Instruction Set Architecture

Contents

- Instruction
- Instruction set
  - Number of Address
  - Addressing modes
  - Operand types
  - Operations types
- Assembly programming

Instruction

- Elements
  - Opcode: What to do
  - Operand(s): data source(s)/destination(s)
- Representation
  - Binary bits
  - Symbolic representation
    - Add, SUB, LOAD, etc.
    - E.G.: ADD X, Y
### Instruction Length

- Affected by
  - Memory size/organization, register numbers, bus structure, etc
- Flexibility vs. Implementation Complexity
- Memory–transfer consideration
- Fixed vs. non-fixed instructions

### Instruction Set

- The collection of different instructions CPU can understand and execute
- Different instructions
  - Number of addresses/addressing modes
  - Operand types
  - Operation types

### Number of Addresses

- 3 addresses
  - Operand 1, Operand 2, Result
    - e.g. a=b+c
- 2 address
  - One address doubles as operand and result
    - e.g. a=a+c
- 1 address
  - Implicit second address (accumulator)
- 0 address
  - All addresses are implicitly defined
  - Stack based computer
Example: $Y = (A-B)/(C+D \times E)$

- **Three Addresses:**
  
<table>
<thead>
<tr>
<th>Operation</th>
<th>Addresses</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB Y, A, B</td>
<td># Y ← A-B</td>
<td></td>
</tr>
<tr>
<td>MPY T, D, E</td>
<td># T ← DxE</td>
<td></td>
</tr>
<tr>
<td>ADD T, T, C</td>
<td># T ← T+C</td>
<td></td>
</tr>
<tr>
<td>DIV Y, Y, T</td>
<td># Y ← Y/T</td>
<td></td>
</tr>
</tbody>
</table>

- **Two Addresses**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Addresses</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV Y, A</td>
<td>#Y ← A</td>
<td></td>
</tr>
<tr>
<td>SUB Y, B</td>
<td>#Y ← Y-B</td>
<td></td>
</tr>
<tr>
<td>MOV T, D</td>
<td>#T ← D</td>
<td></td>
</tr>
<tr>
<td>MPY T, E</td>
<td>#T ← TxE</td>
<td></td>
</tr>
<tr>
<td>ADD T, C</td>
<td># T ← T+C</td>
<td></td>
</tr>
<tr>
<td>DIV Y, T</td>
<td># Y ← Y/T</td>
<td></td>
</tr>
</tbody>
</table>

Example: $Y = (A-B)/(C+D \times E)$

- **One Addresses:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Addresses</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD D</td>
<td># AC ← D</td>
<td></td>
</tr>
<tr>
<td>MPY E</td>
<td># AC ← ACxE</td>
<td></td>
</tr>
<tr>
<td>ADD C</td>
<td># AC ← AC+C</td>
<td></td>
</tr>
<tr>
<td>STOR Y</td>
<td># Y ← AC</td>
<td></td>
</tr>
<tr>
<td>LOAD A</td>
<td># AC ← A</td>
<td></td>
</tr>
<tr>
<td>SUB B</td>
<td># AC ← AC-B</td>
<td></td>
</tr>
<tr>
<td>DIV Y</td>
<td>#...</td>
<td></td>
</tr>
<tr>
<td>STOR Y</td>
<td>#...</td>
<td></td>
</tr>
</tbody>
</table>

- **Stack (0 address):**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Addresses</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A little bit about Stack

- A list of data element
- Data can be added or removed from one of its end (top of the stack)
- Static operations
  - Push
  - Pop
  - Unary operation (such as negation)
  - Binary operation (such as multiplication)
Stack Operation

- Top
- 5
- 10
- More
- Fewer

How Many Addresses

- More addresses
  - More complex (powerful?) instructions
  - More registers
    - Inter-register operations are quicker
  - Fewer instructions per program
- Fewer addresses
  - Less complex (powerful?) instructions
  - More instructions per program
  - Faster fetch/execution of instructions
Instruction Addressing

- What
  - How is the address of an operand specified
- Different addressing mode
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register indirect
  - Displacement
  - Stack

Immediate Addressing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Operand</th>
</tr>
</thead>
</table>

- Operand is part of instruction
- Operand = address field
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space
Direct Addressing Diagram

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Address A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operand</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indirect Addressing

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- EA = (A)
  - Look in A, find address (A) and look there for operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator

Indirect Addressing (Cont’d)

- Large address space
- $2^n$ where $n =$ word length
- May be nested, multilevel, cascaded
  - e.g. EA = (((A)))
  - Draw the diagram?
- Multiple memory accesses to find operand
- Hence slower
Indirect Addressing Diagram

