Instruction Set Architectures 2

Introduction to Computer Architecture
David Black-Schaffer

Contents

• Translation to machine code
  – Instruction formats
  – Large constants

• Procedure calls
  – Register conventions
  – Stack memory

• Other ISAs

Material that is not in this lecture

Readings from the book
  – Sign extension for two’s complement numbers (2.4)
  – Logical operations (2.6)
  – Assembler, linker, and loader (2.12)

You will need 2.4 and 2.6 for this lecture.
(2.12 will be on the exam.)

The book has excellent descriptions of these topics. Please read the book before watching this lecture.
Translation to machine code

Encodings and formats

Instruction format (machine language)

- Machine Language
  - Computers do not understand "add R8, R17, R18"
  - Instructions are translated to machine language (1s and 0s)

- Example:
  - add R8, R17, R18 →
    00000010 00110010 01000000 00100000

- MIPS instructions have logical fields:

<table>
<thead>
<tr>
<th>opcode</th>
<th>rt (rs1)</th>
<th>rt (rs2)</th>
<th>rd (dest)</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>100000</td>
<td>100000</td>
<td>000000</td>
<td>100000</td>
<td>000000</td>
</tr>
</tbody>
</table>

Instruction fields

- **opcode**: Operation (e.g., "add" "lw")
- **rs**: First source register
- **rt**: Second source register
- **rd**: Destination register
- **shamt**: Shift amount
- **funct**: Function selector (add = 32, sub = 34)

Remember from 2’s complement: subtraction is basically addition, so it makes sense to share an opcode.
Constants (immediate)

- Small constants (immediates) are used all over code (~50%)
  - Small constants (immediates) are used all over code (~50%)
  - Small constants (immediates) are used all over code (~50%)
- How can we support this in the processor?
  - Put the “typical constants” in memory and load them (slow)
  - Create hard-wired registers (like R0) for them (how many?)
- MIPS does something in between:
  - Some instructions can have constants inside the instruction
  - The control logic then sends the constants to the ALU
- But there’s a problem:
  - In instructions have only 32 bits. Need some for opcode and registers.
  - How do we tradeoff space for constants and instructions?

MIPS instruction formats

- Different formats for different kinds of instructions
- MIPS has 3 instruction formats:
  - I: operation 2 registers short immediate
  - J: jump 0 registers long immediate
- Formats use the instruction bits differently
  - Tradeoff immediate space and registers

Loading immediate values (constants)

- Control tells the ALU to take one operand from the Register File and the other from the Instruction.
- 16-bit immediate is sign extended to 32 bits.
Large constants and branches

Loading larger values

- The immediate field is limited to 16 bits (-32,768 to +32,767)
- How do we load larger values?
  - Use two instructions to combine two 16 bit immediates
    - Load Upper Immediate (lui): Loads upper 16 bits
    - Or Immediate (ori): Loads lower 16 bits
- Example: 10101010 10101010 11110000 11110000

  lui R2, 10101010 10101010
  ori R2, 11110000 11110000

R2: 10101010 10101010 000000 000000

Ques*on: Is the immediate sign-extended for ori?
Answer: No. If it was we would end up with all 1s in the top bits.
(See the MIPS reference data in the book.)
Addresses in branches and jumps

- Branch instructions
  - `bne/beq` I-format 16 bit immediate
  - `beq` J-format 26 bit immediate
- But addresses are 32 bits! How do we handle this?
  - Treat `bne/beq` as relative offsets (add to current PC)
  - Treat as an absolute value (replace 26 bits of the PC)

Jump addresses example: loops

```
for (j=0; j<10; j++) {
    b = b + j;
}
```

Why do relative branches work?

```
for (int i=0; i<30; i++) {
    /* more code */
}
```
What kind of branches are common?

