Concocting an Instruction Set

Nerd Chef at work.

Lab is posted. Do your prelab!
Stay tuned for the first problem set.
A General-Purpose Computer
The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one upon which nearly all modern computers is based was proposed by John von Neumann in the late 1940s. Its major components are:

Central Processing Unit (CPU): A device which fetches, interprets, and executes a specified set of operations called Instructions.

Memory: storage of N words of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.

I/O: Devices for communicating with the outside world.
Anatomy of an Instruction

- Computers execute a set of primitive operations called instructions.
- Instructions specify an operation and its operands (the necessary variables to perform the operation).
- Types of operands: immediate, source, and destination.

```
add  $t0, $t1, $t2
addi $t0, $t1, 1
```

Why the “$” on some operands?

$X$ is a convention to denote the “contents” of a temporary variable named “X”, whereas immediate operands indicate the specified value.
Meaning of an Instruction

- Operations are abbreviated into **opcodes** (1-4 letters)
- Instructions are specified with a very regular syntax
  - First an **opcode** followed by arguments
  - Usually the destination is next, then source arguments
    (This is not strictly the case, but it is generally true)
- Why this order?
- Analogy to high-level language like Java or C

```
add $t0, $t1, $t2
```

implies
```
int t0, t1, t2
t0 = t1 + t2
```

As opposed to:
```
t0 + t1 = t2
```

The instruction syntax provides operands in the same order as you would expect in a statement from a high level language.

What does that mean in “C”? Ans: Syntax Error
Being the Machine!

- Generally...
  - Instructions are retrieved sequentially from memory
  - An instruction executes to completion before the next instruction is started
  - But, there are exceptions to these rules

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t0, $t1, $t1</td>
<td>$t0: 0 12 24 48</td>
</tr>
<tr>
<td>add $t0, $t0, $t0</td>
<td>$t1: 6 42</td>
</tr>
<tr>
<td>add $t0, $t0, $t0</td>
<td>$t2: 8</td>
</tr>
<tr>
<td>sub $t1, $t0, $t1</td>
<td>$t3: 10</td>
</tr>
</tbody>
</table>

What does this program do?
Analyzing the Machine!

• Repeat the process treating the variables as unknowns or “formal variables”

• Knowing what the program does allows us to write down its specification, and give it a meaningful name

• The instruction sequence then becomes a general-purpose tool

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t0, $t1, $t1</td>
<td>$t0: \times 2x 4x 8x</td>
</tr>
<tr>
<td>add $t0, $t0, $t0</td>
<td>$t1: \times 7x</td>
</tr>
<tr>
<td>add $t0, $t0, $t0</td>
<td>$t2: y</td>
</tr>
<tr>
<td>sub $t1, $t0, $t1</td>
<td>$t3: z</td>
</tr>
</tbody>
</table>
Looping the Flow

- There are instructions that change the flow of sequential execution
- A jump instruction with opcode “j”
- The operand refers to a label of some other instruction

Instructions

- `add $t0, $t1, $t1`
- `add $t0, $t0, $t0`
- `add $t0, $t0, $t0`
- `sub $t1, $t0, $t1`