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Address A</th>
</tr>
</thead>
</table>

Memory

Pointer to operand

Operand

Register Addressing

- Operand is held in register named in address filed
- EA = R
- Limited number of registers
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch

Register Addressing (cont’d)

- No memory access
- Very fast execution
- Very limited address space
- Multiple registers helps performance
  - Requires good assembly programming or compiler writing
  - Compare with Direct addressing
Register Addressing Diagram

Register Indirect Addressing

- EA = (R)
- Operand is in memory cell pointed to by contents of register R
  - Compare with indirect addressing
- Large address space ($2^n$)
- One fewer memory access than indirect addressing

Register Indirect Addressing Diagram
Displacement Addressing

- EA = A + (R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa

Displacement Addressing Diagram

Indexed Addressing

- A = base
- R = displacement
- EA = A + R
- Good for accessing arrays
  - EA = A + R
  - R++
Stack Addressing

- Operand is (implicitly) on top of stack
- e.g.
  - ADD Pop top two items from stack and add

Basic Addressing Mode Summary

<table>
<thead>
<tr>
<th>Mode</th>
<th>Algorithm</th>
<th>Principal Advantage</th>
<th>Principal Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>O(A)</td>
<td>No memory references</td>
<td>Limited operand in stack</td>
</tr>
<tr>
<td>Direct</td>
<td>E = A</td>
<td>Simple</td>
<td>Limited address space</td>
</tr>
<tr>
<td>Indirect</td>
<td>E = (A)</td>
<td>Large address space</td>
<td>Multiple memory references</td>
</tr>
<tr>
<td>Register</td>
<td>E = R</td>
<td>No memory reference</td>
<td>Limited address space</td>
</tr>
<tr>
<td>Register indirect</td>
<td>E = (R)</td>
<td>Large address space</td>
<td>Extra memory reference</td>
</tr>
<tr>
<td>Displacement</td>
<td>E = A + d</td>
<td>Flexibility</td>
<td>Complexity</td>
</tr>
<tr>
<td>Stack</td>
<td>E = top of stack</td>
<td>No memory reference</td>
<td>Limited applicability</td>
</tr>
</tbody>
</table>

Memory alignment

- Addressing a data type larger than byte must be aligned
- An access to an object of size S bytes at byte address A is aligned if \( A \mod s = 0 \).
  - 0bXXXXXXXXX byte (8bit) aligned
  - 0bXXXXXXXXXX0 half word (16bit) aligned
  - 0bXXXXXXXXXXX0 word (32bit) aligned
  - 0bXXXXXXXXXXXX0 double word (64bit) aligned
- Why aligned?
  - Misalignment causes implementation complications
Little/Big Endian

- Little Endian:
  - put the least-significant byte first
- Big Endian:
  - put the most-significant byte first

Big/Little Endian Example

- 32bit data 0xFABC0123 at address 0xFF20

<table>
<thead>
<tr>
<th></th>
<th>0xFF20</th>
<th>0xFF21</th>
<th>0xFF22</th>
<th>0xFF23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Endian</td>
<td>0xFA</td>
<td>0xBC</td>
<td>0x01</td>
<td>0x23</td>
</tr>
<tr>
<td>Little Endian</td>
<td>0x23</td>
<td>0x01</td>
<td>0xBC</td>
<td>0xFA</td>
</tr>
</tbody>
</table>

Operand Types

- Numbers
  - Integer
    - Byte, short, word, long word
    - Unsigned/signed
  - Floating point
    - Float
    - Double
- Characters
- Logical Data
Operation Types

- Instruction types
  - Data Transfer
    - Moving data among registers, memory, and I/O devices
    - Load/store
  - Arithmetic
    - Add, sub, multiply, etc
  - Logical
    - and, or, not, xor
  - Logic shifting and rotating
  - Conversion
  - System Control
  - Transfer of control

Logic Shifting and Rotating

Transfer of Control

- Branch
  - e.g. branch to x if result is zero
- Procedure call
Branch

- Unconditional branch
  - One of its operands is the address of next instruction to be executed

- Conditional branch
  - Using status register
    - Execution of an instruction may change the status, e.g. positive, negative, overflow, ...
  - Multi-address format

Branch Example

Procedure Call

- Procedure
  - Self-contained computer program incorporated into a larger program
    - Economy and modularity

- Procedure call
  - Call instruction
    - Branches to the procedure
  - Return instruction
    - Returns to the place where it was called
Call Instruction

- Use register to save return address (next instruction)
  - R ← PC + sizeof (instruction)
  - PC ← X
- Save the return address at the start of procedure
  - X ← PC + sizeof (instruction)
  - PC ← X + sizeof (instruction)
- Problems?

