BCS-29
Advanced Computer Architecture

Instruction Set Architectures

RISC Processors

RISC vs CISC

Computer Architecture

- The term architecture is used here to describe the attribute of a system as seen by the programmer.
- It is the conceptual structure and functional behavior as distinct from the organization of the data-flow and control-flow, the logic design, and the physical implementation.
- Instruction set architecture: program-visible instruction set
  - Instruction format, memory addressing modes, architectural registers, endian type, alignment, …
  - EX: RISC, CISC, VLIW, EPIC

Instruction Set Architecture

- Elements of ISA
  - Programming Registers
  - Type and Size of Operands
  - Addressing Modes
  - Types of Operations
  - Instruction Encoding

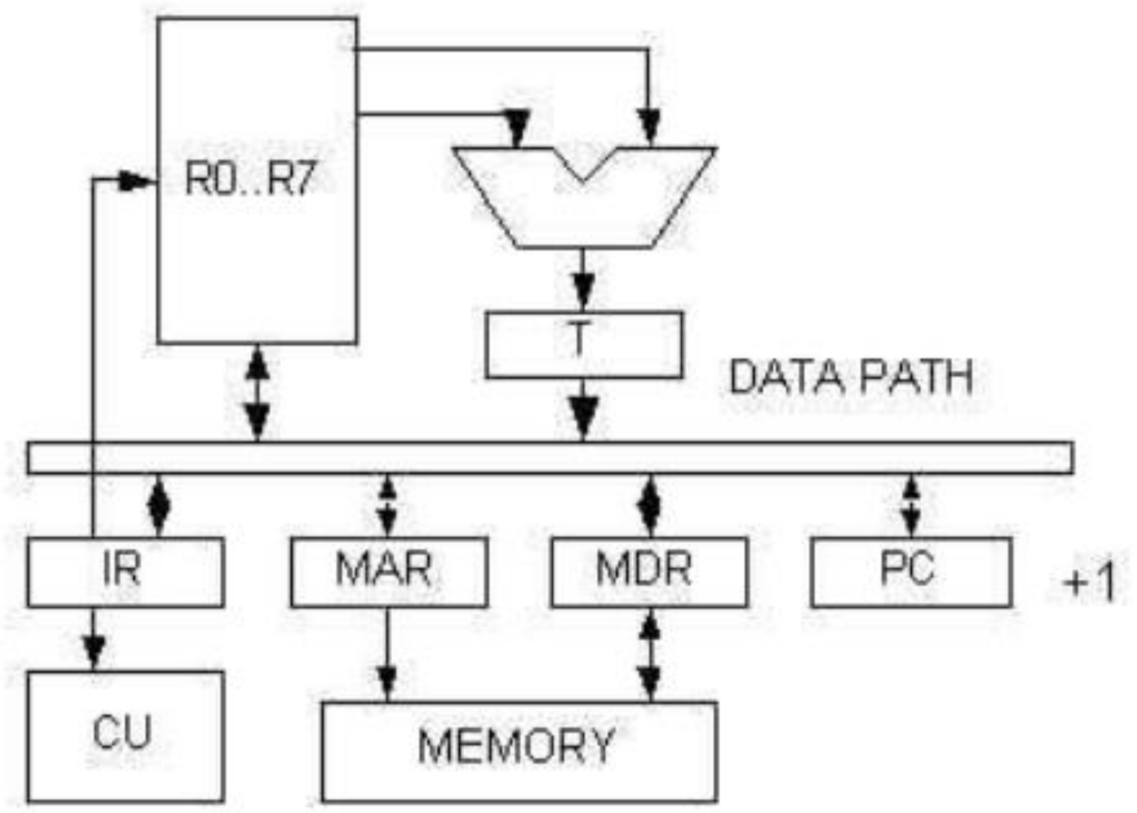
Instruction Set Architecture

- Instruction set architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.
- The instruction set architecture is also the machine description that a hardware designer must understand to design a correct implementation of the computer.

