EECS 361
Computer Architecture
Lecture 4: MIPS Instruction Set Architecture
Today’s Lecture

° Quick Review of Last Lecture
° Basic ISA Decisions and Design
° Announcements
° Operations
° Instruction Sequencing
° Delayed Branch
° Procedure Calling
Quick Review of Last Lecture
Comparing Number of Instructions

Code sequence for \((C = A + B)\) for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C, R3</td>
</tr>
</tbody>
</table>

\[\text{Execution Time} = \frac{1}{\text{Performance}} = \text{Instructions} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}\]
General Purpose Registers Dominate

° 1975-2002 all machines use general purpose registers

° Advantages of registers
  • Registers are faster than memory
  • Registers compiler technology has evolved to efficiently generate code for register files
    - E.g., \((A*B) - (C*D) - (E*F)\) can do multiplies in any order vs. stack
  • Registers can hold variables
    - Memory traffic is reduced, so program is sped up (since registers are faster than memory)
  • Code density improves (since register named with fewer bits than memory location)
  • Registers imply operand locality
Operand Size Usage

- Support for these data sizes and types:
  - 8-bit, 16-bit, 32-bit integers and
  - 32-bit and 64-bit IEEE 754 floating point numbers
Typical Operations (little change since 1960)

Data Movement
- Load (from memory)
- Store (to memory)
- Memory-to-memory move
- Register-to-register move
- Input (from I/O device)
- Output (to I/O device)
- Push, pop (to/from stack)

Arithmetic
- Integer (binary + decimal) or FP
- Add, Subtract, Multiply, Divide

Shift
- Shift left/right, rotate left/right

Logical
- Not, and, or, set, clear

Control (Jump/Branch)
- Unconditional, conditional

Subroutine Linkage
- Call, return

Interrupt
- Trap, return

Synchronization
- Test & set (atomic r-m-w)

String
- Search, translate

Graphics (MMX)
- Parallel subword ops (4 16-bit add)
Addressing Modes

- Addressing modes specify a constant, a register, or a location in memory
  - **Register**   \( \text{add } r1, r2 \quad r1 \leftarrow r1+r2 \)
  - **Immediate** \( \text{add } r1, \#5 \quad r1 \leftarrow r1+5 \)
  - **Direct** \( \text{add } r1, (0x200) \quad r1 \leftarrow r1+\text{M}[0x200] \)
  - **Register indirect** \( \text{add } r1, (r2) \quad r1 \leftarrow r1+\text{M}[r2] \)
  - **Displacement** \( \text{add } r1, 100(r2) \quad r1 \leftarrow r1+\text{M}[r2+100] \)
  - **Indexed** \( \text{add } r1, (r2+r3) \quad r1 \leftarrow r1+\text{M}[r2+r3] \)
  - **Scaled** \( \text{add } r1, (r2+r3*4) \quad r1 \leftarrow r1+\text{M}[r2+r3*4] \)
  - **Memory indirect** \( \text{add } r1, @(r2) \quad r1 \leftarrow r1+\text{M}[\text{M}[r2]] \)
  - **Auto-increment** \( \text{add } r1, (r2)+ \quad r1 \leftarrow r1+\text{M}[r2], \quad r2++ \)
  - **Auto-decrement** \( \text{add } r1, -(r2) \quad r2-- , \quad r1 \leftarrow r1+\text{M}[r2] \)

- Complicated modes reduce instruction count at the cost of complex implementations
Instruction Sequencing

° The next instruction to be executed is typically implied
  • Instructions execute sequentially
  • Instruction sequencing increments a Program Counter

° Sequencing flow is disrupted conditionally and unconditionally
  • The ability of computers to test results and conditionally
    instructions is one of the reasons computers have become so
    useful

Branch instructions are ~20% of all instructions executed
Instruction Set Design Metrics

° Static Metrics
  • How many bytes does the program occupy in memory?

