensive

What changed?
DRAM has increased >1MX, Optimizing compilers can outperform all but expert assembly language coders
Compiler

High-level language – usually platform independent, easier to read/write
Assembly – ISA, platform dependent, more difficult to read/write

# of concurrent ISAs/architectures make writing all applications in assembler impossible

**Takeaway:** Compilers are vital. We won’t talk about how to make one in here.
Assembler – A program that converts assembly to machine code
Assembly – A representation of instructions provided ISA, but critically not the actual ISA

What does this mean?
Not all assembly commands need to be in the ISA

Pseudoinstruction – an instruction available in assembly with no actual ISA instruction
Example:
\[
\text{move } \$t0, \$t1 \rightarrow \text{add } \$t0, \$zero, \$t1 \\
\text{blt } \$t0, \$t1, L \rightarrow \text{slt } + \text{bne}
\]
Let's look at that last instruction again

\[ blt \; t0, \; t1, \; L \rightarrow slt + bne \]

\[ slt \; R?, \; t0, \; t1 \]
\[ bne \; R?, \; zero, \; L \]

Where is R?
Let’s look at that last instruction again

\( \text{blt } t0, t1, L \rightarrow \text{slt } + \text{bne} \)

\( \text{slt } R?, t0, t1 \)

\( \text{bne } R?, \text{zero}, L \)

Where is \( R? \)?

Pseudoinstructions necessitate a temporary register reserved for assembler

R1 – \$at – “reserved for assembler”
Assembler translates program to machine instructions
Provides information to build a complete program:

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

These object files still have unresolved references
Linking object modules

Programs composed of several objects
Linking = Putting objects together into a single executable image

Includes:

- Merge segments
- Resolve labels (determine addresses)
- Patch location-dependent/external references

Executable image typically has same format as object file, with no unresolved references
Page 127-128 runs through a linking example
Loading a Program

How do you load an image file from disk to memory?
Steps:

- Read header to determine segment sizes
- Create virtual address space
- Copy text and initialized data into memory
  - Or set page table entries so they can be faulted in
- Set up arguments on stack
- Initialize registers (including $sp, $f, $gp)
- Jump to startup routine
  - Copies arguments to $a0, ..., and calls main
  - When main returns, do exit syscall

Appendix A.3, A.4 describe linking/loading in more detail
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 version
Lazy Linking

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
What about Java?

The above slides consider the C model of compiling/assembling/linking/loading.

What about Java?

- **Java program**
- **Class files (Java bytecodes)**
- **Java Virtual Machine**
- **Simple portable instruction set for the JVM**
- **Java library routines (machine language)**
- **Compiled Java methods (machine language)**
- **Interprets bytecodes**

Compiles bytecodes of “hot” methods into native code for host machine.
Java Virtual Machine (JVM) – software interpreter
Interpreter – program that simulates and ISA

Advantage: Portability – A Java file runs the same
Disadvantage: Lower performance – A factor of 10 slowdown without a “Just In Time” compiler

Just In Time (JIT) – profile running program to determine where “hot” methods are being called, compile those to native language
Put it all Together – C Sort
Example – C bubble sort

```c
void swap(int v[], int k){
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
} //Assume v in $a0, k in $a1, temp in $t0
```
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
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);
        }
    }
} //Assume v in $a0, k in $a1, i in $s0, j in $s1
move $s2, $a0  # save $a0 into $s2
move $s3, $a1  # save $a1 into $s3
move $s0, $zero  # i = 0

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

exit2: addi $s0, $s0, 1  # i += 1
j for1tst  # jump to test of outer loop
Sort Procedure – Assembly

sort:  addi $sp,$sp, 20  # make room on stack for 5 registers
sw $ra, 16($sp)  # save $ra on stack
sw $s3, 12($sp)  # save $s3 on stack
sw $s2, 8($sp)  # save $s2 on stack
sw $s1, 4($sp)  # save $s1 on stack
sw $s0, 0($sp)  # save $s0 on stack

BODY

exit1: lw $s0, 0($sp)  # restore $s0 from stack
lw $s1, 4($sp)  # restore $s1 from stack
lw $s2, 8($sp)  # restore $s2 from stack
lw $s3, 12($sp)  # restore $s3 from stack
lw $ra, 16($sp)  # restore $ra from stack
addi $sp, $sp, 20  # restore stack pointer
jr $ra  # return to calling routine
Effect of Compiler