Components of an ISA

- Sometimes known as *The Programmer's Model* of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general-purpose cells of same size in memory
  - Storage associated with I/O devices
- The machine instruction set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
  - i.e., register transfers
- The instruction format
  - Size and meaning of fields within the instruction
- The nature of the fetch-execute cycle
  - Things that are done before the operation code is known

A Basic Model of the machine



An Instruction

- Instruction **add r0, r1, r2**
- Operation to be perform **add** r0, r1, r2
  - Ans: Op code: add, load, branch, etc.
- Where to find the operands: add r0, **r1, r2**
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result add **r0,** r1, r2
  - Again CPU register or memory cell
- Location of next instruction

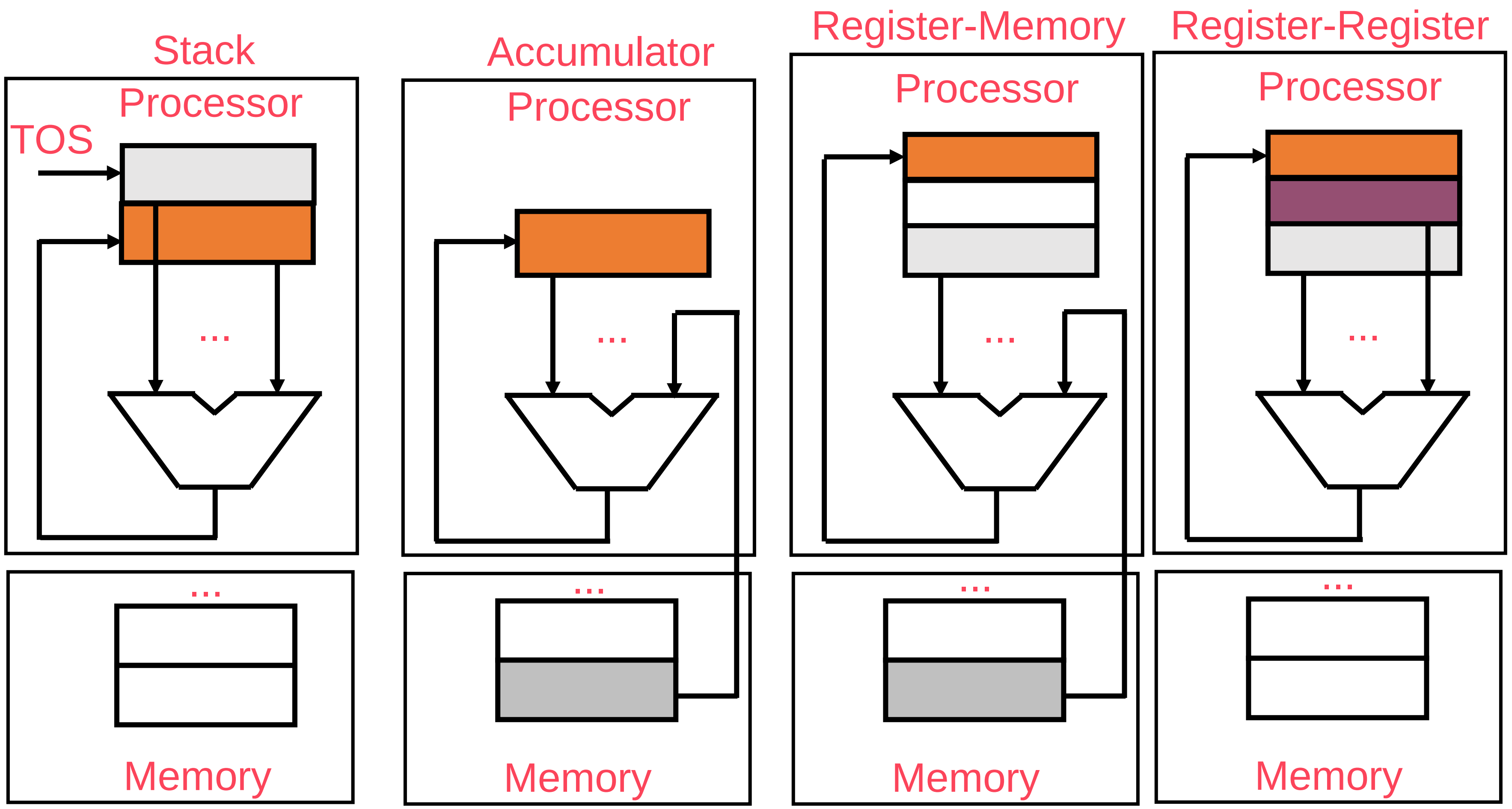
$$\begin{tikzcd}
\text{add r0, r1, r3} \\
\text{br endloop} \arrow[u, bend right]
\end{tikzcd}$$

