Overview of the MIPS Architecture: Part I

CS 161: Lecture 0

1/24/17
Looking Behind the Curtain of Software

• The OS sits between hardware and user-level software, providing:
  • Isolation (e.g., to give each process a separate memory region)
  • Fairness (e.g., via CPU scheduling)
  • Higher-level abstractions for low-level resources like IO devices

• To really understand how software works, you have to understand how the hardware works!
  • Despite OS abstractions, low-level hardware behavior is often still visible to user-level applications
    • Ex: Disk thrashing
Processors: From the View of a Terrible Programmer

Source code

Compilation

add t0, t1, t2
lw t3, 16(t0)
slt t0, t1, 0x6eb21

Machine instructions

A HARDWARE MAGIC OCCURS

Letter “m”

Drawing of bird

The Weekend

ANSWERS
Processors: From the View of a Mediocre Programmer

- Program instructions live in RAM
- PC register points to the memory address of the instruction to fetch and execute next
- Arithmetic logic unit (ALU) performs operations on registers, writes new values to registers or memory, generates outputs which determine whether to branches should be taken
- Some instructions cause devices to perform actions
Processors: From the View of a Mediocre Programmer

- Registers versus RAM
  - Registers are orders of magnitude faster for ALU to access (0.3ns versus 120ns)
  - RAM is orders of magnitude larger (a few dozen 32-bit or 64-bit registers versus GBs of RAM)
Instruction Set Architectures (ISAs)

- ISA defines the interface which hardware presents to software
  - A compiler translates high-level source code (e.g., C++, Go) to the ISA for a target processor
  - The processor directly executes ISA instructions

- Example ISAs:
  - MIPS (mostly the focus of CS 161)
  - ARM (popular on mobile devices)
  - x86 (popular on desktops and laptops; known to cause sadness among programmers and hardware developers)
Instruction Set Architectures (ISAs)

• Three basic types of instructions
  • Arithmetic/bitwise logic (ex: addition, left-shift, bitwise negation, xor)
  • Data transfers to/from/between registers and memory
• Control flow
  • Unconditionally jump to an address in memory
  • Jump to an address if a register has a value of 0
  • Invoke a function
  • Return from a function
RISC vs CISC: ISA Wars

- CISC (Complex Instruction Set Computer): ISA has a large number of complex instructions
  - "Complex": a single instruction can do many things
  - Instructions are often variable-size to minimize RAM usage
  - CISC instructions make life easier for compiler writers, but much more difficult for hardware designers—complex instructions are hard to implement and make fast

- X86 is the classic CISC ISA

  `mov` instruction: Operands can both be registers, or one register/one memory location

  //Copy `%eax` register val to `%ebx`
  `mov %eax, %ebx`

  //Copy `*(%esp+4)` to `%ebx`
  `mov 4(%esp), %ebx`

  //Copy `%ebx` register val to `*(%esp+4)`
  `mov %ebx, 4(%esp)`
RISC vs CISC: ISA Wars

- CISC (Complex Instruction Set Computer): ISA has a large number of complex instructions
  - “Complex”: a single instruction can do many things
  - Instructions are often variable-size to minimize RAM usage
  - CISC instructions make life easier for compiler writers, but much more difficult for hardware designers—complex instructions are hard to implement and make fast
- X86 is the classic CISC ISA

```c
if(cpu_direction_flag == 0){
    *(%edi++) = *(esi++);
} else{
    *(%edi--) = *(esi--);
}
```

//movsd: Copy 4 bytes from one string ptr to another
//%edi: Destination pointer
//%esi: Source pointer
A single hardware instruction has to do:
- a branch
- a memory read
- a memory write
- two register increments or decrements
That’s a lot!

```c
if(cpu_direction_flag == 0){
    *(edi++) = *(esi++);
} else{
    *(edi--) = *(esi--);
}
```
RISC vs CISC: ISA Wars

• RISC (Reduced Instruction Set Computer): ISA w/smaller number of simple instructions
  • RISC hardware only needs to do a few, simple things well—thus, RISC ISAs make it easier to design fast, power-efficient hardware
  • RISC ISAs usually have fixed-sized instructions and a load/store architecture
  • Ex: MIPS, ARM
    //On MIPS, operands for mov instr
    //can only be registers!
    mov a0, a1 //Copy a1 register val to a0

