Instructions

- Instructions are the “words” of a computer
- Instruction set architecture (ISA) is its vocabulary
- This defines most of the interface to the processor (not quite everything)
- Implementations can and do vary
  - MIPS R2K->R3K->R4K->R8K->R10K
Instructions cont’d

- MIPS ISA used in 427:
  - Simple, sensible, regular, widely used
- Most common: x86 (IA-32)
  - Intel Pentium/II/III/IV, AMD Athlon, etc.
- Others:
  - PowerPC (Mac, IBM servers)
  - SPARC (Sun)
  - ARM (Nokia, Ipaq, etc.)
Forecast

- Basics
- Registers and ALU ops
- Memory and load/store
- Branches and jumps
- Etc.
Basics

- C statement
  \[ f = (g + h) - (i + j) \]
- MIPS instructions
  - add t0, g, h
  - add t1, i, j
  - sub f, t0, t1
- Opcode/mnemonic, operands, source/destination
Basics

- Opcode: specifies the kind of operation (mnemonic)
- Operands: input and output data (source/destination)
- Operands t0 & t1 are temporaries
- One operation, two inputs, one output
- Multiple instructions for one C statement
Why not bigger instructions?

- Why not “f = (g + h) – (i + j)” as one instruction?
- Church’s thesis: A very primitive computer can compute anything that a fancy computer can compute – you need only logical functions, read and write memory, and data-dependent decisions
- Therefore, ISA selected for practical reasons:
  - Performance and cost, not computability
- Regularity tends to improve both
  - E.g. H/W to handle arbitrary number of operands is complex and slow and UNNECESSARY
Registers and ALU ops

- Operands must be registers, not variables
  - add R8, R17, R18
  - add R9, R19, R20
  - sub R16, R8, R9
- MIPS has 32 registers R0-R31 (R0 is always 0)
- R8 and R9 are temps, R16 is f, R17 is g, R18 is h, R19 is i and R20 is j
- MIPS also allows one constant called “immediate”
  - Later we will see immediate is restricted to 16 bits
ALU ops

- Some ALU ops:
  - add, addi (or addl), addu, addiu (immediate, unsigned)
  - sub …
  - mul, div
  - and, andi
  - or, ori
  - sll, srl
- Why registers?
  - Short name fits in instruction word: $\log_2(32) = 5$ bits
- But are register enough?
Memory and Load/Store

- Need more than 32 words of storage
- An array of locations $M[j]$ indexed by $j$
- Data movement (on words or integers)
  
  Load word for register $\leq$ memory
  (Assume $R5=1000$)
  LOAD R17, R5, #2 /* get input $g$ from address $1000+2=1002$ */
  Store word for register $\Rightarrow$ memory
  STORE R16, R5, #1 /* save output $f$ to address $1000+1=1001$ */
Memory and load/store

- Important for arrays
  \[ A[i] = A[i] + h \]
  
  # R8 is temp, R18 is h, R21 is (i x 4)
  
  # Astart is &A[0] is 0x8000
  
  LOAD R8, R21, Astart /* address=Astart+R21 */
  
  ADD R8, R18, R8
  
  STORE R8, R21, Astart

- MIPS has other load/store for bytes and halfwords
Memory and load/store

Registers

Processor

ALU

Memory

0

4004

4008

8000

8004

8008

A[0]

A[1]

A[2]

maxmem

© 2000 Mark Hill
Aside on “Endian”

- Big endian:
  - E.g. IBM, SPARC
- Little endian:
  - E.g. Intel x86
- Mode selectable
  - E.g. PowerPC, MIPS (for Windows NT)
2.8.6.1 Big-Endian Order

When configured in **big-endian order**, byte 0 is the most-significant (left-hand) byte. Figure 2-8 shows this configuration.

![Big-Endian Byte Ordering Diagram](image)

**Figure 2-8 Big-Endian Byte Ordering**

2.8.6.2 Little-Endian Order

When configured in **little-endian order**, byte 0 is always the least-significant (right-hand) byte. Figure 2-9 shows this configuration.

