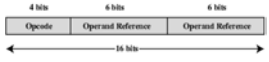
Instruction Set Architecture

Contents

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

Instruction

- Elements
  - Opcode: What to do
  - Oprand(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 x E)

- Three Addresses:

```
SUB   Y, A, B     # Y← A-B
MPY   T, D, E     # T← DxE
ADD   T, T, C     # T← T+C
DIV   Y, Y, T     # Y← Y/T
```

- Two Addresses

```
MOV   Y, A        #Y← A
SUB   Y, B        #Y←Y-B
MOV   T, D        #T←D
MPY   T,E         #T←TxE
ADD   T,C         # T←T+C
DIV   Y, T        # Y←Y/T
```

Example: Y=(A-B)/(C+D x E)

- One Addresses:

```
LOAD  D     # AC← D
MPY   E     # AC← ACxE
ADD   C     # AC← AC+C
STOR  Y     # Y← AC
LOAD  A     # AC←A
SUB   B     # AC←AC-B
DIV   Y     #...
STOR  Y     #...
```

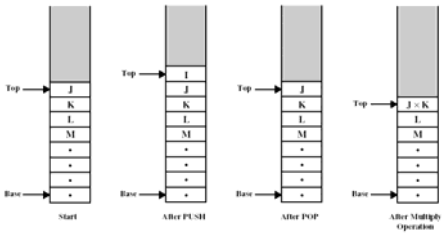
- Stack (0 address)

```
Push  A
Push  B
SUB
Push  C
Push  D
Push  E
MPY
ADD
DIV
Pop   Y
```

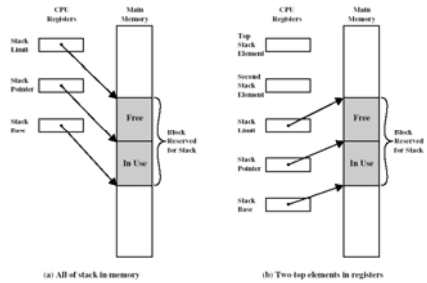
A little bit about Stack

- 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



Stack Organization



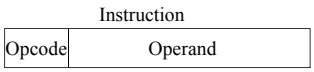
How Many Addresses

- 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

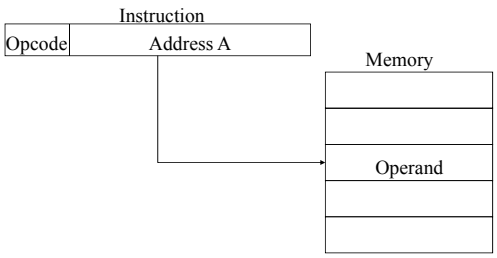


- Operand is part of instruction
- Operand = address field
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