    //In fact, mov is a pseudoinstruction
    //that isn’t in the ISA! Assembler
    //translates the above to:
    addi a0, a1, 0 //a0 = a1 + 0

RAM is cheap, and RISC makes it easier to design fast CPUs, so who cares if compilers have to work a little harder to translate programs?
MIPS R3000 ISA†

• MIPS R3000 is a 32-bit architecture
  • Registers are 32-bits wide
  • Arithmetic logical unit (ALU) accepts 32-bit inputs, generates 32-bit outputs
  • All instruction types are 32-bits long

• MIPS R3000 has:
  • 32 general-purpose registers (for use by integer operations like subtraction, address calculation, etc)
  • 32 floating point registers (for use by floating point addition, multiplication, etc) <--Not supported on sys161
  • A few special-purpose registers (e.g., the program counter pc which represents the currently-executing instruction)

†As represented by the sys161 hardware emulator. For more details on the emulator, see here:
http://os161.eecs.harvard.edu/documentation/sys161/mips.html
http://os161.eecs.harvard.edu/documentation/sys161/system.html
# MIPS R3000: Registers

<table>
<thead>
<tr>
<th>Assembler Name</th>
<th>Reg. Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>Constant 0 value</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>Assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>$2-$3</td>
<td>Procedure return values or expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>$4-$7</td>
<td>Arguments/parameters</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>$8-$15</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>$16-$23</td>
<td>Saved Temporaries</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>$24-$25</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>$26-$27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>Global Pointer (Global and static variables/data)</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>Return address for current procedure</td>
</tr>
</tbody>
</table>
MIPS R3000: A Load/Store Architecture

• With the exception of load and store instructions, all other instructions require register or constant ("immediate") operands
  • Load: Read a value from a memory address into a register
  • Store: Write a value from a register into a memory location
• So, to manipulate memory values, a MIPS program must
  • Load the memory values into registers
  • Use register-manipulating instructions on the values
  • Store those values in memory
• Load/store architectures are easier to implement in hardware
  • Don’t have to worry about how each instruction will interact with complicated memory hardware!
MIPS R3000 ISA

- MIPS defines three basic instruction formats (all 32 bits wide)

<table>
<thead>
<tr>
<th>R-type</th>
<th>opcode (6)</th>
<th>srcReg0 (5)</th>
<th>srcReg1 (5)</th>
<th>dstReg1 (5)</th>
<th>shiftAmt (5)</th>
<th>func (6)</th>
</tr>
</thead>
</table>

Register indices

Used by shift instructions to indicate shift amount

Example

```
000000 00010 00101 10001 00000 100000
```

add $17, $2, $5

Determine operation to perform
MIPS R3000 ISA

- MIPS defines three basic instruction formats (all 32 bits wide)

<table>
<thead>
<tr>
<th>I-type</th>
<th>opcode (6)</th>
<th>srcReg0 (5)</th>
<th>src/dst (5)</th>
<th>immediate (16)</th>
</tr>
</thead>
</table>

Example

```
001000 00010 10001 0000000000000001
```

addi $17, $2, 1

Example

```
100011 00010 10001 00000000000000100
```

lw $17, 4($2)
MIPS R3000 ISA

- MIPS defines three basic instruction formats (all 32 bits wide)

<table>
<thead>
<tr>
<th>J-type</th>
<th>opcode (6)</th>
<th>Jump address (26)</th>
</tr>
</thead>
</table>

Example:

- To form the full 32-bit jump target:
  - Pad the end with two 0 bits (since instruction addresses must be 32-bit aligned)
  - Pad the beginning with the first four bits of the PC

```
000010 00000000000000000000000001000000
```

```
j 64
```
How Do We Build A Processor To Execute MIPS Instructions?
Pipelining: The Need for Speed

• Vin Diesel needs more cars because VIN DIESEL
• A single car must be constructed in stages
  • Build the floorboard
  • Build the frame
  • Attach floorboard to frame
  • Install engine
  • I DON’T KNOW HOW CARS ARE MADE BUT YOU GET THE POINT

• Q: How do you design the car factory?
Factory Design #1

- Suppose that building a car requires three tasks that must be performed in serial (i.e., the tasks cannot be overlapped)
- Further suppose that each task takes the same amount of time
- We can design a single, complex robot that can perform all of the tasks

- The factory will build one car every three time units
Factory Design #2

• Alternatively, we can build three simple robots, each of which performs one task.

