CIS 314
MIPS and x86

Prof. Michel A. Kinsky
Computer Organization v1

- Our second view of computer organization

![Diagram of computer organization](image)

- Processor
- Data transfer
- Memory

- Instruction set (inst<25:21>, inst<20:16>, inst<15:11>, inst<5:0>)
- ALU
- Control
- RegWrite
- Data transfer (Data)
- Address (Address)

Address:
- 3: 100
- 2: 17
- 1: 114
- 0: 2
RISC: MIPS

- MIPS: Microprocessor without Interlocked Pipeline Stages

- There are 3 types of instruction in MIPS
  
  1. R-Type

- op: Basic operation of the instruction, typically called the opcode
- rs: The first register source operand
- rt: The second register source operand
RISC: MIPS

• MIPS: Microprocessor without Interlocked Pipeline Stages

• There 3 types of instruction in MIPS

1. R-Type

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• rd: The register destination operand, it gets the result of the operation

• shamt: Shift amount (0 if not shift instruction)
RISC: MIPS

- MIPS: Microprocessor without Interlocked Pipeline Stages

- There are 3 types of instruction in MIPS

1. R-Type

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- funct: Function. This field selects the specific variant of the operation in the op field, and is sometimes called the function code

- For R-Type op = 0
## RISC: MIPS

- MIPS: Microprocessor without Interlocked Pipeline Stages
- There are 3 types of instruction in MIPS
  1. **R-Type**
     - `add $t0, $s1, $s2`
• MIPS: Microprocessor without Interlocked Pipeline Stages

• There 3 types of instruction in MIPS

2. I-Type

  • Register immediate

    \[
    \text{lw} \ $t0, \ 8($t3)
    \]

  • Branch displacement
### RISC: MIPS

- **MIPS**: Microprocessor without Interlocked Pipeline Stages
- There are 3 types of instructions in MIPS

#### 2. I-Type

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beqz $s0, label</td>
<td>$s0 == 0</td>
<td>goto label</td>
</tr>
<tr>
<td>bnez $s0, label</td>
<td>$s0 != 0</td>
<td>goto label</td>
</tr>
<tr>
<td>bge $s0, $s1, label</td>
<td>$s0 &gt;= $s1</td>
<td>goto label</td>
</tr>
<tr>
<td>ble $s0, $s1, label</td>
<td>$s0 &lt;= $s1</td>
<td>goto label</td>
</tr>
<tr>
<td>blt $s0, $s1, label</td>
<td>$s0 &lt; $s1</td>
<td>goto label</td>
</tr>
<tr>
<td>beq $s0, $s1, label</td>
<td>$s0 == $s1</td>
<td>goto label</td>
</tr>
<tr>
<td>bgez $s0, $s1, label</td>
<td>$s0 &gt;= 0</td>
<td>goto label</td>
</tr>
</tbody>
</table>
RISC: MIPS

- MIPS: Microprocessor without Interlocked Pipeline Stages
- There 3 types of instruction in MIPS
  3. J-Type

- `jal fib`
RISC: MIPS Instruction Format

- **R-type**, 3 register operands
- **I-type**, 2 register operands and 16-bit immediate operand
- **J-type**, 26-bit immediate operand
- **Simple Decoding**
  - 4 bytes per instruction, regardless of format
  - Format and fields easy to extract in hardware
RISC: MIPS Register Conventions

- Register $r1 is reserved for use by the assembler
- Registers $r26-$r27 are reserved for the operating system

<table>
<thead>
<tr>
<th>NAME</th>
<th>Register Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$r0</td>
<td>Hardwired to the constant value 0</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>$r2 - $r3</td>
<td>Subroutine results and expression evaluation</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>$r4 - $r7</td>
<td>Arguments (parameters) to subroutines</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>$r8 - $r15</td>
<td>Temporary registers (caller saves)</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>$r16 - $r23</td>
<td>Saved registers (callee saves)</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>$r24 - $r25</td>
<td>More temporary registers (caller saves)</td>
</tr>
<tr>
<td>$gp</td>
<td>$r28</td>
<td>Global pointer (e.g. to static data area)</td>
</tr>
<tr>
<td>$sp</td>
<td>$r29</td>
<td>Stack pointer (stack grows downwards)</td>
</tr>
<tr>
<td>$fp</td>
<td>$r30</td>
<td>Frame pointer (to local variables on stack)</td>
</tr>
<tr>
<td>$ra</td>
<td>$r31</td>
<td>Return address for subroutine calls</td>
</tr>
</tbody>
</table>
RISC: MIPS