Variables

- `$t0: w` × 8
- `$t1: x` × 49
- `$t2: y` × 343
- `$t3: z` × 392

An infinite loop

```
Open Issues in our Simple Model

• WHERE in memory are INSTRUCTIONS stored?
• HOW are instructions represented?
• WHERE are VARIABLES stored?
• What are LABELs? How do they relate to where instructions are stored?
• How about more complicated data types?
  • Arrays?
  • Structures?
  • Objects?
• Where does a program start executing?
• When does it stop?
The Stored-Program Computer

• The von Neumann architecture addresses these issues as follows:
  • Instructions and Data are stored in a common memory
  • Sequential semantics: To the PROGRAMMER all instructions appear to be “completely” execute in an order, or sequentially

Key idea: Memory holds not only data, but coded instructions that make up a program.

CPU fetches and executes instructions from memory ...

  • The CPU is a H/W interpreter
  • Program IS simply DATA for this interpreter
  • Main memory: Single expandable resource pool
    - constrains both data and program size
    - don’t need to make separate decisions of how large of a program or data memory to buy
Anatomy of a von Neumann Computer

- INSTRUCTIONS coded as binary data
- PROGRAM COUNTER or PC: Address of next instruction to execute
- logic to translate instructions into control signals for data path
Instruction Set Architecture (ISA)

Encoding of instructions raises some interesting choices...

• Tradeoffs: performance, compactness, programmability

• Uniformity. Should different instructions
  • Be the same size?
  • Take the same amount of time to execute?

• Complexity. How many different instructions? What level operations?
  • Level of support for particular software operations: array indexing, procedure calls, “polynomial evaluate”, etc
    ➢ “Reduced Instruction Set Computer”
      (RISC) philosophy: simple instructions, optimized for speed

Mix of Engineering & Art...

Trial (by simulation) is our best technique for making choices!

Today’s representative example: MIPS
MIPS Programming Model
a representative simple RISC machine

In Comp 411 we’ll use a clean and sufficient subset of the MIPS-32 core Instruction set as an example ISA.

Fetch/Execute loop:
- fetch Mem[PC]
- PC = PC + 4
- execute fetched instruction (may change PC!)
- repeat!

†MIPS uses byte memory addresses. However, each instruction is 32-bits wide, and *must* be aligned on a multiple of 4 (word) address. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.

Processor State (inside the CPU)
- PC
- r0
- r1
- r2
- ... (32 bit "words")
- 00

Main Memory
- Addresses
- 32 bit “words” (4 bytes)
- next instruction

General Registers:
A small scratchpad of frequently used or temporary variables
Some MIPS Memory Nits

- Memory locations are 32 bits wide
  - BUT, they are *addressable* in different-sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/double)
- We also frequently need access to individual bits! (Instructions help to do this)
- Every BYTE has a unique address (MIPS is a byte-addressable machine)
- Every instruction is one word

<table>
<thead>
<tr>
<th>Addr</th>
<th>byte0</th>
<th>byte1</th>
<th>byte2</th>
<th>byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4:</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8:</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12:</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
MIPS Register Nits

- There are 32 named registers [$0, $1, …. $31]
- The operands of most instructions are registers
  - This means to operate on a variables in memory you must:
    - Load the value/values from memory into a register
    - Perform the instruction
    - Store the result back into memory
- Going to and from memory can be expensive
  (4x to 20x slower than operating on a register)
- Net effect: Keep variables in registers as much as possible!
- 2 registers have H/W specific “side-effects”
  (ex: $0 always contains the value ‘0’… more later)
- 4 registers are dedicated to specific tasks by convention
- 26 are available for general use
- Further conventions delegate tasks to other registers
MIPS Instruction Formats

All MIPS instructions fit in a single 32-bit word. Every instruction includes various “fields” that encode combinations of:

- a 6-bit operation or “OPCODE”
  - specifying one of < 64 basic operations
  - special “escape codes” enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants (“immediate” values) of various sizes, 16-bits, 5-bits, and 26-bits. Sometimes treated as signed values, sometimes not.

There are three basic instruction formats:

- **R-type**, 3 register operands (2 sources, destination)
  
<table>
<thead>
<tr>
<th>OP</th>
<th>r_s</th>
<th>r_t</th>
<th>r_d</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

- **I-type**, 2 register operands, 16-bit immediate constant
  
<table>
<thead>
<tr>
<th>OP</th>
<th>r_s</th>
<th>r_t</th>
<th>16-bit constant</th>
</tr>
</thead>
</table>

- **J-type**, no register operands, 26-bit immediate
  
<table>
<thead>
<tr>
<th>OP</th>
<th>26-bit constant</th>
</tr>
</thead>
</table>
MIPS ALU Operations

Sample coded operation: ADD instruction

| R-type: 0 1 0 1 1 0 1 0 0 1 0 1 0 0 0 0 | op = 0x00 dictating an ALU function |
| rd = 10 Reg[10] destination |
| rt = 9 Reg[9] source |
| func = 0x20 dictating an add |

What we prefer to write: add $10, $11, $9

The convention with MIPS assembly language is to specify the destination operand first, followed by source operands.