Nested Procedure Calls

Use of Stack

- Last procedure call returns first (LIFO)
- Pass parameters
  - Stack frame
Assembly Programming

Use symbolic representations of instructions to program a computer.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Label</th>
<th>Operation</th>
<th>Register</th>
<th>Format</th>
</tr>
</thead>
</table>
| 000100 | 000000 | 0000 | LB R1, 40(R2) | R1 | 0000000
| 000100 | 000000 | 0001 | LH R1, 40(R2) | R1 | 0000000
| 000100 | 000000 | 0002 | LW R1, 40(R2) | R1 | 0000000
| 000100 | 000000 | 0003 | LD R1, 40(R2) | R1 | 0000000

Case Study: the MIPS Architecture

- General purpose registers/Load-store
  - Registers
    - 32 GPR: R0, ..., R31
    - R0 is always 0
    - 32 floating point registers (FPR): F0, ..., F31
  - Special purpose registers
    - e.g. status register.

- Data types
  - 8, 16, 32, 64 bit
    - Ex: LB R1, 40(R2)
    - LH R1, 40(R2)
    - LW R1, 40(R2)
    - LD R1, 40(R2)
MIPS Addressing

- Can be big/little endian
- All memory access must be aligned
- Addressing modes
  - Register indirect
    - Ex: JR R1
  - Displacement
    - Ex: ST R1, 100(R2)
- Addressing modes encoded in opcode

Encoding MIPS64 ISA

- Fixed length encoding – 32 bits
  - I-type instructions
  - R-type instructions
  - J-type instructions

I-type instructions

- I-Type instruction:
  
<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op-code</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
</tbody>
</table>

- Encodes:
  - Loads and stores of bytes, half words, words, dwords
  - Condition branch
  - Example
    - LW R1, 32(R4)
      
      100011 00100 00001 0000000000000000
    - BENG R0, R1, -1
      
      000101 00000 00001 11111111111111111
R-type instructions

- **R-Type instruction:**
  - **Op code:** 6 bits
  - **rs:** 5 bits
  - **rt:** 5 bits
  - **rd:** 5 bits
  - **func:** 11 bits

**Register-Register ALU operations**
- **rd:** rs func rt
- Function encodes data path operation: add, sub, slt, and
- Read/write special registers and moves

**Example**
- **DADD R2, R3, R4**
  - 000000
  - 00011
  - 00100
  - 00010
  - 00000 00011 00100 00010 00000

J-type instructions

- **J-Type instruction:**
  - **Op code:** 6 bits
  - **immediate:** 26 bits

**Encodes:**
- Jump and jump & link
- Trap and return from exception

**Example**
- **J 0x300**
  - 000010
  - 00000000000000001100000000

MIPS Operations

- **Simple instructions**
  - Load/store, add, subtract, multiply, divide, shift, ...
  - **Ex:** DADD, DADDI, DADDU

- **Control Flow**
  - Compare equal/not equal, compare less, ...
  - **Ex:** BEQZ, JR

- **Floating point**
  - Load/store, add, subtract, ...
  - **Ex:** ADD.D, ADD.S, MUL.D, MUL.S
### Load/Store

<table>
<thead>
<tr>
<th>Instruction type/opcode</th>
<th>Instruction meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store</td>
<td>Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is + offset + content of a GPR</td>
</tr>
<tr>
<td>1L, 2L</td>
<td>Load byte, load byte signed, store byte (low byte integer register)</td>
</tr>
<tr>
<td>1H, 2H</td>
<td>Load half word, load half word unsigned, store half word (low half-word integer register)</td>
</tr>
<tr>
<td>3L, 4L, 6L, 8L, 10L</td>
<td>Load word, load word unsigned, store word (higher integer registers)</td>
</tr>
<tr>
<td>1L</td>
<td>Load double word, store double word (low double-word integer register)</td>
</tr>
<tr>
<td>1.5, 1.5, S, D</td>
<td>Load float, load FP float, store FP float, store FP float (FP register)</td>
</tr>
<tr>
<td>1L, 2L, 3L</td>
<td>Copy from GPR to another GPR or from another GPR to another GPR</td>
</tr>
<tr>
<td>1L</td>
<td>Copy 32-bit float FP register to another FP register</td>
</tr>
</tbody>
</table>