• Case study
  ‣ Assume that A is an array of 64 words and the compiler has associated registers $s1$ and $s2$ with the variables $x$ and $y$. Also assume that the starting address, or base address is contained in register $s3$. Determine the MIPS instructions associated with the following C statement:

• $x = y + A[4];$  // adds 4th element in array $A$ to $y$ and stores result in $x$
• Case study
  ‣ Assume that A is an array of 64 words and the compiler has associated registers $s1$ and $s2$ with the variables $x$ and $y$. Also assume that the starting address, or base address is contained in register $s3$.

  • $x = y + A[4]$; // adds 4th element in array A to y and stores result in x

  ‣ Solution:

  • $lw \ $t0, 16($s3) \ # \ $s3$ contains the base address of array and $16$ is the offset address of the 4th element
  • $add \ $s1, $s2, $t0 \ # \ performs \ addition$
RISC: MIPS

• Function call

1. Caller places parameters in a place where the procedure can access them
2. Transfer control to the procedure
3. Acquire storage resources required by the procedure
4. Execute the statements in the procedure
5. Called function places the result in a place where the caller can access it
6. Return control to the statement next to the procedure call
• Function call
  ‣ Argument Passing
    • Arguments to a function passed through $a0$-$a3$
    • Functions with more than 4 arguments
      ‣ First four arguments are put in $a0$-$a3$
      ‣ Remaining arguments are put on stack by the caller
  ‣ Return Values
    • Return values from a function passed through $v0$-$v1$
    • Functions with more than 2 return values
RISC: MIPS

• Function call
  ‣ Argument Passing
    • Arguments to a function passed through $a0-a3$
    • Functions with more than 4 arguments
  ‣ Return Values
    • Return values from a function passed through $v0-v1$
    • Functions with more than 2 return values
      ‣ First two return values put in $v0-v1$
      ‣ Remaining return values put on stack by the function
      ‣ The remaining return values are popped from the stack by the caller
CISC: x86

- x86 instruction set architecture (ISA)
- IA-32: Intel Architecture 32-bit (i386)
- Intel 80386 microprocessors in 1985
- Traditional Registers in X86
  - General Purpose Registers
    - AX, BX, CX, DX
CISC: x86

• x86 instruction set architecture (ISA)
• IA-32: Intel Architecture 32-bit (i386)
• Intel 80386 microprocessors in 1985
• Traditional Registers in X86
  ‣ General Purpose Registers
  ‣ Pseudo General Purpose Registers
    • Stack: SP (stack pointer), BP (base pointer)
    • Strings: SI (source index), DI (destination index)
CISC: x86

• x86 instruction set architecture (ISA)
• IA-32: Intel Architecture 32-bit (i386)
• Intel 80386 microprocessors in 1985
• Traditional Registers in X86
  ‣ General Purpose Registers
  ‣ Pseudo General Purpose Registers
  ‣ Special Purpose Registers
    • IP (instruction pointer) and EFLAGS
CISC: x86

• EFLAGS
  ‣ CF – Carry Flag
    • Set by arithmetic instructions that generate a carry or borrow
  ‣ ZF – Zero Flag
    • Set if the result of the arithmetic operation is zero
  ‣ SF – Sign Flag
    • On signed operands it indicates if the result is positive or negative
  ‣ OF – Overflow Flag
    • Set this flag to indicate that the result was at least 1 bit too large to fit in the destination
CISC: x86

• The x86-64 architecture is a 64-bit superset of the 32-bit x86 instruction set architecture
  ‣ x86-64 was designed by AMD who named it AMD64
  ‣ It has been cloned by Intel under the name Intel 64
• All instructions in the x86 instruction set can be executed by x86-64 CPUs
• GPRs are extended to 16
• x86-64 should not be confused with the Intel Itanium architecture known as IA-64
CISC: x86

