SISTEMI EMBEDDED

Instruction Set Architecture: RISC/CISC

Addressing Modes

Assembly Language

Federico Baronti

Last version: 20180321

Instructions and Sequencing

- **Instruction Set Architecture (ISA)** can be seen as the specifications of a processor
  - ISA affects processor performances (RISC, CISC, ASIP*)
  - Possible different implementations of the same ISA
- Instructions for a computer must support:
  - data transfers to and from the memory
  - arithmetic and logic operations on data
  - program sequencing and control
  - input/output transfers
- Let's start by introducing some notation

* Application-Specific Instruction set Processor

Register Transfer Notation (1)

- **Register transfer notation (RTN)** is used to describe hardware-level data transfers and operations
- Source/Destination can be either processor registers (e.g. R0, R1,…) or memory locations (usually memory addresses are represented by symbols, such as LOC, VARx, A, B,...)
- [...] to denote the content of a location
- $\leftarrow$ to denote transfer to a destination
- Example: R2 $\leftarrow$ [LOC]
  (transfer from memory location LOC to register R2)

Register Transfer Notation (2)

- RTN can be extended to show arithmetic operations involving locations
- Example: $R4 \leftarrow [R2] + [R3]$
  (add the contents of registers R2 and R3, place the sum in register R4)
- Right-hand expression always denotes a value, left-hand side always names a location (i.e., specifies its address)

Assembly-Language Notation (1)

- RTN shows data transfers and arithmetic (useful to describe the behavior of an instruction)
- Another notation is needed to represent machine instructions & programs using them
- **Assembly language** is used for this purpose
- The two preceding examples using RTN can be related to the assembly-language instructions:

```
Load    R2, LOC
Add     R4, R2, R3
```

Assembly-Language Notation (2)

- **An instruction specifies the requested operation and the operands that are involved**
- We will use English words for the operations (e.g., Load, Store, and Add)
- Commercial processors use **mnemonics**, usually abbreviations (e.g., LD, ST, and ADD)
- **Mnemonics** differ from processor to processor

RISC and CISC Instruction Sets

- Nature of instructions distinguishes computer
- Two fundamentally different approaches
  - Reduced Instruction Set Computers (RISC) have **one-word instructions** and require arithmetic operands to be in registers
  - Complex Instruction Set Computers (CISC) have multi-word instructions and allow operands directly from memory

RISC Instruction Sets (1)

- **Each RISC instruction occupies a single word**
- A load/store architecture is used, meaning:
  - only Load and Store instructions are used to access memory operands
  - operands for arithmetic/logic instructions must be in registers, or one of them can be provided explicitly in the instruction word (*Immediate* operand)
  - Load *proc_register*, *mem_location*
  - Addressing mode specifies actual memory location

RISC Instruction Sets (2)

- Consider high-level language statement:

  int A, B, C ; C = A + B;

- A, B, and C correspond to memory locations
- RTN specification with these symbolic names:

  $$C \leftarrow [A] + [B]$$

- Steps involved: retrieve contents of locations A and B, compute sum, and transfer result to location C

RISC Instruction Sets (3)