### Arithmetical/Logical

<table>
<thead>
<tr>
<th>Instruction type/opcode</th>
<th>Operations on integer or logical data in GPRs; signed arithmetic; trap on overflow, abort on immediate, return from interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1L, 2L, 4L, 8L, D, D8</td>
<td>Add, add immediate (all integers are 32 bits), signed and unsigned</td>
</tr>
<tr>
<td>3L, 5L, 9L</td>
<td>Subtract, signed and unsigned</td>
</tr>
<tr>
<td>1L</td>
<td>Multiply and divide, signed and unsigned, multiply-accumulate; operations take and yield 64-bit values</td>
</tr>
<tr>
<td>3L, 9L, 10L</td>
<td>And, not immediate</td>
</tr>
<tr>
<td>2L, 10L, 20L</td>
<td>Or, or immediate, exclusive or, exclusive or immediate, load operating immediate, then sign-extend 64-bit integer</td>
</tr>
<tr>
<td>D1, D10, 20L</td>
<td>Shifts; both immediate (0...7) and variable form (0...31), shifts are left logical, right logical, right arithmetic</td>
</tr>
<tr>
<td>S1, S10, 20L</td>
<td>Set less than, not less than immediate, signed and unsigned</td>
</tr>
</tbody>
</table>
## Arithmetical/Logical

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU R1, R2, R3</td>
<td>Add signed</td>
<td>Regs[R1] = Regs[R2] + Regs[R3]</td>
</tr>
<tr>
<td>ADDIU R1, R2, R3</td>
<td>Add immediate</td>
<td>Regs[R1] = Regs[R2] + R3</td>
</tr>
<tr>
<td>LUI R1, #R2</td>
<td>Load upper</td>
<td>Regs[R1] = 0b44444444</td>
</tr>
<tr>
<td>SLL R1, R2, #5</td>
<td>Shift left</td>
<td>Regs[R1] = Regs[R2] &lt;&lt; 5</td>
</tr>
<tr>
<td>SLT R1, R2, R3</td>
<td>Set less</td>
<td>if (Regs[R2] &lt; Regs[R3]) Regs[R1] = 0 else Regs[R1] = 1</td>
</tr>
</tbody>
</table>

## Floating point

- **Floating point**: FP operations in DP and SP formats

- **ADD, SUB, MUL, DIV**: Add, subtract, multiply, and divide FP numbers
- **FADD, FSUB, FMUL, FADD**: Add, subtract, multiply, and divide FP numbers

- **FCVT**: Convert FP number to int, and vice versa
- **FCVT**: Convert FP number to int, and vice versa

- **FP**: FP operations on FP registers
- **FADD, FSUB, FMUL, FADD**: Add, subtract, multiply, and divide FP numbers

- **FP**: FP operations on FP registers
- **FADD, FSUB, FMUL, FADD**: Add, subtract, multiply, and divide FP numbers

- **FP**: FP operations on FP registers
- **FADD, FSUB, FMUL, FADD**: Add, subtract, multiply, and divide FP numbers

- **FP**: FP operations on FP registers
- **FADD, FSUB, FMUL, FADD**: Add, subtract, multiply, and divide FP numbers

- **FP**: FP operations on FP registers
- **FADD, FSUB, FMUL, FADD**: Add, subtract, multiply, and divide FP numbers

- **FP**: FP operations on FP registers
- **FADD, FSUB, FMUL, FADD**: Add, subtract, multiply, and divide FP numbers

## Control Flow

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL name</td>
<td>Jump Link</td>
<td>Regs[R3] = PC + 4; PC = (name);</td>
</tr>
<tr>
<td>JAL R2, #R3</td>
<td>Jump Link register</td>
<td>Regs[R3] = PC + 4; PC = Regs[R2]</td>
</tr>
<tr>
<td>JAL R2, R3</td>
<td>Jump register</td>
<td>PC = Regs[R2]</td>
</tr>
<tr>
<td>BEQZ R1, =name</td>
<td>Branch equal zero</td>
<td>if (Reg[R1] = 0) PC = target;</td>
</tr>
<tr>
<td>BNE R1, =name</td>
<td>Branch not equal zero</td>
<td>if (Reg[R1] != 0) PC = target;</td>
</tr>
<tr>
<td>#REG R1, R2</td>
<td>Conditional move if zero</td>
<td>if (Reg[R2] = 0) Regs[R1] = Regs[R2]</td>
</tr>
</tbody>
</table>
MIPS Assembly Programming


Summary

- Instructions
- Instruction set
  - Number of Address
  - Addressing modes
  - Operand types
  - Operations types
- Assembly programming