Compiled with gcc for Pentium 4 under Linux

- **Relative Performance**
  - none: 0
  - O1: 2
  - O2: 3
  - O3: 3

- **Instruction count**
  - none: 120000
  - O1: 60000
  - O2: 40000
  - O3: 40000

- **Clock Cycles**
  - none: 180000
  - O1: 120000
  - O2: 100000
  - O3: 100000

- **CPI**
  - none: 1.5
  - O1: 2
  - O2: 1.5
  - O3: 1.5
Effect of Compiler

- **Bubblesort Relative Performance**
  - C/none: 0.5
  - C/O1: 2.5
  - C/O2: 2.5
  - C/O3: 2.5
  - Java/int: 1.0
  - Java/JIT: 2.0

- **Quicksort Relative Performance**
  - C/none: 1.0
  - C/O1: 1.5
  - C/O2: 1.5
  - C/O3: 2.5
  - Java/int: 0.1
  - Java/JIT: 0.5

- **Quicksort vs. Bubblesort Speedup**
  - C/none: 2500
  - C/O1: 1500
  - C/O2: 1500
  - C/O3: 1500
  - Java/int: 500
  - Java/JIT: 50
Compiler Lessons

Instruction Count/CPI not good performance indicators in isolation
Compiler optimizations are sensitive to algorithm
Java/JIT compiled code is significantly faster than JVM interpreted
  • Comparable to C in some cases

Nothing can fix a bad algorithm
Arrays vs. Pointers

Array indexing involves:

- Multiply index by element size
- Add offset to array base address

Pointers correspond directly to a memory location – Avoid indexing complexity
# Clear Array Example

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

| move $t0,$zero     # i = 0 | move $t0,$a0 # p = & array[0] |
| loop1: sll $t1,$t0,2 # $t1 = i * 4 | sll $t1,$a1,2 # $t1 = size * 4 |
| add $t2,$a0,$t1 # $t2 = | add $t2,$a0,$t1 # $t2 = |
|   # &array[i] |   # &array[size] |
| sw $zero, 0($t2) # array[i] = 0 | loop2: sw $zero, 0($t0) # Memory[p] = 0 |
| addi $t0,$t0,1 # i = i + 1 | addi $t0,$t0,4 # p = p + 4 |
| slt $t3,$t0,$a1 # $t3 = | slt $t3,$t0,$t2 # $t3 = |
|   # (i < size) |   #(p<&array[size]) |
| bne $t3,$zero,loop1 # if (...) | bne $t3,$zero,loop2 # if (...) |
|   # goto loop1 |   # goto loop2 |
Comparison of Array vs. Pointer

Multiply complexity reduced to a shift
Array version requires shift to be inside loop

- Part of index calculation for incremented i
- Compare to incrementing a pointer

Compiler can achieve same effect as manual use of pointers

- Induction variable elimination
- Better to make program clearer and safer
Comparison To Other ISAs
ARM & MIPS Similarities

ARM – 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>
ARM – Uses condition codes for result of arithmetic/logical instruction

- Negative, zero, carry, overflow
- Compare instructions to set condition codes without keeping results

Each instruction can then be conditional:

- Top 4 bits of instruction word (IW) = condition value
- Can avoid branches over a single instruction
## Instruction Encoding

<table>
<thead>
<tr>
<th>Mode</th>
<th>ARM Length</th>
<th>MIPS Length</th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register-register</strong></td>
<td>31-20</td>
<td>31</td>
<td>Op&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Rs&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>27-16</td>
<td>26-15</td>
<td>Rs&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Rd&lt;sup&gt;5&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>12-0</td>
<td>11-0</td>
<td>Opx&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Rs&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td><strong>Data transfer</strong></td>
<td>31-21</td>
<td>31</td>
<td>Opx&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Rs&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>20-16</td>
<td>16-15</td>
<td>Rs&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Rd&lt;sup&gt;5&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>12-0</td>
<td>11-0</td>
<td>Opx&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;12&lt;/sup&gt;</td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td>31-26</td>
<td>31</td>
<td>Opx&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Rs&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>24-19</td>
<td>25-15</td>
<td>Rs&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Opx&lt;sup&gt;5&lt;/sup&gt;/Rs&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>12-0</td>
<td>11-0</td>
<td>Opx&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;16&lt;/sup&gt;</td>
</tr>
<tr>
<td><strong>Jump/Call</strong></td>
<td>31-26</td>
<td>31</td>
<td>Opx&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Rs&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>24-19</td>
<td>25-15</td>
<td>Rs&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Opx&lt;sup&gt;6&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>12-0</td>
<td>11-0</td>
<td>Opx&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;26&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
Intel x86 ISA