° Dynamic Metrics
  • How many instructions are executed?
  • How many bytes does the processor fetch to execute the program?
  • How many clocks are required per instruction?
  • How "lean" a clock is practical?

\[
\text{Execution Time} = \frac{1}{\text{Performance}} = \text{Instructions} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]
MIPS R2000 / R3000 Registers

- Programmable storage
MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide
MIPS R2000 / R3000 Operation Overview

° Arithmetic logical
° Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
° AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, LUI
° SLL, SRL, SRA, SLLV, SRLV, SRAV

° Memory Access
° LB, LBU, LH, LHU, LW, LWL, LWR
° SB, SH, SW, SWL, SWR
Multiply / Divide

° Start multiply, divide
  • MULT rs, rt
  • MULTU rs, rt
  • DIV rs, rt
  • DIVU rs, rt

° Move result from multiply, divide
  • MFHI rd
  • MFLO rd

° Move to HI or LO
  • MTHI rd
  • MTLO rd
Multiply / Divide

- Start multiply, divide
  - MULT rs, rt Move to HI or LO
  - MTHI rd
  - MTLO rd

- Why not Third field for destination?
  (Hint: how many clock cycles for multiply or divide vs. add?)
## MIPS arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>
MIPS logical instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ⊕ $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~($2</td>
<td>$3)</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1,$2,10</td>
<td>$1 = ~$2 &amp;~10</td>
<td>Logical XOR reg, constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slvv $1,$2,$3</td>
<td>$1 = $2 &lt;&lt; $3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srlv $1,$2,$3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>srav $1,$2,$3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>
## MIPS data transfer instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW 500(R4), R3</td>
<td>Store word</td>
</tr>
<tr>
<td>SH 502(R2), R3</td>
<td>Store half</td>
</tr>
<tr>
<td>SB 41(R3), R2</td>
<td>Store byte</td>
</tr>
<tr>
<td>LW R1, 30(R2)</td>
<td>Load word</td>
</tr>
<tr>
<td>LH R1, 40(R3)</td>
<td>Load halfword</td>
</tr>
<tr>
<td>LHU R1, 40(R3)</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>LB R1, 40(R3)</td>
<td>Load byte</td>
</tr>
<tr>
<td>LBU R1, 40(R3)</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>LUI R1, 40</td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

![Diagram of LUI R5 instruction](image)
Methods of Testing Condition

° Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

ex: add r1, r2, r3
    bz label

° Condition Register

Ex: cmp r1, r2, r3
    bgt r1, label

° Compare and Branch

Ex: bgt r1, r2, label
Condition Codes

Setting CC as side effect can reduce the # of instructions

X:  .  vs.  .
   .
   .
   SUB r0, #1, r0
   BRP X
   SUB r0, #1, r0
   CMP r0, #0
   BRP X

But also has disadvantages:

--- not all instructions set the condition codes; which do and which do not often confusing!
   e.g., shift instruction sets the carry bit

--- dependency between the instruction that sets the CC and the one that tests it: to overlap their execution, may need to separate them with an instruction that does not change the CC

```
ifetch    read    compute    write

Old CC read    New CC computed
```

ifetch    read    compute    write
Compare and Branch

° Compare and Branch
  • BEQ rs, rt, offset if R[rs] == R[rt] then PC-relative branch
  • BNE rs, rt, offset <>0

° Compare to zero and Branch
  • BLEZ rs, offset if R[rs] <= 0 then PC-relative branch
  • BGTZ rs, offset >0
  • BLT <0
  • BGEZ >=0
  • BLTZAL rs, offset if R[rs] < 0 then branch and link (into R 31)
  • BGEZAL >=0