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



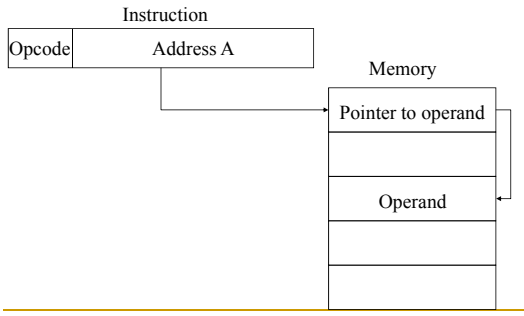
Indirect Addressing

- 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



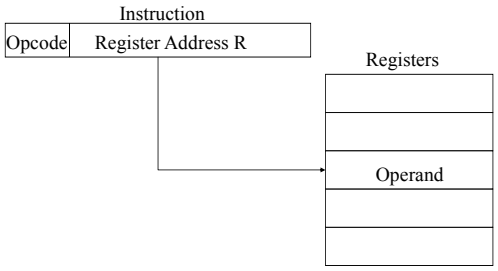
Register Addressing

- 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

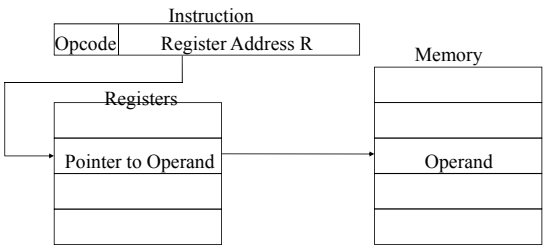
Register Addressing Diagram



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

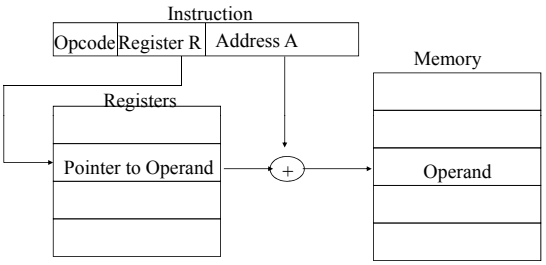
Register Indirect Addressing Diagram



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

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Memory alignment

- Addressing a data type large 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.
  - 0bxxxxxxxxxx byte (8bit) aligned
  - 0bxxxxxxxxx0 half word (16bit) aligned
  - 0bxxxxxxxx00 word (32bit) aligned
  - 0bxxxxxxx000 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

	0xFF20	0xFF21	0xFF22	0xFF23
Big Endian	0xFA	0xBC	0x01	0x23
Little Endian	0x23	0x01	0xBC	0xFA

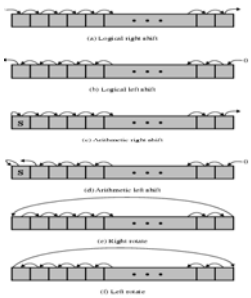
Operand Types

- 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



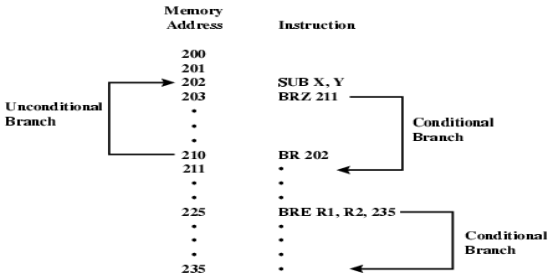
Transfer of Control

- 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



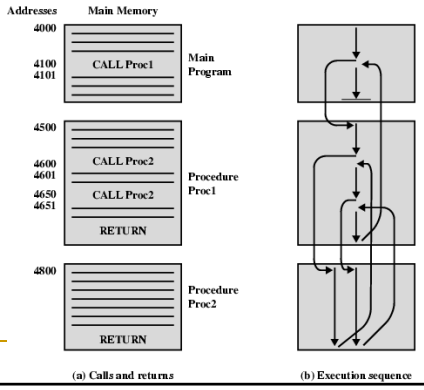
Procedure Call

- 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

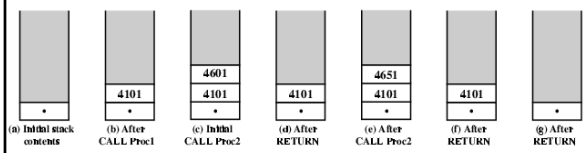
Call Instruction

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

Nested Procedure Calls

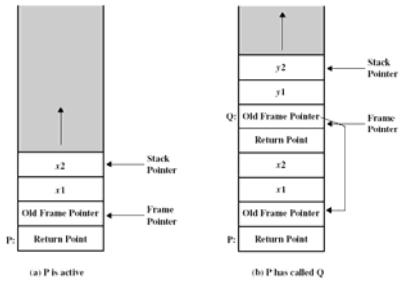


Use of Stack



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

Stack Frame



Assembly Programming

Use symbolic representations of instructions to program a computer.