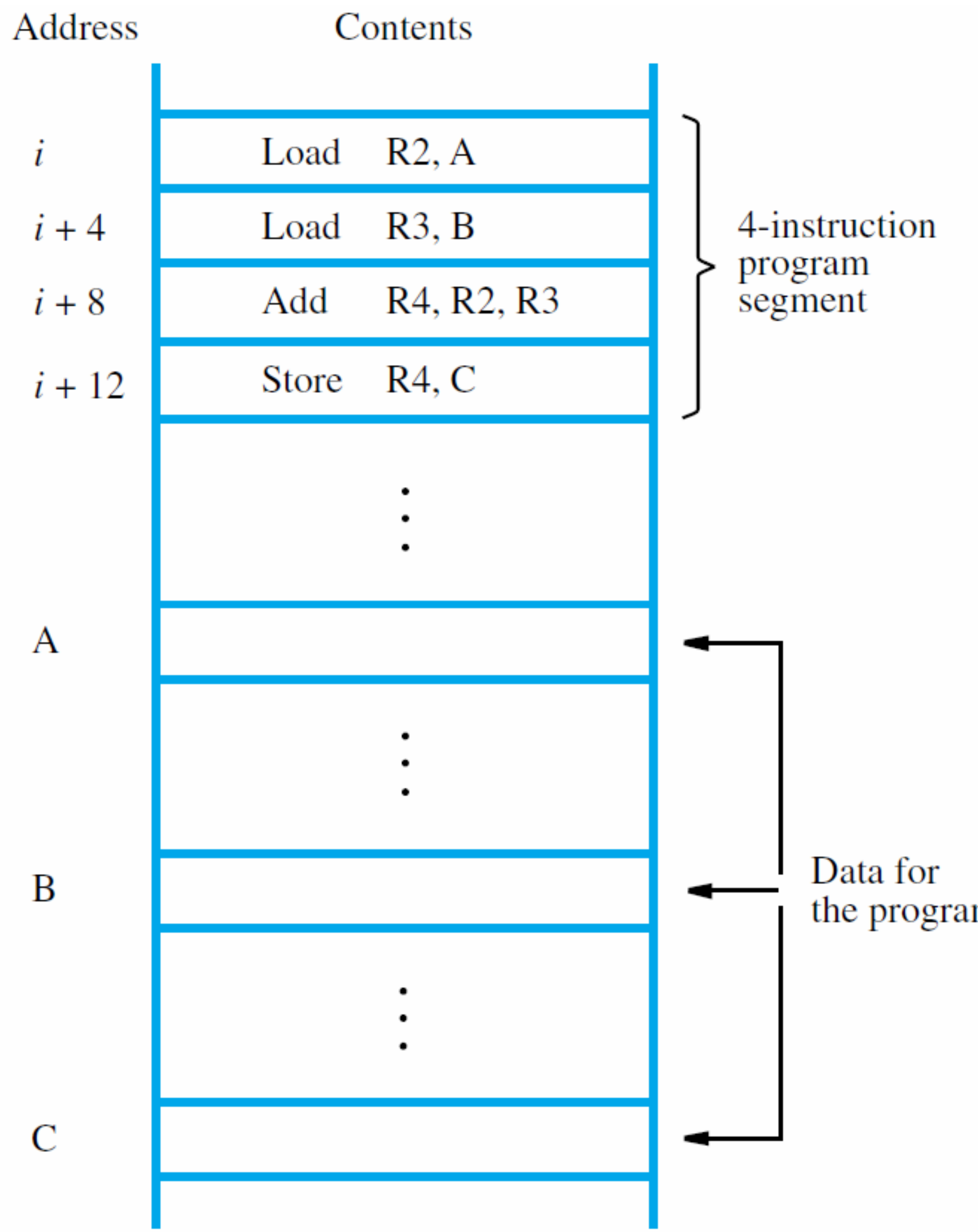
- Sequence of simple RISC instructions for task
  int A, B, C ; C = A + B;

```
Load    R2, A
Load    R3, B
Add     R4, R2, R3          (Add R3, R2, R3)
Store   R4, C               (Store R3, C)
```

- Load instruction transfers data to register
- Store instruction transfers data to the memory. **Store uses the reverse operand order** compared to the Load or Add instructions

Assumptions:

- Memory is 32-bit word length and byte-addressable
- Load/Store instructions have the desired operand address specified **directly**
  - Limitations on usable memory locations for variables because of RISC single word instructions
  - **Need for alternative addressing modes**