```
add rd, rs, rt:
Reg[rd] = Reg[rs] + Reg[rt]
```

“Add the contents of rs to the contents of rt; store the result in rd”

Similar instructions for other ALU operations:
- arithmetic: add, sub, addu, subu, mul, div
- compare: slt, sltu
- logical: and, or, xor, nor
- shift: sll, srl, sra, sslv, srav, srlv

References to register contents are prefixed by a "$" to distinguish them from constants or memory addresses.
ADD vs. ADDU

- The designers of MIPs wanted to insure that the results of an instruction were always *correct* according to their specification.
- This desire for correctness conflicts with the constraints of a finite representation, particularly in the case of some arithmetic operations. For example, adding two 32-bit numbers might result in a 33-bit result. Or even worse, when using a 2s-complement representation adding two positive numbers might result in a negative result. These anomalies are called *OVERFLOWS*.

- Two ways to fix this:
  - Perform an explicit test either before or after every operation (expensive overhead)
  - Generate an *Exception* in the case of an overflow

- ADD – generates exceptions on overflows
- ADDU – does not generate exceptions on overflows
- Guess which one most compilers use?
Logical “Word” operations

- The logical operations on words operate “bitwise”, that is each corresponding bit of both source operands is considered independently.

<table>
<thead>
<tr>
<th>$1$: 0x0000ff00</th>
<th>and</th>
<th>$3$, $1$, $2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2$: 0x0000f0f0</td>
<td>or</td>
<td>$3$, $1$, $2$</td>
</tr>
<tr>
<td>xor</td>
<td>nor</td>
<td>$3$, $1$, $2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$1$: 0000 0000 0000 0000 1111 1111 0000 0000</th>
<th>After AND:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2$: 0000 0000 0000 0000 0000 1111 0000 1111 0000</td>
<td>$3$: 0000 0000 0000 0000 0000 1111 0000 0000</td>
</tr>
<tr>
<td>After OR: $3$: 0000 0000 0000 0000 0000 1111 1111 1111 0000</td>
<td></td>
</tr>
<tr>
<td>After XOR: $3$: 0000 0000 0000 0000 0000 1111 1111 1111 0000</td>
<td></td>
</tr>
<tr>
<td>After NOR: $3$: 1111 1111 1111 1111 0000 0000 0000 1111</td>
<td></td>
</tr>
</tbody>
</table>

1s “select” bits, and 0s clear or “mask” bits
1s “set” bits
1s “complement” bits
1s clear and 0s “complement” bits
MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction

How are shifts useful?

Assembly: `sll $2,$2,4`

`sll rd, rt, shamt`

Reg[rd] = Reg[rt] << shamt

“Shift the contents of rt to the left by shamt; store the result in rd”

$s2$ Before: 0000000000000000000000000000000000011110010101010

$s2$ After: 00000000000000000000000000000000000111100101010100

Assembly: `sllv $2,$2,$8`

`sllv rd, rt, rs`

Reg[rd] = Reg[rt] << Reg[rs]

“This shift the contents of rt left by the contents of rs; store the result in rd”

This is peculiar syntax for MIPS, in this ALU instruction the rt operand precedes the rs operand. Usually, it's the other way around.
Utility of Shifts

Shifts are used in a wide range of applications

- For accessing bits
  - Recall that MIPS can access memory as bytes (8-bit blocks), shorts (16-bit blocks), words (32-bit blocks), and longs (64-bit blocks). But if you need to access or modify a particular bit, or a block of bits you need to use a combination of shifts and logical operations.
  - Example to examine the exponent part of a float

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

|  |
|---|---|
| $1$: 0xC22B0000 | srl $2$, $1$, 23 |
| $2$: 0x00000084 | andi $2$, $2$, 255 |