![Little-Endian Byte Ordering Diagram](image)

**Figure 2-9 Little-Endian Byte Ordering**
Branches and Jumps

While (i != j) {
    j = j + i;
    i = i + 1;
}

Loop: BEQ R8, R9, Exit
    ADD R9, R9, R8
    ADDI R8, R8, #1
    JMP Loop

Exit:
Branches and Jumps

# better:

```asm
BEQ R8, R9, Exit  # not !=
```

Loop: ADD R9, R9, R8

```asm
ADDI R8, R8 , #1
```

```asm
BNE R8, R9, Loop
```

Exit:

- Best to let compilers worry about such
Branches and Jumps

- What does bne do really?
  read R8, read R9, compare
  Set PC = PC + 4 or PC = Target
- To do compares other than = or !=
  E.g.
  BLT R8, R9, Target # pseudoinstruction
  Expands to:
  SLT R1, R8, R9  # R1==(R8<R9)==(R8-R9)<0
  BNE R1, R0, Target  # R0 is always 0
Branches and Jumps

- **Other MIPS branches/jumps**
  
  BEQ R8, R9, imm  # if (R8==R9) PC = PC + imm<< 2 else PC += 4;
  
  BNE ...
  
  SLT, SLE, SGT, SGE

- **With immediate, unsigned**
  
  JMP addr # PC = addr
  
  JR R12 # PC = R12
  
  JAL addr # R31 = PC + 4; PC = addr; used for ??
Nice Site for MIPS Information

http://www.geocities.com/SunsetStrip/Palladium/1303/java/mint/mint.htm#The%20instruction%20set
Layers of Software

- Notation: program; input data -> output data
  - Executable: input data -> output data
  - Loader: executable file -> executable in memory
  - Linker: object files -> executable file
  - Compiler: HLL file -> assembly file
  - Editor: editor commands -> HLL file
- Programs are manipulated as data
- How does Java differ?
MIPS Machine Language

- All instructions are 32 bits wide
- Assembly: ADD R1, R2, R3
- Machine language:

```
33222222222211111111110000000000
10987654321098765432109876543210
00000000010000110000100000010000
000000 00010 00011 00001 00000 010000
alu-rr 2 3 1 zero add/signed
```
Instruction Format

- R-format
  
  Opc  rs  rt  rd  shamt  function
  6  5  5  5  5  6

- Digression:
  
  How do you store the number 4,392,976?
  - Same as ADD R1, R2, R3

- Stored program: instructions are represented as numbers
  
  Programs can be read/written in memory like numbers
Instruction Format

- Other R-format: addu, sub, subi, etc.
- Assembly: LOAD R1, R2, #100
- Machine: 100011 00010 00001 0000000001100100
  load 2 1 100 (in binary)

- I-format
  
<table>
<thead>
<tr>
<th>Opc</th>
<th>rs</th>
<th>rt</th>
<th>address/immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

© 2000 Mark Hill
Instruction Format

- I-format also used for ALU ops with immediates
  
  \[
  \text{ADDI R1, R2, 100} \\
  \text{001000 00010 00001 0000000001100100}
  \]

- What about constants larger than 16 bits = \([-32768, 32767]\)?
  
  \[
  \text{LUI R4, 12} \quad \text{# $4 == 0000 0000 0000 1100 0000 0000 0000 0000}
  \]
  
  \[
  \text{ORI R4, R4, 15} \quad \text{# R4 == 0000 0000 0000 1100 0000 0000 0000 1111}
  \]

- All loads and stores use I-format
Instruction Format

- BEQ R1, R2, #7
  000100 00001 00010 0000 0000 0000 0111
  PC = PC + (0000 0111 << 2) # word offset

- Finally, J-format
  JMP address
  Opcode  addr
  6       26

- Addr is weird in MIPS:
  addr = 4 MSB of PC // addr // 00
# Summary: Instruction Formats

R: opcode  rs  rt  rd  shamt  function
    6  5  5  5  5  6

I: opcode  rs  rt  address/immediate
    6  5  5  16

J: opcode  addr
    6  26

- Instruction decode:
  - Read instruction bits
  - Activate control signals
Procedure Calls

- Caller
  - Save registers
  - Set up parameters
  - Call procedure
  - Get results
  - Restore registers

- Callee
  - Save more registers
  - Do some work, set up result
  - Restore registers
  - Return

- JAL is special, otherwise just software convention
Procedure Calls

- Stack is all-important
- Stack grows from larger to smaller addresses (arbitrary)
- R29 is stack pointer; points just beyond valid data
- Push R2:
  ADDI R29, R29, #-4
  STORE R2, R29, #4
- Pop R2:
  LOAD R2, R29, #4
  ADDI R29, R29, #4

- Cannot change order. Why?