  - Almost always memory cell pointed to by program counter(PC) or Instruction pointer (IP)

Instructions May be Divided into 3 Classes

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - *Load*—source is memory and destination is register
  - *Store*—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Change the form of one or more operands to produce a result stored in another location
  - *Add, Sub, Shift*, etc.
- Branch instructions (control flow instructions)
  - Alter the normal flow of control from executing the next instruction in sequence
  - *Br Loc, Brz Loc2*,—unconditional or conditional branches

Classifying ISA



Example: X = (A+B)*(C+D)

- **Zero Address Instructions:**
  - A stack based computer do not use address field in instruction. To evaluate an expression first it is converted Post fix Notation.

PUSH	A	TOP = A
PUSH	B	TOP = B
ADD		TOP = A+B
PUSH	C	TOP = C
PUSH	D	TOP = D
ADD		TOP = C+D
MUL		TOP = (C+D)*(A+B)
POP	X	M[X] = TOP

Example: X = (A+B)(C+D)*

- **One Address Instructions**
  - There is an implied ACCUMULATOR register for data manipulation. One operand is in accumulator and other is in register or memory location.
  - The ALU always consider one operand from the ACCUMULATOR register and route the result into the ACCUMULATOR register.

Instruction Format:

opcode	Operand/address of operand

LOAD	A	AC = M[A]
ADD	B	AC = AC + M[B]
STORE	T	M[T] = AC
LOAD	C	AC = M[C]
ADD	D	AC = AC + M[D]
MUL	T	AC = AC * M[T]
STORE	X	M[X] = AC

Example: X = (A+B)*(C+D)

- **Two Address Instructions:**
- This is common in commercial computers. Here two address can be specified in the instruction. Unlike earlier in one address instruction the result was stored in accumulator here result can be stored at different location rather than just accumulator but require more number of bit to represent address.

Instruction Format:

opcode	Destination Address	Source Address

```
MOV     R1, A       R1 = M[A]
ADD     R1, B       R1 = R1 + M[B]
MOV     R2, C       R2 = C
ADD     R2, D       R2 = R2 + D
MUL     R1, R2      R1 = R1 * R2
MOV     X, R1       M[X] = R1
```

Example: X = (A+B)(C+D)*

- **Three Address Instruction:**
  - This has three address field to specify a register or a memory location. Program created are much short in size but number of bits per instruction increase. These instructions make creation of program much easier but it does not mean that program will run much faster because now instruction only contain more information, but each micro-operation will be performed in one cycle only.
- Instruction Format:

opcode	Destination Address	Source Address	Source Address

ADD	R1, A, B	R1 = M[A] + M[B]
ADD	R2, C, D	R2 = M[C] + M[D]
MUL	X, R1, R2	M[X] = R1 * R2

Classifying ISA (Instructions)

- Stack Architectures -
  operands are implicitly on the top of the stack

0 **address**	**add**	**tos <- tos + next**

- Accumulator Architectures -
  one operand is implicitly accumulator

1 **address**	**add A**	**acc <- acc + mem[A]**

- General-Purpose Register Architectures -
  only explicit operands, either registers or memory locations
  - Memory-Memory :
    access memory Locations as part of any instruction

2 **address**	**add A, B**	**mem[A] <- mem[A] + mem[B]**
3 **address**	**add A, B, C**	**mem[A] <- mem[B] + mem[C]**

Classifying ISA (Instructions)