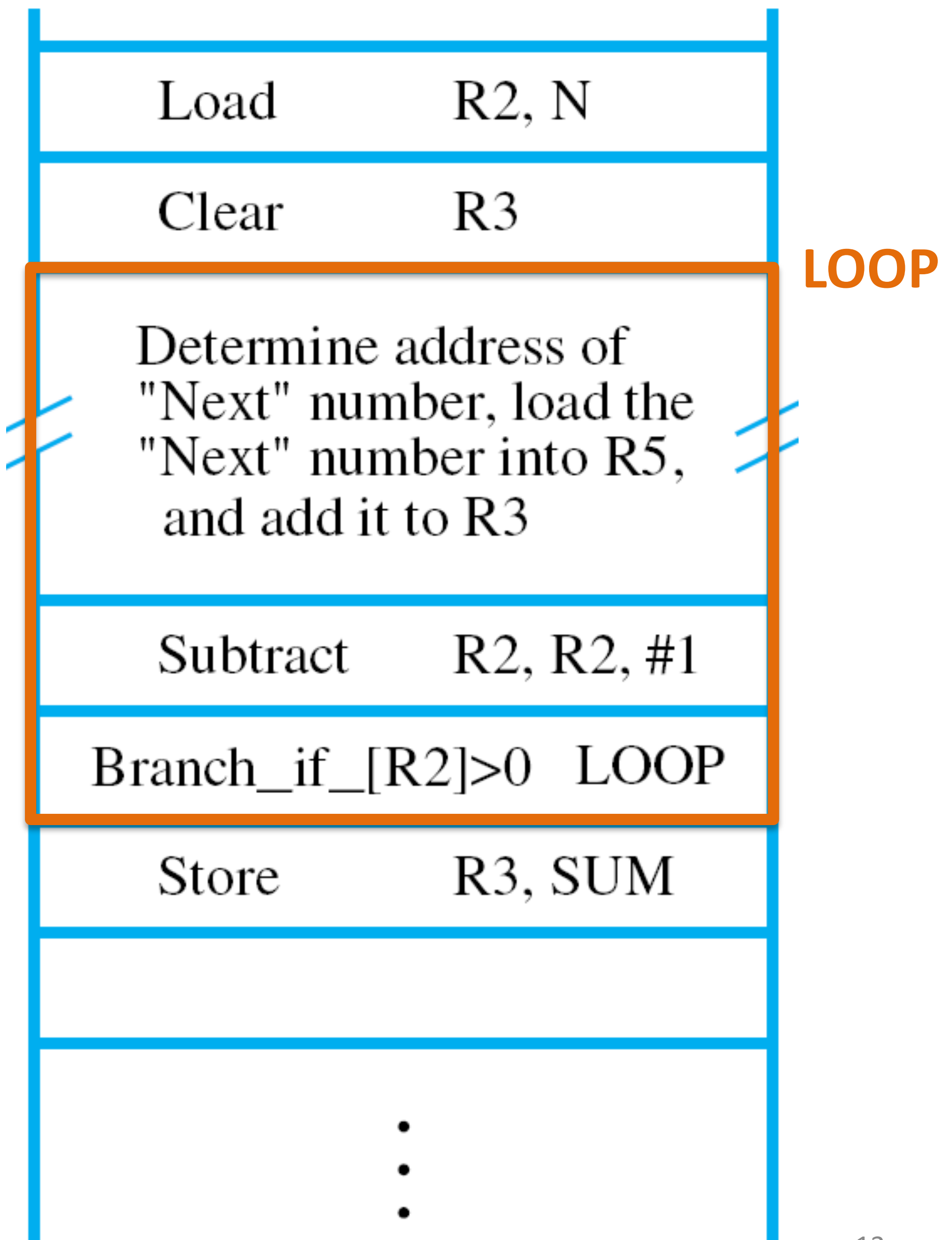


Instruction Execution/Sequencing

- How is the program executed?
- Processor has program counter (PC) register
- Address i of the first instruction placed in PC
- Control circuits fetch and execute instructions, one after another $\rightarrow$ straight-line sequencing
- During execution of each instruction (after fetch), PC register is incremented by 4
- PC content is $i + 16$ after Store is executed

Sum *n* numbers stored in memory at consecutive locations (array) starting at location address NUM1

- *n* is stored at location with address N
- Result needs to be stored at location address SUM
- **Conditional branching**
- **Register indirect addressing mode**



Branching

- Branches that test a condition are used in loops and various other programming tasks
- One way to implement conditional branches is to compare the contents of two registers, e.g.,

  Branch_if_[R4]>[R5]   LOOP
- In generic assembly language with mnemonics the instruction above might actually appear as

  BGT   R4, R5, LOOP

Generating Memory Addresses

- Loop must obtain "Next" number at each loop iteration
- Load instruction cannot contain full address since address size (32 bits) = instruction size
- Also, Load instruction itself would have to be modified in each pass to change address
- Instead, use register R*i* for address location
  - Initialize R*i* to NUM1 and increment it by 4 inside the loop
  - This method works well for accessing the elements of an array

Addressing Modes (1)

- Programs use data structures to organize the information used in computations
- High-level languages enable programmers to describe operations for data structures
- Compiler translates into assembly language
- **Addressing modes** provide compiler with different ways to specify operand locations
- Consider modes used in RISC-style processors

Addressing Modes (2)

RISC-type addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute	LOC	EA = LOC
Register indirect	(Ri)	EA = [Ri]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri,Rj)	EA = [Ri] + [Rj]

