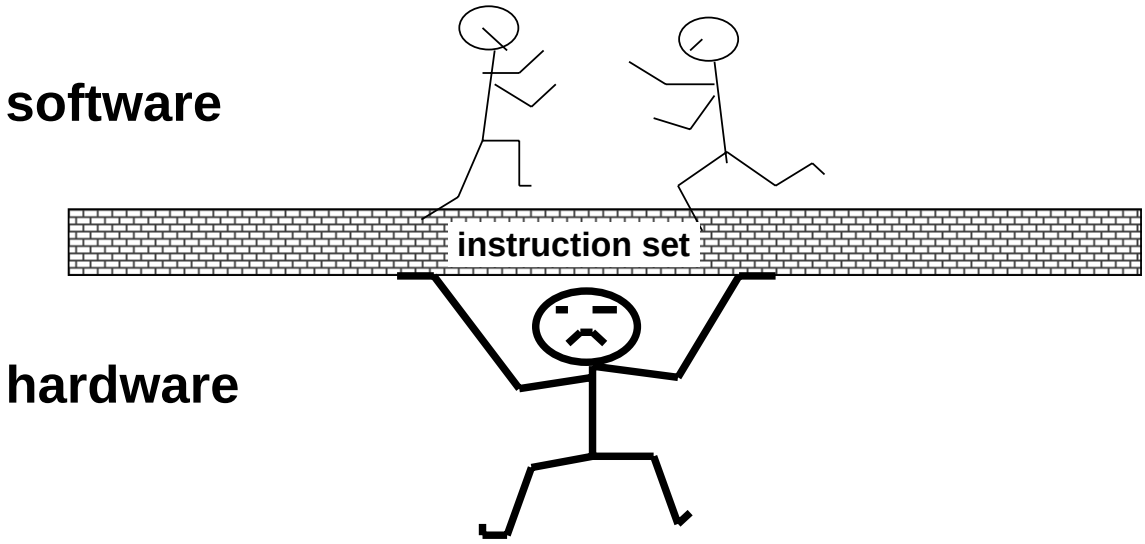
Computer Architectures

Chapter 2

Tien-Fu Chen

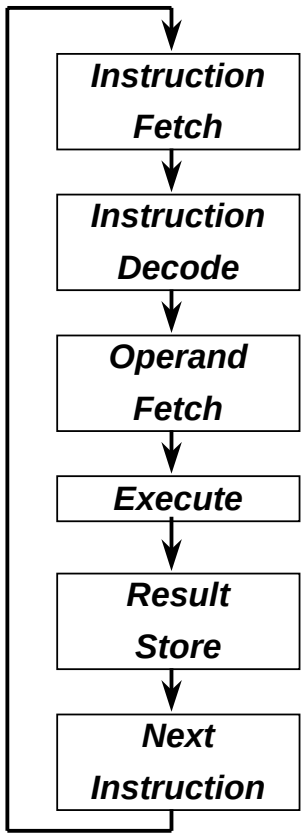
National Chung Cheng Univ.

Instruction Set Design



Which is easier to change/design???

Instruction Set Architecture: What must be specified?



- **Instruction Format or Encoding**
  - how is it decoded?
- **Location of operands and result**
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- **Data type and Size**
- **Operations**
  - what are supported
- **Successor instruction**
  - jumps, conditions, branches
  - *fetch-decode-execute is implicit!*

Instruction Sets

- **Instruction Set**

  An agreement between architects and machine language programmers

- **Aspects**
  - Operations
    - Arithmetic and logical +, -, X, /...
    - Data Transfer - load/store
    - Control - branch, jump, call, return
    - Floating Point operations - FADD, FDIV
    - String
    - System support
    - HLL (high-level languages)
  - Operands
    - operand storage
    - number of operands
    - addressing operands
    - Type and size of operands
    - Implicit/Explicit operands

Evolution of Instruction Sets

- **Major advances in computer architecture are typically associated with landmark instruction set designs**
  - Ex: Stack vs GPR (System 360)
  - CISC vs RISC
- **Design decisions must take into account:**
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
- **And they in turn influence these factors**

What is good for instruction set?

- **Orthogonality**

  No special registers, few special cases, all operand modes available with any data type or instruction type
- **Completeness**

  Support for a wide range of operations and target applications
- **Regularity**

  No overloading for the meanings of instruction fields
- **Streamlined**

  Resource needs easily determined
- **Ease of compilation (programming?)**
- **Ease of implementation**
- **Scalability**

Classifying Instruction Set Architecture

- **Stack: 0 address**

  add (sp)<- (sp) + (sp-1)

- **Accumulator: 1 address, 1+x address**

  add A acc=acc + mem[A]
  addx A acc=acc + mem[A + x]

- **General Purpose Register:** 2,3 address

  add A,B EA(A)=EA(A) + EA(B)
  add A B C EA(A)=EA(B) + EA(C)