° Remaining set of compare and branch take two instructions

° Almost all comparisons are against zero!
## MIPS jump, branch, compare instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1,$2,100</td>
<td>if ($1 == $2) go to PC+4+100 ( \text{Equal test; PC relative branch} )</td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>if ($1! = $2) go to PC+4+100 ( \text{Not equal test; PC relative} )</td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3</td>
<td>if ( $2 &lt; $3 ) $1=1; else $1=0 ( \text{Compare less than; 2's comp.} )</td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1,$2,100</td>
<td>if ( $2 &lt; 100 ) $1=1; else $1=0 ( \text{Compare &lt; constant; 2's comp.} )</td>
</tr>
<tr>
<td>set less than uns.</td>
<td>sltu $1,$2,$3</td>
<td>if ( $2 &lt; $3 ) $1=1; else $1=0 ( \text{Compare less than; natural numbers} )</td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>sltiu $1,$2,100</td>
<td>if ( $2 &lt; 100 ) $1=1; else $1=0 ( \text{Compare &lt; constant; natural numbers} )</td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000 ( \text{Jump to target address} )</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31 ( \text{For switch, procedure return} )</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000 ( \text{For procedure call} )</td>
</tr>
</tbody>
</table>
Signed vs. Unsigned Comparison

**Value?**
2’s comp  Unsigned?

R1 = 0...00 0000 0000 0000 0001 two
R2 = 0...00 0000 0000 0000 0010 two
R3 = 1...11 1111 1111 1111 1111 two

° After executing these instructions:
  
  `slt r4, r2, r1 ; if (r2 < r1) r4=1; else r4=0`
  
  `slt r5, r3, r1 ; if (r3 < r1) r5=1; else r5=0`
  
  `sltu r6, r2, r1 ; if (r2 < r1) r6=1; else r6=0`
  
  `sltu r7, r3, r1 ; if (r3 < r1) r7=1; else r7=0`

° What are values of registers r4 - r7? Why?

r4 =     ; r5 =     ; r6 =     ; r7 =     ;
Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:

Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)
Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:

How is empty stack represented?

Little --> Big/Last Full

POP: Read from Mem(SP)
Decrement SP

PUSH: Increment SP
Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP
Read from Mem(SP)

PUSH: Write to Mem(SP)
Increment SP
Many variations on stacks possible (up/down, last pushed / next)

Block structured languages contain link to lexically enclosing frame

Compilers normally keep scalar variables in registers, not memory!
### MIPS: Software conventions for Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0 expression evaluation &amp;</td>
</tr>
<tr>
<td>3</td>
<td>v1 function results</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0 temporary: caller saves</td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td>...</td>
<td>(callee can clobber)</td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8 temporary (cont’d)</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0 reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp Pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>sp Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra Return Address (HW)</td>
</tr>
</tbody>
</table>

Plus a 3-deep stack of mode bits.
Example in C: swap

```c
swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

° Assume swap is called as a procedure
° Assume temp is register $15$; arguments in $a1$, $a2$; $16$ is scratch reg:
° Write MIPS code
swap: MIPS

```assembly
addiu $sp,$sp, -4 ; create space on stack
sw $16, 4($sp) ; callee saved register put onto stack
sll $t2, $a2,2 ; multiply k by 4
addu $t2, $a1,$t2 ; address of v[k]
lw $15, 0($t2) ; load v[k]
lw $16, 4($t2) ; load v[k+1]
sw $16, 0($t2) ; store v[k+1] into v[k]
sw $15, 4($t2) ; store old value of v[k] into v[k+1]
lw $16, 4($sp) ; callee saved register restored from stack
addiu $sp,$sp, 4 ; restore top of stack
jr $31 ; return to place that called swap
```
Delayed Branches

```
li    r3, #7
sub   r4, r4, 1
bz    r4, LL
addi  r5, r3, 1
subi  r6, r6, 2
LL:  slt  r1, r3, r5
```

° In the “Raw” MIPS the instruction after the branch is executed even when the branch is taken?
  • This is hidden by the assembler for the MIPS “virtual machine”
  • allows the compiler to better utilize the instruction pipeline (???)
By the end of Branch instruction, the CPU knows whether or not the branch will take place.

However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

Why not execute it?
Filling Delayed Branches

Branch: Inst Fetch  Dcd & Op Fetch  Execute

execute successor even if branch taken!
Then branch target or continue

Single delay slot impacts the critical path

- Compiler can fill a single delay slot with a useful instruction 50% of the time.
  - try to move down from above jump
  - move up from target, if safe

add r3, r1, r2
sub r4, r4, 1
bz r4, LL
NOP
...
LL: add rd, ...

Is this violating the ISA abstraction?
Standard and Delayed Interpretation