![Bar chart showing frequency of comparison types in branches]

- LT/GE: 7% 40%
- GT/LE: 7% 21%
- EQ/NE: 57% 86%

Optimize for this case because it is most common (e.g., MIPS)

Summary: machine code and immediates

- Instructions have different encodings to store different types of data (3 register vs. immediate)
- MIPS has 3 types, for different uses
- Encodings limit how much data we can have
- These are tradeoffs in design:
  - Optimize for the common case (short immediates)
  - Support the general case (long immediates)
Procedure calls

- Procedures (functions/subroutines) are needed for structured programming

```c
main() {
    for (j=0; j<10; j++)
        if (a[j] == 0)
            a[j] = update(a[j], j);
}
```

- The difficulty is that the procedure needs to:
  1. Put data where the procedure can access it
  2. Start executing
  3. Acquire the registers needed for the procedure
  4. Execute the code
  5. Place the results back in the caller
  6. Return control to where we were before we called the procedure

But it needs to do this without messing up the caller’s registers!

More specifically:

```c
main() {
    for (j=0; j<10; j++)
        if (a[j] == 0)
            a[j] = update(a[j], j);
}
```

We need to:
1. Put the parameters in a place where the procedure (callee) can get them
2. Transfer control to the callee
3. Acquire the registers needed for the procedure
4. Execute the code
5. Place the results in a place where the calling program (caller) can access them
6. Return control to where we were before we called the procedure

...without messing up the caller’s registers!
Caller context

Example procedure:

```c
add $t1, $t2, $t3
add $t4, $t5, $t6
sub $t7, $t8, $t9
```  

If the caller (e.g., main()) uses $t1, $t2 or $t3 they would have to be saved because the caller overwrites them when it executes.

Problems:

- The callee does not know which registers the caller is using!
  (Could have multiple different callers)
- The callee does not know which registers the caller will use!  
  (Could call multiple sub-procedures)

MIPS has a convention on who saves which registers

- Divided between the callee and caller
- Following this convention allows any caller to call any callee
- Caller and callee both know what they need to save

Saving registers: MIPS conventions

- MIPS convention
  - Agreed upon "contract" or "protocol" that everyone follows
  - Specifies the correct (and expected) usage and some naming conventions
  - Established as part of the architecture
  - Used by all compilers, programs, and libraries
  - Assures compatibility

  - Caller saves the following registers if it uses them:
    - $s0-$s7 (procedural)
    - $sp, $fp, $ra

  - Caller must save anything else it uses

MIPS register names and conventions

Arguments (input to callee) and values (outputs to caller)
How to do a procedure call

- Transfer control to the callee:
  ```assembly
  jal ProcedureAddress ; jump-and-link to the procedure
  ```
  - The return address (PC+4) is stored in $ra

- Return control to the caller:
  ```assembly
  jr $ra       ; jump-return to the address in $ra
  ```
  - This is why you need to store the return address!

- Register convention for procedure calling:
  - $a0-$a3: Argument registers (4) for passing parameters
  - $v0-$v1: Value registers (2) for returning results
  - $ra: Return address for where to go when done

Procedure call examples and the stack
Example: which registers to save?

Questions:
1. What resources does the caller need to save?
2. What resources does the caller need to use?
3. What does the caller need to save?
4. What does the caller need to use?

Callee needs to save anything not in the registers.

Example: saving to the stack

Questions:
1. What resources does the callee need to save?
2. What resources does the callee need to use?
3. What does the callee need to save?
4. What does the callee need to use?

Caller needs to save anything not in the registers.

Saving registers (on the stack)

- The stack is a part of memory for storing temporary data.
- The Stack Pointer (kept in $sp) points to the end of the stack in memory. In MIPS the stack grows down.
- Procedures move the stack pointer when they store data on the stack.
- Each procedure returns the stack to the state it was before it was called.

- Gives procedures a secure place to store data that does not fit in registers. (e.g., saved registers)
- Each procedure manages its own stack space so they don’t interfere.
- Works great as long as you return the stack to the way it was before.