Address	Contents
101	0010 0000 0001
102	0001 0010 0000 0010
103	0001 0010 0000 0011
104	0011 0010 0000 0100
201	0000 0000 0000 0010
202	0000 0000 0000 0011
203	0000 0000 0000 0100
204	0000 0000 0000 0000

(a) Binary program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(b) Symbolic program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(c) Hexadecimal program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

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

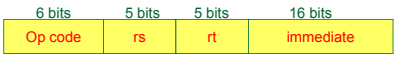
- Can be big/little endian
- All memory access must be aligned
- Addressing modes
  - Register indirect
    - Ex: JR R1
  - Displacement
    - Ex: ST R1, 100(R2)
- Addressing modes encoded in opcode

Encoding MIPS64 ISA

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

I-type instructions

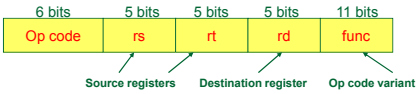
- I-Type instruction:



- Encodes:
  - Loads and stores of bytes, half words, words, dwords
  - Condition branch
- Example
  - LW R1, 32(R4)
    100011 00100 00001 0000000000100000
  - BENQ R0, R1, -1
    000101 00000 00001 1111111111111111

R-type instructions

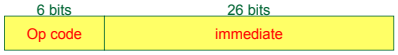
- R-Type instruction:



- 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 100000

J-type instructions

- J-Type instruction:



- 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

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR*
LB,LBU,SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH,LHU,SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW,LWU,SW	Load word, load word unsigned, store word (to/from integer registers)
LD,SD	Load double word, store double word (to/from integer registers)
L.S,L.D,S.S,S.D	Load SP float, load DP float, store SP float, store DP float
MFC0,MTC0	Copy from/to GPR to/from a special register
MOV.S,MOV.D	Copy one SP or DP FP register to another FP register
MFC1,MTC1	Copy 32 bits from/to FP registers to/from integer registers

Load/Store

Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1]$\leftarrow_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1]$\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1]$\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1]$\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1]$\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1]$\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40+Regs[R3]]##Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0]$\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0]$\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]]$\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]]$\leftarrow_{32}$ Regs[R3]
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]]$\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]]$\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]]$\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]]$\leftarrow_{8}$ Regs[R2]$_{56..63}$

Arithmetical/Logical

Arithmetic/logical	*Operations on integer or logical data in GPRs; signed arithmetic trap on overflow*
DADD,DADDI,DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB,DSUBU	Subtract; signed and unsigned
DMUL,DMULU,DDIV, DDIVU,MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL,DSRL,DSRA,DSLLV, DSRLV,DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT,SLTI,SLTU,SLTIU	Set less than, set less than immediate; signed and unsigned

Arithmetical/Logical

Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	Regs[R1]←Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]←Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]←0^{32}##42##0^{16}
DSLL R1,R2,#5	Shift left logical	Regs[R1]←Regs[R2]<<5
DSLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0

Floating point

Floating point	*FP operations on DP and SP formats*
ADD.D,ADD.S,ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D,SUB.S,ADD.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D,MUL.S,MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D,MADD.S,MADD.PS	Multiply-add DP, SP numbers and pairs of SP numbers
DIV.D,DIV.S,DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C._.D,C._.S	DP and SP compares: "_" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

Control Flow

Example instruction	Instruction name	Meaning
J name	Jump	$PC_{36..63}$←name
JAL name	Jump and link	Regs[R31]←PC+4; $PC_{36..63}$←name; ((PC+4)–2^{27}) ≤ name < ((PC+4)+2^{27})
JALR R2	Jump and link register	Regs[R31]←PC+4; PC←Regs[R2]
JR R3	Jump register	PC←Regs[R3]
BEQZ R4,name	Branch equal zero	if (Regs[R4]==0) PC←name; ((PC+4)–2^{17}) ≤ name < ((PC+4)+2^{17})
BNE R3,R4,name	Branch not equal zero	if (Regs[R3]!= Regs[R4]) PC←name; ((PC+4)–2^{17}) ≤ name < ((PC+4)+2^{17})
MOVZ R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1]←Regs[R2]

MIPS Assembly Programming

"MIPS Assembly Language Programming",
http://www.eecs.harvard.edu/~ellard/Courses/cs50-asm.pdf

Summary

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