```
$1: 1100 0010 0010 1011 0000 0000 0000 0000
$1: 0000 0000 0000 0000 0000 0001 1000 0100
& 0000 0000 0000 0000 0000 0000 0000 1111
$1: 0000 0000 0000 0000 0000 0000 1000 0100
```
More Shift Utility

Shifts are used in a wide range of applications

- For multiplying by powers of 2
  - Recall that in most number representations each bit indicates a power of 2. Thus shifting bits as a unit is the same as increasing or decreasing the entire number by a power of 2. This is used a lot when accessing elements in arrays and other data structures.
  
- Example to access the 5th element in an array of words (i.e. integers or floats)

```c
int myArray = {0,5,10,15,20,25,30};

$1: 0x00000005   ⇒ sll$2,$1,2
$2: 0x00000019   ⇒ lw  $2,myArray($2)
```

We haven’t seen this instruction yet, but it gets words from memory.
**Even More Shift Utility**

Shifts are used in a wide range of applications

- For dividing by powers of 2
  - If shifts left are like multiplying by powers of 2, then shifts rights are like dividing by powers of two. However there are two versions of right shifts, an unsigned version, `srl` (we used it the slide before last), and a signed version, `sra`

<table>
<thead>
<tr>
<th>$1$: 0xFFFFFFFF821</th>
<th>sra</th>
<th>$2$, $1$, 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2$: 0xFFFFFFFFFE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| $1$: 1111 1111 1111 1111 1111 1000 0010 0001 |
| $2$: 0xFFFFFFFE |

- Why is there no signed left shift? What does this imply?
  - We need to be careful when using left shifts for multiplication!
MIPS ALU Operations with Immediate

addi instruction: adds register contents, signed-constant:

**Symbolic version:** \( \text{addi } $9, $11, -3 \)

\[ \text{addi } rt, rs, \text{imm:} \]
\[ \text{Reg}[\text{rt}] = \text{Reg}[\text{rs}] + \text{sxt}(\text{imm}) \]

“Add the contents of rs to const; store result in rt”

**Similar instructions for other ALU operations:**
- arithmetic: addi, addiu
- compare: slti, sltiu
- logical: andi, ori, xori, lui

Immediate values are sign-extended for arithmetic and compare operations, but not for logical operations.
Why Built-in Constants? (Immediate)

- Alternatives? Why not? Do we have a choice?
  - put constants in memory (was common in older instruction sets)
  - create more hard-wired registers for constants (like $0).
- SMALL constants are used frequently (50% of operands)
  - In a C compiler (gcc) 52% of ALU operations involve a constant
  - In a circuit simulator (spice) 69% involve constants
  - e.g., B = B + 1; C = W & 0x00ff; A = B + O;
- ISA Design Principle: Make the common cases fast
- MIPS Instructions:
  ```
  addi $29, $29, 4
  slti $8, $18, 10
  andi $29, $29, 6
  ori $29, $29, 4
  ```

One way to answer architectural questions is to evaluate the consequences of different choices using carefully chosen representative benchmarks (programs and/or code sequences). Make choices that are “best” according to some metric (cost, performance, …).
How About Larger Constants?

- In order to load a 32-bit constant into a register a two instruction sequence is used, "load upper immediate"

  \[ \text{lui} \quad \text{\$8,0xaaaa} \]

  1010101010101010 | 0000000000000000

- Then must get the lower order bits right, i.e.,

  \[ \text{ori} \quad \text{\$8,\$8,0xaaaaa} \]

  1010101010101010 | 0000000000000000

  0000000000000000 | 1010101010101010

  1010101010101010 | 1010101010101010

NOTE: In MIPS, Logical Immediate instructions (ANDI, ORI, XORI) *DO NOT* sign-extend their constant operand
First MIPS Program  
(fragment)

Suppose you want to compute the following expression:
\[ f = (g + h) - (i + j) \]
Where the variables \( f, g, h, i, \) and \( j \) are assigned to registers $16$, $17$, $18$, $19$, and $20$ respectively. What is the MIPS assembly code?