• The robots can work in parallel, performing their tasks on different cars simultaneously.

• Once the factory ramps up, it can make one car every time unit!

The factory has ramped up: the pipeline is now full!
Pipelining a MIPS Processor

• Executing an instruction requires five steps to be performed
  • Fetch: Pull the instruction from RAM into the processor
  • Decode: Determine the type of the instruction and extract the operands (e.g., the register indices, the immediate value, etc.)
  • Execute: If necessary, perform the arithmetic operation that is associated with the instruction
  • Memory: If necessary, read or write a value from/to RAM
  • Writeback: If necessary, update a register with the result of an arithmetic operation or a RAM read

• Place each step in its own hardware stage
  • This increases the number of instructions finished per time unit, as in the car example

• A processor’s clock frequency is the rate at which its pipeline completes instructions
Processors: From the View of a Master Programmer

Next sequential PC

4

Current instruction register

PC

Addr

Memory

Instr

Id/Ex

Ex/Mem

Mem/WB

ALU

ID/EX

Ex/Mem

MEM/WB

Current instruction register

Read reg0 idx

Read reg1 idx

Write reg idx

Write reg data

Read data 0

Read data 1

Registers

Sign extend 16-bit imm to 32-bit

Write reg idx

Sign-extended imm

Reg0==0?

Opcode

RdData

Memory

Addr

WrData
Fetch:
add t0, t1, t2

//PC=addr of add instr
//Fetch add instr and
//increment PC by 4
Decode: \texttt{add t0, t1, t2}

// Read reg0=t1
// Write reg=t0
// opcode=add

// Read reg0=0
// Write reg=0

Current instruction register:
- PC
- Addr
- Memory
- Instr
- Next sequential PC
- 4
- +

Registers:
- Read reg0 idx
- Read reg1 idx
- Write reg idx
- Write reg data

Memory:
- Addr
- WrData
- RdData

ALU:
- Opcode
- \texttt{?}
- Sign-extended imm
- \texttt{?}
-\texttt{16-bit imm to 32-bit}
- \texttt{?}
- Write reg idx

ID/EX:
- \texttt{?}
- \texttt{?}

EX/MEM:
- \texttt{?}
- \texttt{?}

MEM/WB:
- \texttt{?}
Execute: add t0, t1, t2

// Calculate read data 0 + read data 1

Current instruction register

Next sequential PC

4

Current instruction register

Addr

Memory

Instr

PC

ID/EX

ALU

EX/MEM

MEM/WB

Read reg0 idx

Read reg1 idx

Write reg idx

Write reg data

Read data 0

Read data 1

Registers

Sign-extended imm

Write reg idx

Sign extend 16-bit imm to 32-bit

Write reg data

Opcode

RdData

Memory

Addr

WrData
Next sequential PC

4

Current instruction register

PC
Addr
Memory
Instr

Sign-extended imm

Read reg0 idx
Read reg1 idx
Write reg idx
Write reg data

Read data 0
Read data 1

Registers

Sign extend 16-bit imm to 32-bit

Write reg idx

ID/EX

 Opcode

EX/MEM

MEM/WB

Memory:add t0, t1, t2
//Nothing to do here; all operands are registers
Writeback: add t0, t1, t2

// Update t2 with the result of the add
Fetch: lw t0, 16(t1)

//PC=addr of lw instr
//Fetch lw instr and
//increment PC by 4
Decode: lw t0, 16(t1)

// Read reg0=t1, Write reg=t0
// imm=16
// opcode=lw
Execute: `lw t0, 16(t1)`  
//Calculate the mem addr
// (value of t1) + 16
Memory: `lw t0, 16(t1)`  //Ask the memory hardware to fetch data at addr

Current instruction register

Next sequential PC

4

PC

Addr Memory Instr

Current instruction register

Write reg idx

Write reg data

Read data 0

Read data 1

Registers

Sign extended 16-bit imm to 32-bit

Write reg idx

Opcode

ALU

Memory

Addr WrData

RdData

MEM/WB
Writeback: \texttt{lw \ t0, 16(t1)}

//Update t0 with the //value from memory