- General-Purpose Register Architectures - only explicit operands, either registers or memory locations
  - register-memory: access memory as part of any instruction

2 **address**	**add R1, A**	**R1 <- R1 + mem[A]**
	load R1, A	**R1 <_ mem[A]**

  - register-register: access memory only with load and store instructions

3 **address**	**add R1, R2, R3**	**R1 <- R2 + R3**
	load R1, R2	**R1 <- mem[R2]**
	store R1, R2	**mem[R1] <- R2**

Operand Access

- Register-Register (0,3)

  (m, n) means m memory operands and n total operands in an ALU instruction

  - Pure RISC, register to register operations
  - Advantages
    - Simple, fixed length instruction encoding.
    - Simple code generation.
    - Instructions take similar number of clocks to execute. Uniform CPI
  - Disadvantages
    - Higher inst. count.
    - Some instructions are short and bit encoding may be wasteful.

Operand Access

- Register-Memory (1,2)
  - Register – Memory ALU Architecture
  - In later evolutions of RISC and CISC
  - Advantages
    - Data can be accessed without loading first.
    - Instruction format easy to encode
    - Good instruction density
  - Disadvantages
    - Source operand also destination, data overwritten
    - Need for memory address field may limit # registers
    - CPI varies by operand location

Operand Access

- Memory-Memory (3,3)
  - True memory-memory ALU model, e.g. full orthogonal CISC architecture
  - Advantages
    - Most compact instruction density, no temporary registers needed
  - Disadvantages
    - Memory access create bottleneck
    - Variable CPI
    - Large variation in instruction size
    - Expensive to implement
  - Not used in today's architectures

Memory Addressing

- What is accessed - byte, word, multiple words?
  - today's machine are byte addressable, due to legacy issues
- But main memory is organized in 32 - 64 byte lines
  - matches cache model
  - Retrieve data in, say, 4 byte chunks
- Alignment Problem
  - accessing data that is not aligned on one of these boundaries will require multiple references
    - E.g. fetching 16 bit integer at byte offset 3 requires two four byte chunks to be read in (line 0, line 1)
  - Can make it tricky to accurately predict execution time with mis-aligned data
  - Compiler should try to align! Some instructions auto-align too

Addressing Modes

- The addressing mode specifies the address of an operand we want to access
  - Register or Location in Memory
  - The actual memory address we access is called the **effective address**
- Effective address may go to memory or a register array
  - typically dependent on location in the instruction field
  - multiple fields may combine to form a memory address
  - register addresses are usually simple - needs to be fast
- Effective address generation is important and should be fast!
  - Falls into the common case of frequently executed instructions

Memory Addressing

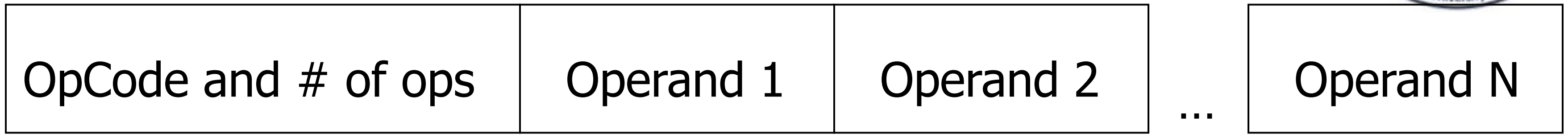
Mode	Example	Meaning	When used
Register	Add R4, R3	Regs[R4]←Regs[R4] + Regs[R3]	Value is in a register
Immediate	Add R4, #3	Regs[R4] ← Regs[R4] + 3	For constants
Displacement	Add R4, 100(R1)	Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]	Access local variables
Indirect	Add R4, (R1)	Regs[R4]←Regs[R4] + Mem[Regs[R1]]	Pointers
Indexed	Add R3, (R1+R2)	Regs[R3]←Mem[Regs [R1] + Regs[R2]]	Traverse an array
Direct	Add R1, $1001	Regs[R1] ← Regs[R1] + Mem[1001]	Static data, address constant may be large

