

MIPS

CS153: Compilers
Greg Morrisett

Notes:

- Problem Set 1 due Friday before class
 - SML exercises
 - MIPS interpreter
 - Problems? Email Gregory or me.
- Problem Set 2 coming soon:
 - Given: AST for a small, Fortran-like language
 - expressions & statements only
 - Your task: build lexer and parser
 - using ML-Lex and Yacc

Today:

Quick overview of the MIPS instruction set.

- We're going to be compiling to MIPS assembly language.
- So you need to know how to program at the MIPS level.
- Helps to have a bit of architecture background to understand *why* MIPS assembly is the way it is.

There's an online manual that describes things in gory detail.

MIPS

- Reduced Instruction Set Computer (RISC)
 - Load/store architecture
 - All operands are either registers or constants
 - All instructions same size (4 bytes) and aligned on 4-byte boundary.
 - Simple, orthogonal instructions
 - e.g., no `subi`, (`addi` and negate value)
 - All registers (except \$0) can be used in all instructions.
 - Reading \$0 always returns the value 0
- Easy to make fast: pipeline, superscalar

x86

- Complex Instruction Set Computer (CISC)
 - Instructions can operate on memory values
 - e.g., `add [eax],ebx`
 - Complex, multi-cycle instructions
 - e.g., `string-copy`, `call`
 - Many ways to do the same thing
 - e.g., `add eax,1` `inc eax`, `sub eax,-1`
 - Instructions are variable-length (1-10 bytes)
 - Registers are not orthogonal
- Hard to make fast...(but they do anyway)

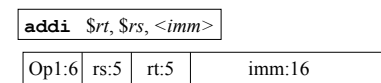
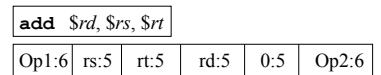
Tradeoffs

- x86 (as opposed to MIPS):
 - Lots of existing software.
 - Harder to decode (i.e., parse).
 - Harder to assemble/compile to.
 - Code can be more compact (3 bytes on avg.)
 - I-cache is more effective...
 - Easier to add new instructions.
- Today's implementations have the best of both:
 - Intel & AMD chips suck in x86 instructions and compile them to "micro-ops", caching the results.
 - Core execution engine more like MIPS.

MIPS instructions:

- Arithmetic & logical instructions:
 - `add`, `sub`, `and`, `or`, `sll`, `srl`, `sra`, ...
 - Register and immediate forms:
 - `add $rd, $rs, $rt`
 - `addi $rd, $rs, <16-bit-immed>`
 - Any registers (except \$0 returns 0)
 - Also a distinction between overflow and no-overflow (we'll ignore for now.)

Encodings:



Movement:

- Assembler provides pseudo-instructions:

`move $rd, $rs → or $rd, $rs, $0`

`li $rd, <32-bit-imm> →
lui $rd, <hi-16-bits>
ori $rd, $rd, <lo-16-bits>`

MIPS instructions:

- Multiply and Divide
 - Use two special registers `mflo`, `mghi`
 - i.e., `mul $3, $5` produces a 64-bit value which is placed in `mghi` and `mflo`.
 - Instructions to move values from `mflo/mhi` to the general purpose registers `$r` and back.
 - Assembler provides pseudo-instructions:
 - `mult $2, $3, $5` expands into:


```
mul $3,$5
mflo $2
```

MIPS instructions:

- Load/store
 - `lw $rd, <imm>($rs)` ; $rd := \text{Mem}[rs+imm]$
 - `sw $rs, <imm>($rt)` ; $\text{Mem}[rt+imm] := rs$
- Traps (fails) if $rs+imm$ is not word-aligned.
- Other instructions to load bytes and half-words.

Conditional Branching:

- `beq $rs,$rt, <imm16>`
if $\$rs == \rt then $pc := pc + imm16$
- `bne $rs,$rt, <imm16>`
- `b <imm16> == beq $0,$0, <imm16>`
- `bgez $rs, <imm16>`
if $\$rs \geq 0$ then $pc := pc + imm16$
- Also `bgtz`, `blez`, `bltz`
- Pseudo instructions:
`b<comp> $rs,$rt, <imm16>`

In Practice:

Assembler lets us use symbolic labels instead of having to calculate the offsets.

Just as in BASIC, you put a label on an instruction and then can branch to it:

```
LOOP: ...
      bne $3, $2, LOOP
Assembler figures out actual offsets.
```

Tests:

- `slt $rd, $rs, $rt` ; `rd := (rs < rt)`
- `slt $rd, $rs, <imm16>`
- Additionally: `sltu, sltiu`
- Assembler provides pseudo-instructions for `seq, sge, sgeu, sgt, sne, ...`

Unconditional Jumps:

- `j <imm26>` ; `pc := (imm26 << 2)`
- `jr $rs` ; `pc := $rs`
- `jal <imm26>` ; `$31 := pc+4 ;`
`pc := (imm26 << 2)`
- Also, `jalr` and a few others.
- Again, in practice, we use labels:
`fact: ...`
`main: ...`
`jal fact`

Other Kinds of Instructions:

- Floating-point (separate registers `$fj`)
- Traps
- OS-trickery

An Example:	
<pre>int sum(int n) { int s = 0; for (; n != 0; n--) s += n; }</pre>	<pre>sum: ori \$2,\$0,\$0 b test loop: add \$2,\$2,\$4 subi \$4,\$4,1 test: bne \$4,\$0,loop j \$31</pre>
<pre>int main() { return sum(42); }</pre>	<pre>main: ori \$4,\$0,42 move \$17,\$31 jal sum jr \$17</pre>

Better:	
<pre>int sum(int n) { int s = 0; for (; n != 0; n--) s += n; }</pre>	<pre>sum: ori \$2,\$0,\$0 b test loop: add \$2,\$2,\$4 subi \$4,\$4,1 test: bne \$4,\$0,loop j \$31</pre>
<pre>int main() { return sum(42); }</pre>	<pre>main: ori \$4,\$0,42 j sum</pre>

One Final Point

We're going to program to the MIPS *virtual* machine which is provided by the assembler.

- lets us use macro instructions, labels, etc.
- (but we must leave a scratch register for the assembler to do its work.)
- lets us ignore *delay slots*.
- (but then we pay the price of not scheduling those slots.)