EA = effective address

Value = a signed number

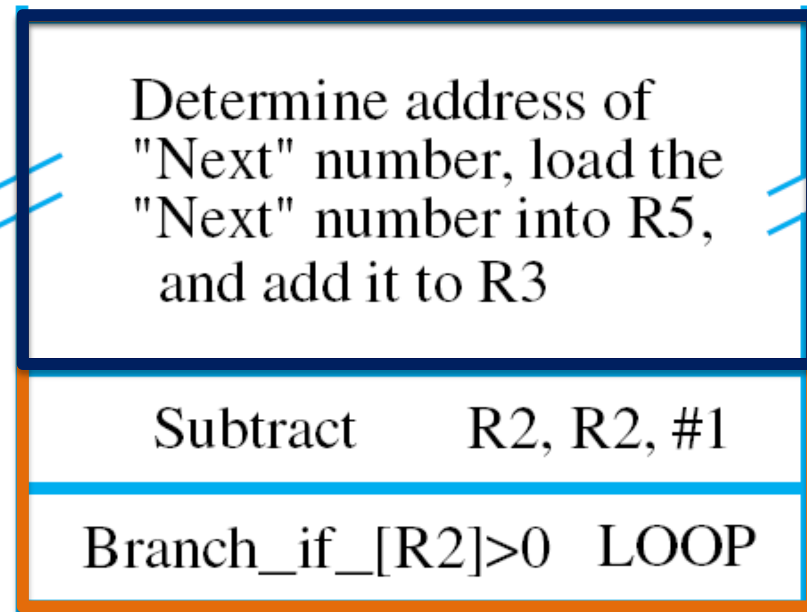
X = index value

Addressing Modes (3)

- RISC-style instructions have a fixed size (single word of 32 bits), hence **immediate**, **absolute** and **index** modes information limited to 16 bits
  - Usually sign-extended (depends on the kind of instruction) to full 32-bit value/address
  - **Absolute** addressing mode is therefore limited to a subset of the full 32-bit address space

Addressing Modes (4)

Sum *n* numbers stored in memory at consecutive locations (array) starting at location address NUM1



	Load	R2, N	Load the size of the list.
	Clear	R3	Initialize sum to 0.
	Move	R4, #NUM1	Get address of the first number.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer to the list.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	Branch back if not finished.
	Store	R3, SUM	Store the final sum.

32-bit Immediate Values

- To construct 32-bit immediates or addresses, use two instructions in sequence:

```
OrHigh   R4, R0, #0x2000
Or       R4, R4, #0x4FF0
```

- R0 always contains 0
- Result is NUM1=0x20004FF0 in register R4
- Useful pseudoinstruction for above sequence:
  Move(ImmediateAddress)   R4, #NUM1
  - Assembler substitute the pseudoinstruction with OrHigh & Or and appropriate 16-bit values for each instruction

Assembly Language

- Mnemonics (LD/ADD instead of Load/Add) used when programming specific computers
- The mnemonics represent the OP codes
- Assembly language is the set of mnemonics and rules for using them to write programs
- The rules constitute the language syntax
- Example: suffix 'I' to specify immediate mode

  ADDI   R2, R3, 5   (instead of #5)

Assembler Directives

- Other information also needed to translate source program to object program
- How should symbolic names be interpreted?
- Where should instructions/data be placed?
- Assembler directives provide this information
- ORIGIN defines instruction/data start position
- RESERVE and DATAWORD define data storage
- EQU associates a name with a constant value

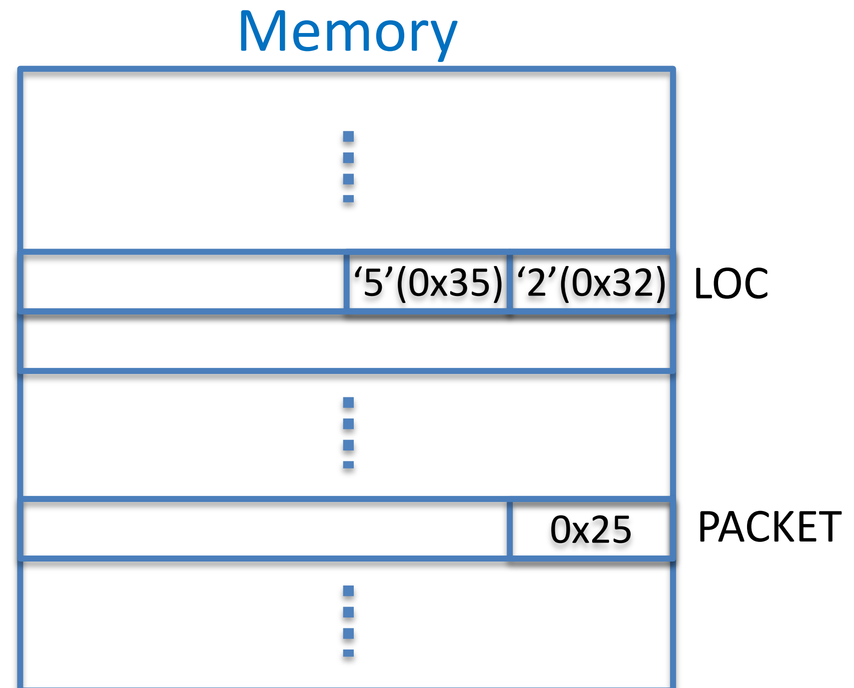
	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Program Assembly & Execution