```
add $8,$17,$18       # (g + h)
add $9,$19,$20       # (i + j)
sub $16,$8,$9        # f = (g + h) - (i + j)
```

These three instructions are like our little ad-hoc machine from the beginning of lecture. Of course, assuming that all variables are in registers is rather limiting ....

Needed: instruction-set support for reading and writing locations in main memory...
MIPS Load & Store Instructions

MIPS is a LOAD/STORE architecture. This means that *all* data memory accesses are limited to load and store instructions, which transfer register contents to-and-from memory. ALU operations work only on registers.

I-type: \[ \begin{array}{|c|c|c|} \hline \text{OP} & \text{rs} & \text{rt} & \text{16-bit signed constant} \\ \hline \end{array} \]

\text{lw rt, imm(rs)} \quad \text{Reg[rt] = Mem[Reg[rs] + sxt(imm)]}

“Fetch into rt the contents of the memory location whose address is \textit{const} plus the contents of rs”

Abbreviation: \text{lw rt,imm} for \text{lw rt, imm($0$)}

\text{sw rt, imm(rs)} \quad \text{Mem[Reg[rs] + sxt(imm)] = Reg[rt]}

“Store the contents of rt into the memory location whose address is \textit{const} plus the contents of rs”

Abbreviation: \text{sw rt, imm} for \text{sw rt, imm($0$)}

**BYTE ADDRESSES, but lw and sw 32-bit word access word-aligned addresses. The resulting lowest two address bits must be 0!**
Storage Conventions

- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results

Addresses assigned at compile time

```
int x, y;
y = x + 37;
```

Compilation approach: LOAD, COMPUTE, STORE

```
lw $t0, 0x1008($0)
addi $t0, $t0, 37
sw $t0, 0x100C($0)
```

Or, more humanely, to

```
x=0x1008
y=0x100C
lw $t0, x
addi $t0, $t0, 37
sw $t0, y
```

RS defaults to Reg[0]

ex: x same as x($0)
### MIPS Register Usage Conventions

By convention, the MIPS registers are assigned to specific uses, and names. These are supported by the assembler, and higher-level languages. We’ll use these names increasingly.

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>4-7</td>
<td>arguments</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</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$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</td>
</tr>
</tbody>
</table>
Capabilities thus far: Expression Evaluation

Translation of an Expression:

```c
int x, y;
y = (x-3)*(y+123456)
```

```c
x: .word 0
y: .word 0
c: .word 123456
```

```c
lw $t0, x
addi $t0, $t0, -3
lw $t1, y
lw $t2, c
add $t1, $t1, $t2
mul $t0, $t0, $t1
sw $t0, y
```

- VARIABLES are allocated storage in main memory
- VARIABLE references translate to LD or ST
- OPERATORS translate to ALU instructions
- SMALL CONSTANTS translate to ALU instructions w/ built-in “immediate” constant
- “LARGE” CONSTANTS translate to initialized variables or a LUI-ORI sequence

NB: Here we assume that variable addresses fit into 16-bit constants!
Can We Run Any Algorithm?

Model thus far:

- Executes instructions sequentially –
- Number of operations executed = number of instructions in our program!

Good news: programs can’t “loop forever”!

- Halting problem is solvable for our current MIPS subset!

Bad news:

- Straight-line code
- Can’t do a loop
- Can’t reuse a block of code

Needed: ability to change the PC.
MIPS Branch Instructions

MIPS branch instructions provide a way of conditionally changing the PC to some nearby location...

I-type: 

| OPCODE | rs | rt | 16-bit signed constant |

| beq rs, rt, label # Branch if equal | bne rs, rt, label # Branch if not equal |

if (REG[RS] == REG[RT]) {
  PC = PC + 4*offset;
}

if (REG[RS] != REG[RT]) {
  PC = PC + 4*offset;
}

Notice on memory references offsets are multiplied by 4 (unlike LW and SW). So, branch targets are restricted to word boundaries!

NB: Branch targets are specified RELATIVE to the current instruction. The assembler hides the calculation of these offset values from the user, by allowing them to specify a target address (usually a label) and it does the job of computing the offset’s value. The size of the constant field (16-bits) limits the range of branches.
MIPS Jumps

• The range of MIPS branch instructions is limited to approximately ±32K instructions from the branch instruction. In order to branch farther an unconditional jump instruction is used.

• Instructions:
  - j label  # jump to label (PC = PC[31-28] || CONST[25:0]*4)
  - jal label  # jump to label and store PC+4 in $31
  - jr $t0  # jump to address specified by a register’s contents
  - jalr $t0,$ra  # jump to address specified by a register’s contents and store PC+4 in $t0

• Formats:
  - J-type: used for j
    - OP = 2
    - 26-bit constant
  - J-type: used for jal
    - OP = 3
    - 26-bit constant
  - R-type, used for jr
    - OP = 0
    - rs
    - 0
    - 0
    - 0
    - func = 8
  - R-type, used for jalr
    - OP = 0
    - rs
    - 0
    - rd
    - 0
    - func = 9
Now we can do a real program: Factorial...

Synopsis (in C):
- Input in n, output in ans
- r1, r2 used for temporaries
- follows algorithm of our earlier data paths.

MIPS code, in assembly language:

```
...  
addi $t0, $0, 1  # t0 = 1
lw    $t1, n     # t1 = n
loop: beq $t1, $0, done  # while (t1 != 0)
mul   $t0, $t0, $t1  # t0 = t0 * t1
addi $t1, $t1, -1  # t1 = t1 - 1
beq   $0, $0, loop  # Always branch
done: sw $t0, ans   # ans = r1
...  
n:    .word 123
ans:  .word 0
```

```c
int n, ans;
r0 = 1;
r1 = n;
while (r1 != 0) {
    r0 = r0 * r1;
    r1 = r1 - 1;
}
ans = r0;
```
To summarize:

### MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $s0-$a3, $v0-$v1, $gp, $sp, $ra, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>2^{16} memory words</td>
<td>Memory[0], Memory[4], ...</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 =$s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 =$s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 =$s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td></td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[+$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[+$s2 + 100] =$s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[+$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[+$s2 + 100] =$s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2^{16}</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
<td>$(s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
<td>$(s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>$(s2 &lt; $s3) $s1=1, else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td></td>
<td>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
<td>$(s2 &lt; 100) $s1=1, else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td></td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra= PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
# MIPS Instruction Decoding Ring

<table>
<thead>
<tr>
<th>OP</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ALU</td>
<td>j</td>
<td>jal</td>
<td>beq</td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>addi</td>
<td>addiu</td>
<td>slti</td>
<td>sltiu</td>
<td>andi</td>
<td>ori</td>
<td>xori</td>
<td>lui</td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lw</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sw</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>sll</td>
<td>srl</td>
<td>sra</td>
<td>slv</td>
<td>srlv</td>
<td>sra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>jr</td>
<td>jalr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>mul</td>
<td>div</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>add</td>
<td>addu</td>
<td>sub</td>
<td>subu</td>
<td>and</td>
<td>or</td>
<td>xor</td>
<td>nor</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• We will use a subset of MIPS instruction set as a prototype
  • Fixed-size 32-bit instructions
  • Mix of three basic instruction formats
    • R-type - Mostly 2 source and 1 destination register
    • I-type - 1-source, a small (16-bit) constant, and a destination register
    • J-type - A large (26-bit) constant used for jumps
  • Load/Store architecture
  • 31 general purpose registers, one hardwired to 0, and, by convention, several are used for specific purposes.
• ISA design requires tradeoffs, usually based on
  • History
  • Art
  • Engineering
  • Benchmark results