© 2000 Mark Hill
Swap(int v[], int k) {
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

# R4 is v[] & R5 is k -- 1st & 2nd incoming argument
# R8, R9 & R10 are temporaries that callee can use w/o saving

swap:   ADD  R9,R5,R5       ; R9 = k+k
        ADD  R9,R9,R9       ; R9 = k*4 or shl ???
        ADD  R9,R4,R9       ; R9 = v + k*4 = &(v[k])
        LOAD  R8,R9,#0      ; R8 = temp = v[k]
        LOAD  R10,R9,#4     ; R10 = v[k+1]
        STORE R10,R9,#0     ; v[k] = v[k+1]
        STORE R8,R9,#4      ; v[k+1] = temp
        JR     R31           ; return
Addressing Modes

- There are many ways of accessing operands
- Register addressing:
  \[ \text{ADD R1, R2, R3} \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>...</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>register</td>
</tr>
</tbody>
</table>
Addressing Modes

- Base addressing (aka displacement)

LOAD R1, R2, #100 ; R2 == 400, M[500] == 42

```
op  rs  rt  address
```

- register
- +
- 100
- 400
- Effective address
- Memory
- 42

© 2000 Mark Hill
Addressing Modes

- Immediate addressing
  ADDI R1, R2, #100

| op | rs | rt | immediate |
Addressing Modes

- PC relative addressing
  
  \[ \text{BEQ } R1, R2, \#100 \; ; \; \text{if } (R1==R2) \; PC = PC + (100 \ll 2) \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
</table>

PC

\[ \text{shl} \]

Effect address

Memory
Addressing Modes

● Not found in MIPS:
  Indexed: add two registers – base + index
  Indirect: M[M[addr]] – two memory references
  Autoincrement/decrement: add operand size
  Autoupdate – found in PowerPC, PA-RISC
    ● Like displacement, but update base register
How to Choose ISA

- Minimize what?
  Instrs/prog \times \text{cycles/instr} \times \text{sec/cycle} !!!
- In 1985-1995 technology, simple modes like MIPS were great
  As technology changes, computer design options change
- If memory is limited, dense instructions are important
- For high speed, pipelining and ease of pipelining is important
## Intel x86 (IA-32) History

<table>
<thead>
<tr>
<th>Year</th>
<th>CPU</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1978</td>
<td>8086</td>
<td>16-bit with 8-bit bus from 8080; selected for IBM PC</td>
</tr>
<tr>
<td>1980</td>
<td>8087</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>1982</td>
<td>80286</td>
<td>24-bit addresses, memory-map, protection</td>
</tr>
<tr>
<td>1985</td>
<td>80386</td>
<td>32-bit registers, flat memory addressing, paging</td>
</tr>
<tr>
<td>1989</td>
<td>80486</td>
<td>Pipelining</td>
</tr>
<tr>
<td>1992</td>
<td>Pentium</td>
<td>Superscalar</td>
</tr>
<tr>
<td></td>
<td>Pro</td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>P-III</td>
<td>SSE – streaming SIMD</td>
</tr>
</tbody>
</table>

© 2000 Mark Hill
Intel 386 Registers & Memory

- Registers
  8 32b registers (but backward 16b & 8b: EAX, AX, AH, AL)
  4 special registers: stack (ESP) & frame (EBP)
  Condition codes: overflow, sign, zero, parity, carry
  Floating point uses 8-element stack

- Memory
  Flat 32b or segmented (rarely used)
  Effective address =
  \[(\text{base}_\text{reg} + (\text{index}_\text{reg} \times \text{scaling}_\text{factor}) + \text{displacement})\]
Intel 386 ISA

- Two register instructions: src1/dst, src2 reg/reg, reg/immed, reg/mem, mem/reg, mem/imm

- Examples
  mov EAX, 23 # 32b 2’s C imm 23 in EAX
  Faddp ST(7), ST # ST = ST + ST(7)
  Jle label # PC = label if sign or zero flag set
Intel 386 ISA cont’d

- Decoding nightmare
  Instructions 1 to 17 bytes
  Optional prefixes, postfixes alter semantics
  Crazy “formats”
    - E.g. register specifiers move around
  But key 32b 386 instructions not terrible
  Yet entire ISA has to correctly implemented
Current Approach

- Current technique in P-III, P-IV, Athlon
  Decode logic translates to RISC uops
  Execution units run RISC uops
  Backward compatible
  Very complex decoder
  Execution unit has simpler (manageable) control logic, data paths
- We use MIPS to keep it simple and clean
- Learn x86 in industry!
Complex Instructions

- More powerful instructions not faster
- E.g. string copy
  - Option 1: move with repeat prefix for memory-to-memory move
    - Special-purpose
  - Option 2: use loads/stores to/from registers
    - Generic instructions
- Option 2 faster on same machine!
- (but which code is denser?)
Conclusions

- **Simple and regular**
  
  Constant length instructions, fields in same place

- **Small and fast**
  
  Small number of operands in registers

- **Compromises inevitable**
  
  Pipelining should not be hindered

- **Make common case fast!**

- **Backwards compatibility!**

© 2000 Mark Hill