- From source program, assembler generates machine-language object program
- Assembler uses ORIGIN and other directives to determine address locations for code/data
- For branches, assembler computes ±offset from present address (in PC) to branch target
- Loader places object program in memory
- Debugger can be used to trace execution

Example Program: Digit Packing

- Memory contains two ASCII decimal digits starting at address LOC. We want to extract the BCD of the two decimal digits and "pack" them to a byte to be stored at memory location PACKET



Move is a pseudo instr. The assembler replaces it with

OrHigh	R2, R0, $\#LOC_{31\text{-}16}$
Or	R2, R2, $\#LOC_{15\text{-}0}$

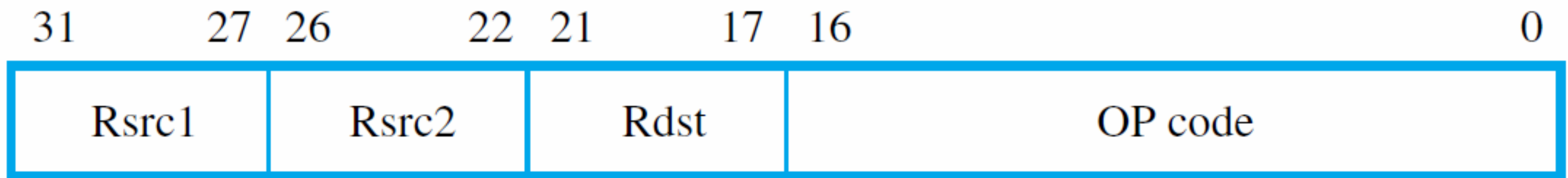
Move	R2, #LOC	R2 points to data.
LoadByte	R3, (R2)	Load first byte into R3.
LShiftL	R3, R3, #4	Shift left by 4 bit positions.
Add	R2, R2, #1	Increment the pointer.
LoadByte	R4, (R2)	Load second byte into R4.
And	R4, R4, #0xF	Clear high-order bits to zero.
Or	R3, R3, R4	Concatenate the BCD digits.
StoreByte	R3, PACKED	Store the result.

Using the *index* (indirect with displacement) address mode, these 2 instr. can be replaced with

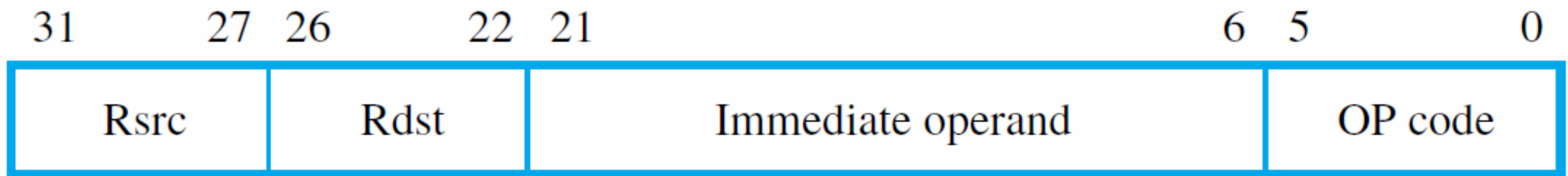
LoadByte	R4, 1(R2)