A story of evolution combined w/ backward compatibility:

8080 (1974) – 8-bit microprocessor
  - Accumulator, plus 3 index-register pairs

8086 (1978) – 16-bit microprocessor extension to 8080
  - Complex instruction set (CISC)

8087 (1980) – floating-point coprocessor (+~80 instructions)
  - Adds FP instructions and register stack

80286 (1982) – 24-bit addresses + memory-mapped protection
  - 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

i486 (1989) – pipelined, on-chip caches and FP unit

- Compatible competitors: AMD, Cyrix

Pentium (1993) – superscalar, 64-bit datapath (+4 instructions)

- Later versions add MMX (multimedia extension +57 instructions)
- Infamous FDIV bug – Small bug in floating point division – resulted in \(~\$475M\) loss
Intel x86 ISA


- New microarchitecture, (+4 instructions)
- Expanded Pentium II and pro with MMX

Pentium III (1999)

- Added SSE (Streaming SIMD Extensions) and associated registers (+70 instructions)

Pentium 4 (2001)

- New microarchitecture
- Added SSE2 instructions (+144 instructions)
AMD64 (2003) – extended x86 to 64 bits EM64T – Extended Memory 64 Technology (2004)

- AMD64 adopted by Intel (with refinements (including an atomic compare and swap))
- Added SSE3 instructions (+13 instructions)

Intel Core (2006)

- Added SSE4 instructions, virtual machine support (+54 instructions)

AMD64 (announced 2007) – SSE5 instructions (+170 instructions) Intel declines following Advanced Vector Extension (announced 2008)

- Longer SSE registers, more instructions (250 instructions refined, +128 new instructions)
Lessons:

- If/When Intel didn’t extend maintaining compatibility, its competitors did
- Technical elegance $\neq$ Market success
80386 extended 16-bit regs to 32, prefixing E
80386 has only 8 GPRs
Intel x86 Basic 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 = $R_{base} + s^{scale} \times R_{index}$ (scale = 0,1,2,3)
- Address = $R_{base} + s^{scale} \times R_{index} + \text{displacement}$
x86 Instruction Encoding

Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
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

- Why?
- Compilers avoid complex instructions
ARM v8 Instructions

ARM → 64-bit = complete overhaul
ARM v8 resembles MIPS
Changes from v7:

- No conditional execution field
- Immediate field is 12-bit constant
- Dropped load/store multiple
- PC no longer a GPR
- GPR set expanded to 32
- Addressing modes work for all word sizes
- +Divide instruction
- +beq/bne instructions
Fallacies/Pitfalls
Fallacies

1) Powerful instruction == higher performance
   • Fewer instructions required! but...
   • Complex instructions hard to implement
   • May slow down all instructions, including simple ones
   • Compilers are good at making fast code from simple instructions

2) Use Assembly code for high performance
   • Modern compilers better at dealing with modern processors
   • More lines of code == more errors/less productivity
3) Backwards Compatibility == ISA doesn’t change

- Maintain compatibility, but add instructions
1) Sequential words are not at sequential addresses
   - For 4-byte IW, increment by 4, not 1

2) Keeping a pointer to an automatic variable after procedure returns
   - Passing pointer back via argument
   - Pointer becomes invalid when stack popped
   - This is a really hard bug, because it works sometimes!
Chapter 2 Concluding Remarks

Stick to design principles!

- Simplicity favors regularity
- Smaller \( \rightleftharpoons \) faster
- Make the common case fast
- Good design demands good compromises

Layers of software/hardware

- Compiler, Assembler, Hardware, Algorithm all important

MIPS – A typical RISC ISA, good to study how to design ISAs

- More difficult to study x86 for example
Chapter 2 Concluding Remarks

Measure MIPS instruction execution in benchmark programs

- Consider making 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>