```
add     rd, rs, rt   R[rd] <- R[rs] + R[rt];
PC   <- PC + 4;
beq     rs, rt, offset  if  R[rs] == R[rt] then PC <- PC + SX(offset)
else PC <- PC + 4;
sub     rd, rs, rt   ...
...
L1:    target
```

```
add     rd, rs, rt   R[rd] <- R[rs] + R[rt];
PC   <- nPC;  nPC <- nPC + 4;
beq     rs, rt, offset  if  R[rd] == R[rt] then nPC <- nPC + SX(offset)
else nPC <- nPC + 4;
PC   <- nPC
sub     rd, rs, rt   ...
...
L1:    target
```

*Delayed Loads?*
Delayed Branches (cont.)

Execution History

<table>
<thead>
<tr>
<th>instr0</th>
<th>PC</th>
<th>instr1</th>
<th>PC</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCND X</td>
<td></td>
<td>nPC</td>
<td></td>
<td>nPC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Branch Not Taken</td>
<td>Branch Taken</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>t2</td>
<td>t0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branches are the bane (or pain!) of pipelined machines
Delayed branches complicate the compiler slightly, but make pipelining easier to implement and more effective
Good strategy to move some complexity to compile time
Miscellaneous MIPS instructions

- **break**  
  A breakpoint trap occurs, transfers control to exception handler

- **syscall**  
  A system trap occurs, transfers control to exception handler

- **coprocessor instrs.**  
  Support for floating point: discussed later

- **TLB instructions**  
  Support for virtual memory: discussed later

- **restore from exception**  
  Restores previous interrupt mask & kernel/user mode bits into status register

- **load word left/right**  
  Supports misaligned word loads

- **store word left/right**  
  Supports misaligned word stores
Details of the MIPS instruction set

° Register zero always has the value zero (even if you try to write it)
° Branch and jump instructions put the return address PC+4 into the link register
° All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, ...) 
° Immediate arithmetic and logical instructions are extended as follows:
   • logical immediates are zero extended to 32 bits
   • arithmetic immediates are sign extended to 32 bits
° The data loaded by the instructions lb and lh are extended as follows:
   • lbu, lhu are zero extended
   • lb, lh are sign extended
° Overflow can occur in these arithmetic and logical instructions:
   • add, sub, addi
   • it cannot occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu
Other ISAs

° Intel 8086/88 => 80286 => 80386 => 80486 => Pentium => P6
  • 8086 few transistors to implement 16-bit microprocessor
  • tried to be somewhat compatible with 8-bit microprocessor 8080
  • successors added features which were missing from 8086 over next 15 years
  • product several different intel engineers over 10 to 15 years
  • Announced 1978

° VAX simple compilers & small code size =>
  • efficient instruction encoding
  • powerful addressing modes
  • powerful instructions
  • few registers
  • product of a single talented architect
  • Announced 1977
MIPS / GCC Calling Conventions

fact:
addiu $sp, $sp, -32
sw $ra, 20($sp)
sw $fp, 16($sp)
addiu $fp, $sp, 32
...
sw $a0, 0($fp)
...
lw $31, 20($sp)
lw $fp, 16($sp)
addiu $sp, $sp, 32
jr $31

First four arguments passed in registers.
## Machine Examples: Address & Registers

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Memory Sizes</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 8086</td>
<td>$2^{20} \times 8$ bit bytes</td>
<td>acc, index, count, quot stack, string code, stack, data segment</td>
</tr>
<tr>
<td></td>
<td>$2^{16} \times 8$ bit bytes</td>
<td>r15-- program counter</td>
</tr>
<tr>
<td></td>
<td>$16 \times 32$ bit GPRs</td>
<td>r14-- stack pointer</td>
</tr>
<tr>
<td>VAX 11</td>
<td>$2^{32} \times 8$ bit bytes</td>
<td>r13-- frame pointer</td>
</tr>
<tr>
<td></td>
<td>$16 \times 32$ bit GPRs</td>
<td>r12-- argument ptr</td>
</tr>
<tr>
<td>MC 68000</td>