Stack before procedure call

Stack during procedure call

Stack after procedure call
Example: saving to the stack

Caller uses $t0, $s0, $s1.
Set up the arguments
add $a0, $t0, 2
add $a1, $s0, $zero
add $a2, $s1, $t0
add $a3, $t0, 3
addi $sp, $sp, -4
Set up stack
sw $t0, 0($sp)
Save $t0
jal leaf_example
Call the leaf_example procedure
lw $t0, 0($sp)
Restore $t0 from the stack
addi $sp, $sp, 4
Adjust stack
add $t2, $v0, $zero
Move result
Caller
Callee
leaf_example:
Calculates $f=(g+h)-(i+j)
g, h, i, and j are in $a0, $a1, $a2, $a3
addi $sp, $sp, -4
Adjust stack
sw $s0, 0($sp)
Save $s0 for the caller
add $t0, $a0, $a1
$g = $a0
add $t1, $a2, $a3
$i = $a2, $j = $a3
sub $s0, $t0, $t1
Return $f to the result register
lw $s0, 0($sp)
Restore $s0 for the caller
addi $sp, $sp, 4
Adjust stack
jr $ra
Jump to the calling routine

Nested calls

The stack grows and shrinks as procedure calls add (push) data when they are called and remove (pop) data when they return.

Summary: procedure calls

- Procedures need to:
  - Not corrupt caller resources
  - Return to the right place when done
- To accomplish this we:
  - Have conventions for who saves registers
  - Save them on the stack
  - Use jal and jr $ra to enter and exit procedures
- As long as everyone follows the convention we get interoperability

- Some machines provide stacks as part of the architecture (VAX, JVM)
- Others implement them in software

Some machines provide stacks as part of the architecture (VAX, JVM)
Others implement them in software

20/09/2012

© 2012 David Black-Schaffer
Other ISAs

- We’ve looked at MIPS in detail, but there are a lot of other ISAs:
  - x86 (Intel/AMD)
  - ARM (Apple)
  - JVM (Java)
  - PPC (IBM, Motorola)
  - SPARC (Oracle, Fujitsu)
  - PTX (Nvidia)
  - etc.

- Let’s take a look at a few issues:
  - Machine types
  - ISA classes
  - Addressing modes
  - Instruction width
  - CISC vs. RISC
Basic machine types

- **Memory-to-Memory** machines:
  - Instructions can directly manipulate memory
  - Example: \( \text{Mem}[0] + \text{Mem}[1] \times \text{Mem}[2] \)
- **Architectural registers**:
  - Hold temporary variables
  - Far faster than memory ⇒ faster programs
  - Fewer addresses in code ⇒ smaller programs
- **But it's never that simple…**
  - x86 has a few registers and supports memory operations
  - ARM has many addressing modes that complicate register operations
  - When you run out of registers you have to \textit{“split”} data to memory

Basic ISA classes

- **Accumulator** (1 register)
  - 1 address
  - \( \text{acc} \leftarrow \text{acc} + \text{mem}[A] \)
- **General purpose register file** (load/store)
  - 3 addresses
  - \( \text{Ra} \leftarrow \text{Ra} + \text{Rb} \)
  - \( \text{Ra} \leftarrow \text{Mem}[Rb] \)
- **General purpose register file** (Register-Memory)
  - 2 address
  - \( \text{Ra} \leftarrow \text{Ra} + \text{mem}[Rb] \)
- **Stack** (not a register file but an operand stack)
  - 0 address
  - \( \text{tos} \leftarrow \text{tos} + 1 \)
  - \( \text{tos} = \text{top of stack} \)

Comparison:

- Bytes per instruction? Number of instructions?
- Cycles per instruction?

Comparing number of instructions

<table>
<thead>
<tr>
<th>Code for ( C = A + B )</th>
</tr>
</thead>
</table>
| \begin{align*}
  \text{Stack} & : \text{Push A, Push B, ADD, Pop C} \\
  \text{Accumulator} & : \text{ADD} \\
  \text{Register} & : \text{ADD} \\
  \text{Register} & : \text{Mem} \end{align*} |

More simple instructions: fewer overheads.