- **Load/Store: 3 address**

  add Ra Rb Rc Ra=Rb + Rc
  load Ra Rb Ra= mem[Rb]
  store Ra Rb mem[Rb]= Ra

  - **access memory only with load and store instructions**

Comparing Number of Instructions

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

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

Issues of Operands

❑ Operand Storage

either registers, memory, implicit

❑ Number of operands

- 0: stack machine
- 1: accumulator
- 2,3: register set/memory

	Advantages	Disadvantages
reg-reg (load/store) (0,3)	simple, fixed-length, fixed cycles	larger instr count, must be loaded into registers
reg-mem (1,2)	not necessarily loaded easy encoding good code density	inequivalent operands clocks may vary
mem-mem (3,3)	most compact, no register limit	large variation of instruction length, bottleneck in memory

Issues of Operands(Cont)

❑ Addressing Operands

- Endian Convention
  Ordering the bytes within a word
- **Big Endian**: MSB at xx00

word addr	MSB			LSB
0	0	1	2	3
4	4	5	6	7

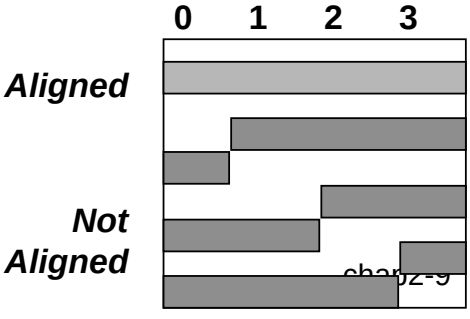
ex. : IBM, Motorola

- **Little Endian**: LSB at xx00

word addr	MSB			LSB
0	3	2	1	0
4	7	6	5	4

ex. Intel, Dec

- Aligned vs. Misaligned



Issues of Operands(Cont)

❑ Addressing Mode:determination of effective address

- register Ri
- Immediate #n
- Base+displacement M[Ri + #n]
- Register indirect M[Ri]
- Indexed M[Ri, Rj]
- Absolute M[#n]
- Memory indirect M[M[Ri]]
- PC-relative M[PC + #n]
- Auto-Increment M[Ri]; Ri+=d
- Auto-Decrement Ri-=d; M[Ri]

Implicit/Explicit operands

❑ Compiler issue

❑ Take branch on conditional code as example

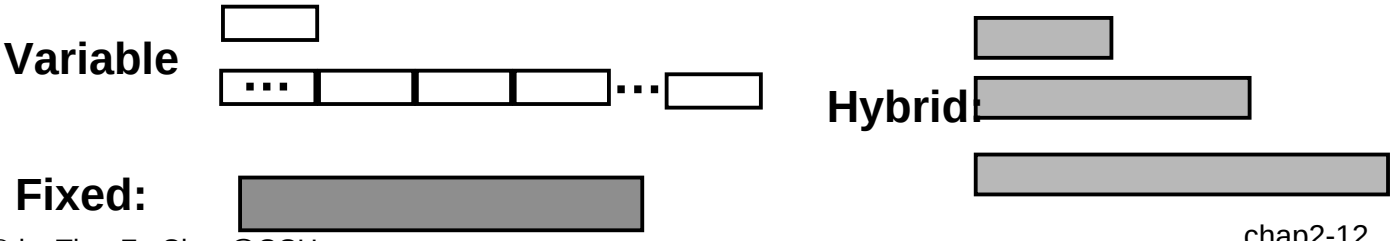
	Advantages	Disadvantages
Condition code	-be set for free	-Constrains code reordering, -Extra state to save
Condition register	-Simple, -No special state to save	-Use up a register
Compare & branch	-No extra compare, -No state passed between instructions	-May be too much work per instruction

Instruction Set Architecture

❑ CISC vs. RISC

	CISC	RISC
Instruction set	-large instrn variety -variable formats -variable instr length	-Small instr. Set -Fixed format -Fixed length
operand storage	reg-mem mem-reg	reg-reg
Addressing mode	complex	simple
GP registers	8-24 +special regs	large number
CPU control	microcode, hardwired	hardwired
cache/TLB	extern cache	on-chip cache&TLB

❑ Instruction Format Widths



Encoding Instruction

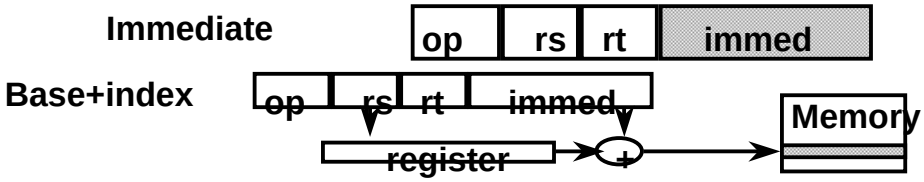
Fixed	Hybrid	Variable
L/S load/store	R/M Reg-mem	R+M Reg-plus-mem
3-addr format 32b instr size	2-addr format 16/32/64b instr sizes	2/3-addr format byte-variable size
IBM RS/6000 IBM PowerPC MIPS R2000 HP PA RISC DEC Alpha AMD 29000 SPARC (reg window)	IBM S/360 IBM 3033 Fujitsu Hitachi Intel x86 (byte-variable size)	VAX Motorola 680x0