<td>$2^{24} \times 8$ bit bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$8 \times 32$ bit GPRs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$7 \times 32$ bit addr reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1 \times 32$ bit SP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1 \times 32$ bit PC</td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>$2^{32} \times 8$ bit bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$32 \times 32$ bit GPRs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$32 \times 32$ bit FPRs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI, LO, PC</td>
<td></td>
</tr>
</tbody>
</table>
VAX Operations

° General Format:

\[(\text{operation}) \ (\text{datatype}) \ (2, \ 3)\]

2 or 3 explicit operands

° For example

\[\text{add} \quad (b, \ w, \ l, \ f, \ d) \ (2, \ 3)\]

Yields

\[
\begin{align*}
\text{addb2} & \quad \text{addw2} & \quad \text{addl2} & \quad \text{addf2} & \quad \text{addd2} \\
\text{addb3} & \quad \text{addw3} & \quad \text{addl3} & \quad \text{addf3} & \quad \text{addd3}
\end{align*}
\]
swap: MIPS vs. VAX

swap:
addiu $sp,$sp, –4 .word ^m<r0,r1,r2,r3> ; saves r0 to r3
sw $16, 4($sp)
sll $t2, $a2,2 movl r2, 4(ap) ; move arg v[] to reg
addu $t2, $a1,$t2 movl r1, 8(ap) ; move arg k to reg
lw $15, 0($t2) movl r3, (r2)[r1] ; get v[k]
lw $16, 4($t2) addl3 r0, #1,8(ap) ; reg gets k+1
lw $15, 4($t2) sw $16, 0($t2) movl (r2)[r1],(r2)[r0] ; v[k] = v[k+1]
sw $15, 4($t2) sw $16, 0($t2) movl (r2)[r0],r3 ; v[k+1] gets old v[k]
lw $16, 4($sp) addiu $sp,$sp, 4
jr $31 ret ; return to caller, restore r0 - r3
Details of the MIPS instruction set

- Register zero **always** has the value **zero** (even if you try to write it)
- Branch/jump **and link** put the return addr. PC+4 into the link register (R31)
- All instructions change **all 32 bits** of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates ops are zero extended to 32 bits
  - arithmetic immediates ops are sign extended to 32 bits (including addu)
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- **Overflow can occur** in these arithmetic and logical instructions:
  - add, sub, addi
  - it **cannot** occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu
Miscellaneous MIPS I instructions

- **break**: A breakpoint trap occurs, transfers control to exception handler
- **syscall**: A system trap occurs, transfers control to exception handler
- **coprocessor instrs.**: Support for floating point
- **TLB instructions**: Support for virtual memory: discussed later
- **restore from exception**: Restores previous interrupt mask & kernel/user mode bits into status register
- **load word left/right**: Supports misaligned word loads
- **store word left/right**: Supports misaligned word stores
Summary

° Use general purpose registers with a load-store architecture: **YES**

° Provide at least 16 general purpose registers plus separate floating-point registers: **31 GPR & 32 FPR**

° Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; **YES: 16 bits for immediate, displacement (disp=0 => register deferred)**

° All addressing modes apply to all data transfer instructions: **YES**

° Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size: **Fixed**

° Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: **YES**

° Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: **YES, 16b**

° Aim for a minimalist instruction set: **YES**
Summary: Salient features of MIPS R3000

• **32-bit fixed format inst** (3 formats)
• **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  • partitioned by software convention
• **3-address, reg-reg arithmetic instr.**
• **Single address mode for load/store:** base+displacement
  • no indirection
  • 16-bit immediate plus LUI
• **Simple branch conditions**
  • compare against zero or two registers for =
  • no condition codes
• **Delayed branch**
  • execute instruction after the branch (or jump) even if the branch is taken (Compiler can fill a delayed branch with useful work about 50% of the time)