Lots of single source data.
Addressing modes (not all are in MIPS)

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>add R4, R3</td>
<td>R4 ← R4 + R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>add R4, 23</td>
<td>R4 ← R4 + 23</td>
</tr>
<tr>
<td>Displacement</td>
<td>add R4, 100(R1)</td>
<td>R4 ← R4 + Mem[100 + R1]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>add R4, (R1)</td>
<td>R4 ← Mem[R1]</td>
</tr>
<tr>
<td>Indexed/Signed</td>
<td>add R4, (R1+R2)</td>
<td>R4 ← Mem[R1 + R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>add R4, (R1)</td>
<td>R4 ← Mem[R1]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>add R4, 100(R1)</td>
<td>R4 ← Mem[100 + R1]</td>
</tr>
<tr>
<td>Sub-increment</td>
<td>add R4, R1</td>
<td>R4 ← Mem[R1]</td>
</tr>
<tr>
<td>Sub-decrement</td>
<td>add R4, (R1)</td>
<td>R4 ← Mem[R1]</td>
</tr>
<tr>
<td>Indirect</td>
<td>add R1, (1001)</td>
<td>R1 ← Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>add R1, @ (R3)</td>
<td>R1 ← Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>add R1, (R2)+</td>
<td>R1 ← Mem[R2]; R2 ← R2 + d</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>add R1, -(R2)</td>
<td>R2 ← R2 - d; R1 ← Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>add R1, 100(R2)</td>
<td>R1 ← Mem[100 + R2 + R3 * d]</td>
</tr>
</tbody>
</table>

Question: Why auto-increment/decrement? Why scaled?

Answer: Helpful for walking through arrays.

Instruction widths (number of bits)

- Variable width
  - Different widths for different instructions
  - x86: 2-6 bytes for add, 2-4 bytes for load
  - Better for generating compact code
  - Hard for hardware to know where instructions start/stop

- Fixed width
  - Same width for every instruction
  - MIPS: 4 bytes for add, 4 bytes for load
  - Larger code size
  - Easy for hardware to decode

- Multiple widths
  - ARM and MIPS support both 32-bit and 16-bit instructions
  - 16-bit instructions are limited, but can reduce code size

General purpose register machines dominate

- Literally all machines use general purpose registers

Advantages
- Faster than memory
- Can hold temporary variables
- Easier for compilers to use
- Improved code density
- Fewer bits to select a register than a memory address

But we just talked about how x86 was a memory-register architecture...what’s going on?
The truth about ISAs

The ISA lies, but you can trust it

- The ISA presents a simple view of the processor
  - Atomic – instructions execute one at a time
  - Sequential – instructions execute in order
  - Flat memory – can access any location easily

- Convert x86 into MIPS-like instructions
- Schedule instructions to run out of order
- Execute multiple instructions at the same time
- Make memory look faster than it really is (most of the time)
- Turn the power off to save energy.

CISC vs. RISC

- "Simple" computations are not always simple
  - Often requires a sequence of more primitive instructions
  - E.g., Mem[R1] = Mem[R2] + R3

- Architectures that provide complex instructions are Complex Instruction Set Computing = CISC
  - PRO: assembly programs are easier to write, denser code
  - CON: hardware gets really, really complicated by rarely-used instructions. Compilers are hard to write.

- Architectures that provide only primitive instructions are Reduced Instruction Set Computing = RISC
  - CON: compiler generates lots of instructions for even simple code
  - PRO: hardware and compiler are easier to design and optimize

Everything is RISC inside today to make the hardware simpler
**Summary: ISAs**

- **Architecture** = what's visible to the program about the machine
  - Not everything in the implementation is "visible"
  - The implementation may not follow the architecture
  - The invisible stuff is the "microarchitecture" and it's very messy, but very fun (huge engineering challenges; lots of money)

- A big piece of the ISA is the **assembly language** structure
  - Primitive instructions (appear to) execute sequentially and atomically
  - Formats, computations, addressing modes, etc.
    - CISC: lots of complicated instructions
    - RISC: a few basic instructions
    - All recent machines are RISC, but x86 is still CISC (although they do RISC tricks on the inside)