DLX

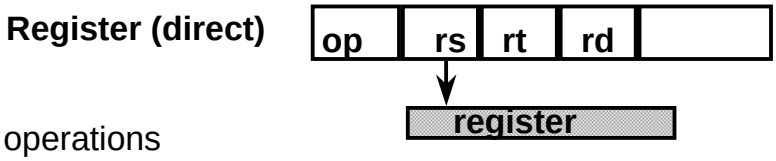
- **A RISC architecture related to MIPS**
- **32-bit byte addresses**
- **Load/Store - only displacement**
- **Registers**
  - 32 32-bit General-Purpose Registers
  - 16 64-bit (32 32-bit) Floating-Point Reg
  - FP status register
- **Emphasize**
  - A simple load/store instruction set
  - Design for pipelining efficiency
  - An easily decoded instruction set
  - Efficiency as a compiler target

DLX (Cont)

- **Three fixed-length instruction format**
  - I-Format



    - ALU ops with immed **rd<-rs1 op #immed**
    - load/store **rd<-> M[rs1+#imm]**
    - conditional branch **pc<-cond: pc+=#im**
  - R-Format



    - Reg-Reg ALU operations
      rd <- rs1 func rs2
  - J-Format

unconditional jumps
pc += #offset

DLX Instruction Set

❑ Data transfers

load/store full word
load/store byte/halfword
load/store FP single/double
moves between GPR and FP registers

❑ Arithmetic/Logical

Add/Substract unsigned, immediate
MUL/DIV signed, unsigned
AND, OR, XOR immediate
Load high immediate
Shift left/right

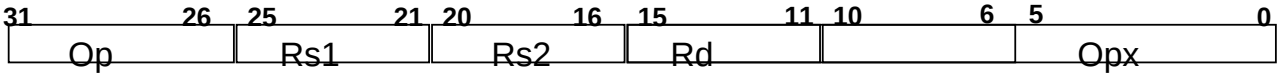
❑ Control

Conditional branch
Conditional branch testing FP bit
Jump&link(JAL),Jump&link register(JALR)
Jump, Jump register

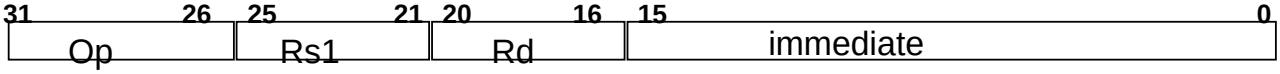
❑ Floating Point

MIPS instruction format

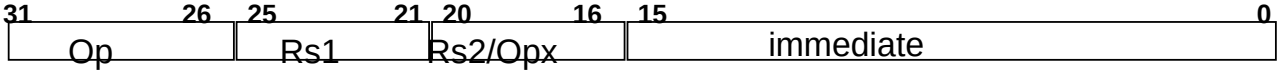
Register-Register



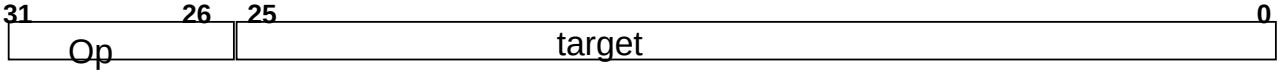
Register-Immediate



Branch



Jump / Call



❑ A "Typical" RISC

- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
- no indirection
- Simple branch conditions
- Delayed branch

When does CPU need to sign extend?

- **When value is sign extended, copy upper bit to full value:**
  Examples of sign extending 8 bits to 16 bits:

  00001010 ⇒ 00000000 00001010
  10001100 ⇒ 11111111 10001100

- **When is an immediate value sign extended?**
  - Arithmetic instructions (add, sub, etc.) sign extend immediates *even for the unsigned versions of the instructions!*
  - Logical instructions *do not sign extend*

- **Load/Store half or byte *do sign extend*, but unsigned versions do not.**

MIPS data transfer instructions

Instruction	*Comment*
SW 500(R4), R3	**Store word**
SH 502(R2), R3	**Store half**
SB 41(R3), R2	**Store byte**
LW R1, 30(R2)	**Load word**
LH R1, 40(R3)	**Load halfword**
LHU R1, 40(R3)	**Load halfword unsigned**
LB R1, 40(R3)	**Load byte**
LBU R1, 40(R3)	**Load byte unsigned**
LUI R1, 40	**Load Upper Immediate (16 bits shifted left by 16)**

Why need LUI?