Encoding of Machine Instructions

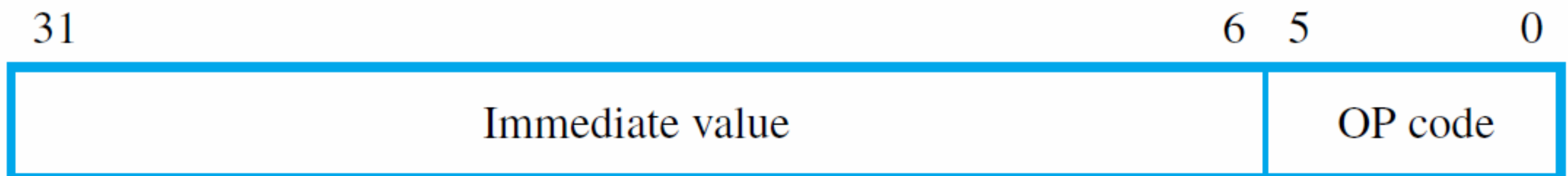
- Assembly-language instructions express the actions to be performed by processor circuitry
- Assembler converts to machine instructions
- Three-operand RISC instructions require enough bits in single word to identify registers
- 16-bit immediates must be supported
- Instruction must include bits for OP code
- Call instruction also needs bits for address



(a) Register-operand format



(b) Immediate-operand format



(c) Call format

RISC Summary

- Single word instructions
- Operands of arithmetic and logic operations in REGISTERS only
- Load/store architecture (no memory to memory transfers)
- Simple addressing modes

CISC Instruction Sets (1)

- **Not constrained to load/store architecture**
- **Instructions may be larger than one word**
- Typically use two-operand instruction format, with at least one operand in a register
- Move instruction equivalent to Load/Store, but can also transfer immediate values
  and possibly operate between two memory locations
- Arithmetic instructions may employ addressing modes for operands in memory:
  Subtract LOC, R*i*
  Add R*j*, 16(R*k*)

CISC Instruction Sets (2)

- Implementation of $C = A + B$ using CISC:

Move	C, A
Add	C, B

In some CISC ISA only **one** operand can be in memory

Move	R*i*, A
Add	R*i*, B
Move	C, R*i*

Additional Addressing Modes (1)

- CISC style has other modes not usual for RISC
- ***Autoincrement*** mode: effective address given by register contents; after accessing operand, register contents incremented to point to next
- Useful for adjusting pointers in loop body:

```
Add         SUM, (Ri)+
MoveByte    (Rj)+, Rk
```

- Increment by 4 for words, and by 1 for bytes
  - R*i* is incremented by 4 and R*j* by 1

Additional Addressing Modes (2)

- ***Autodecrement*** mode: before accessing operand, register contents are decremented, then new contents provide effective address
- Notation in assembly language:

  Add   R*j*, –(R*i*)
- Use autoinc. & autodec. for stack operations:

  Move   –(SP), NEWITEM   (push)

  Move   ITEM, (SP)+   (pop)
- SP is the Stack Pointer REGISTER, NEWITEM and ITEM are two generic REGISTERS

Condition Codes

- Processor can maintain information on results to affect subsequent conditional branches
- Results from arithmetic/logic (& Move)
- Condition code flags in a status register:

N	(negative)	1 if result negative, else 0
Z	(zero)	1 if result zero, else 0
V	(overflow)	1 if overflow occurs, else 0
C	(carry)	1 if carry-out occurs, else 0

Branches using Condition Codes

- CISC branches check condition code flags
- For example, decrementing a register causes N and Z flags to be cleared if result is $>$ zero
- A branch to check logic condition $N + Z = 0$:

  Branch>0   LOOP

- Other branches test conditions for $<$, $=$, $\neq$, $\leq$, $\geq$
- Also Branch_if_overflow and Branch_if_carry
- Consider CISC-style array-summing program

```
        Move         R2, N         Load the size of the list.
        Clear        R3            Initialize sum to 0.
        Move         R4, #NUM1     Load address of the first number.
LOOP:   Add          R3, (R4)+     Add the next number to sum.
        Subtract     R2, #1        Decrement the counter.
        Branch>0     LOOP          Loop back if not finished.
        Move         SUM, R3       Store the final sum.
```

RISC vs CISC Summary

- Single word instructions
- Operands of arithmetic and logic operations in REGISTERS only
- Load/store architecture (no memory to memory transfers)
- Simple addressing modes
- Faster instruction execution
- Larger size programs

- Instructions may span multiple words
- Operands of arithmetic and logic operations may be in memory
- Move instructions wider scope than load/store
- More powerful addressing modes
- Smaller size programs
- Slower instruction execution

RISC reduces hardware complexity at the expense of the software complexity. Need for sophisticated compilers.

References

- C. Hamacher, Z. Vranesic, S. Zaky, N. Manjikian "Computer Organization and Embedded Systems," *McGraw-Hill International Edition*
  - Cap. II all except sections 2.6 and 2.7