- **GPR usage**
  - AX ← Accumulator (arithmetic)
  - BX ← Base (memory addressing)
  - CX ← Counter (loops)
  - DX ← Data (data manipulation)

- **Modern extensions**
  - “E” prefix for 32 bit variants → EAX, ESP
  - “R” prefix for 64 bit variants → RAX, RSP
CISC: x86

- Registers usage conventions
  - Caller-save registers - eax, edx, ecx
    - The caller is responsible for saving the value before a call to a subroutine, and restoring the value after the call returns
  - Callee-save registers - ebp, ebx, esi, edi
    - If the callee needs to use more registers than are saved by the caller, the callee is responsible for making sure the values are stored/restored
#include <stdio.h>

int main(){
    printf("Hello World!\n");
    return 0x1234;
}

08048374 <main>:

08048374:    8d 4c 24 04           lea    0x4(%esp),%ecx
08048378:    83 e4 f0            and    $0xffffffff,%esp
0804837b:    ff 71 fc           pushl  -0x4(%ecx)
0804837e:    55                  push   %ebp
0804837f:    89 e5            mov     %esp,%ebp
08048381:    51                  push   %ecx
08048382:    83 ec 04          sub     $0x4,%esp
08048385:    c7 04 24 60 84 04 08 movl     $0x8048460,(%esp)
0804838c:    e8 43 ff ff ff     call    80482d4 <puts@plt>
08048391:    b8 2a 00 00 00     mov      $0x1234,%eax
08048396:    83 c4 04          add      $0x4,%esp
08048399:    59                  pop       %ecx
0804839a:    5d                  pop       %ebp
...

CISC: x86

- Too complex so let us consider Y86
  - The Y86 is a “toy” machine that is similar to the x86
  - It is a gentler introduction to assembly level programming than the full x86.
  - Few instructions as opposed to thousands for the x86
    - Fewer addressing modes
    - Simpler system state
    - Absolute addressing
  - Everything you learn about the Y86 will apply to the x86
CISC: x86 – Y86

• Program Registers
  ‣ 8 same as IA32 with each 32 bits
• Program Counter
  ‣ Indicates address of instruction

Program registers

<table>
<thead>
<tr>
<th>%eax</th>
<th>%esi</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>%edi</td>
</tr>
<tr>
<td>%edx</td>
<td>%esp</td>
</tr>
<tr>
<td>%ebx</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

Condition codes

<table>
<thead>
<tr>
<th>OF</th>
<th>ZF</th>
<th>SF</th>
</tr>
</thead>
</table>

Memory

PC
## CISC: x86 – Y86

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovl rA, rB</td>
<td>2</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovl V, rB</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>rB</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>mrmovl D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>OPl rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jxx Dest</td>
<td>7</td>
<td>fn</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### CISC: x86 - 86

<table>
<thead>
<tr>
<th>IA32</th>
<th>Y86</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0xabcd, %edx</td>
<td>irmovl $0xabcd, %edx</td>
<td>30 82 cd ab 00 00</td>
</tr>
<tr>
<td>movl %esp, %ebx</td>
<td>rrmovl %esp, %ebx</td>
<td>20 43</td>
</tr>
<tr>
<td>movl -12(%ebp),%ecx</td>
<td>mrmovl -12(%ebp),%ecx</td>
<td>50 15 f4 ff ff ff</td>
</tr>
<tr>
<td>movl %esi,0x41c(%esp)</td>
<td>rmmovl %esi,0x41c(%esp)</td>
<td>40 64 1c 04 00 00</td>
</tr>
</tbody>
</table>

| Instruction  |  |  |
|--------------|  |  |
| movl $0xabcd, (%eax)  | — |  |
| movl %eax, 12(%eax,%edx) | — |  |
| movl (%ebp,%eax,4),%ecx | — |  |
CISC: x86 – Y86

- Address Computation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
CISC: x86 – Y86

• Simulator
  ‣ http://www.cs.unm.edu/~bradykey/lab4Y86Sim341.html
Next Class

• Introduction to Digital Logic