Memory Addressing

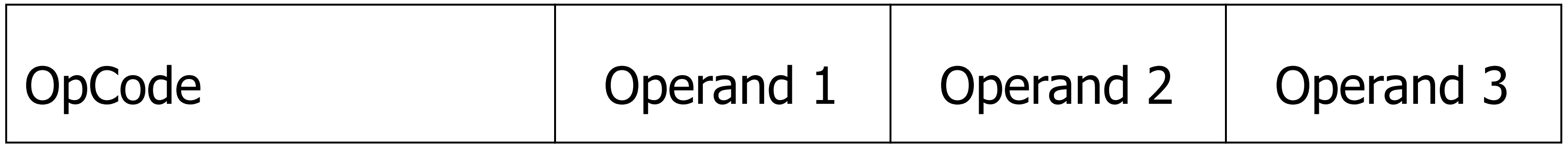
Mode	Example	Meaning	When used
Memory Indirect	Add R1, @(R3)	Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]]	*p if R3=p
Autoinc	Add R1, (R2)+	Regs[R1]←Regs[R1]+ Mem[Regs[R2]], Regs[R2]←Regs[R2]+1	Stepping through arrays in a loop
Autodec	Add R1, (R2)-	Regs[R1]←Regs[R1]+ Mem[Regs[R2]], Regs[R2]←Regs[R2]-1	Same as above. Can push/pop for a stack
Scaled	Add R1, 100(R2)[R3]	Regs[R1]← Regs[R1]+ Mem[100+Regs[R2] + Regs[R3] * d]	Index arrays by a scaling factor, e.g. word offsets

Instruction Set Encoding Options

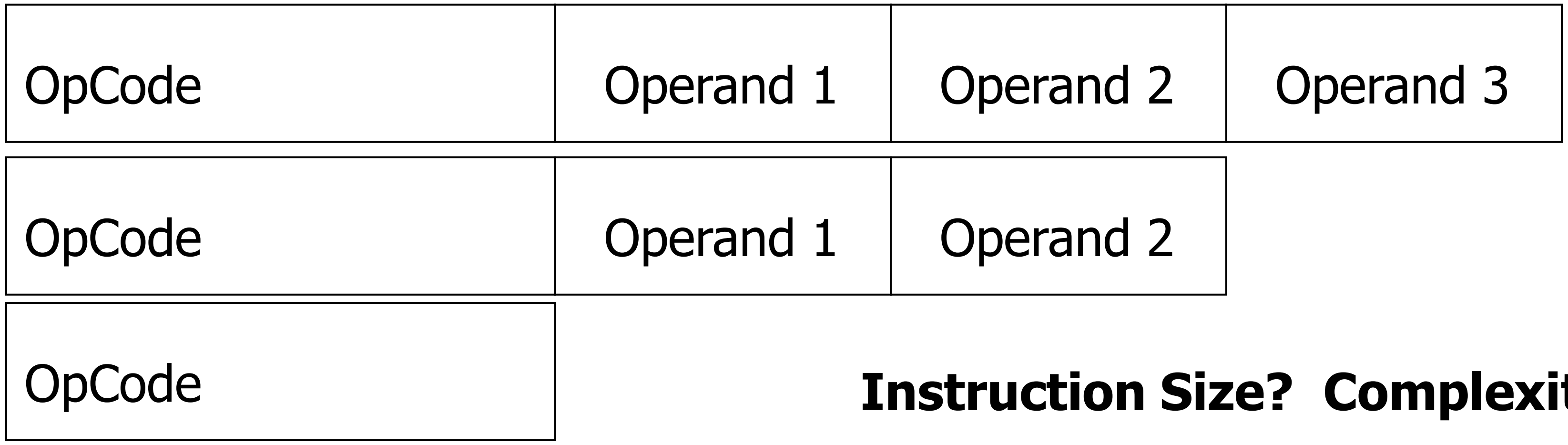
Variable (e.g. VAX)



Fixed (e.g. DLX, SPARC, PowerPC)



Hybrid (e.g. x86, IBM 360)

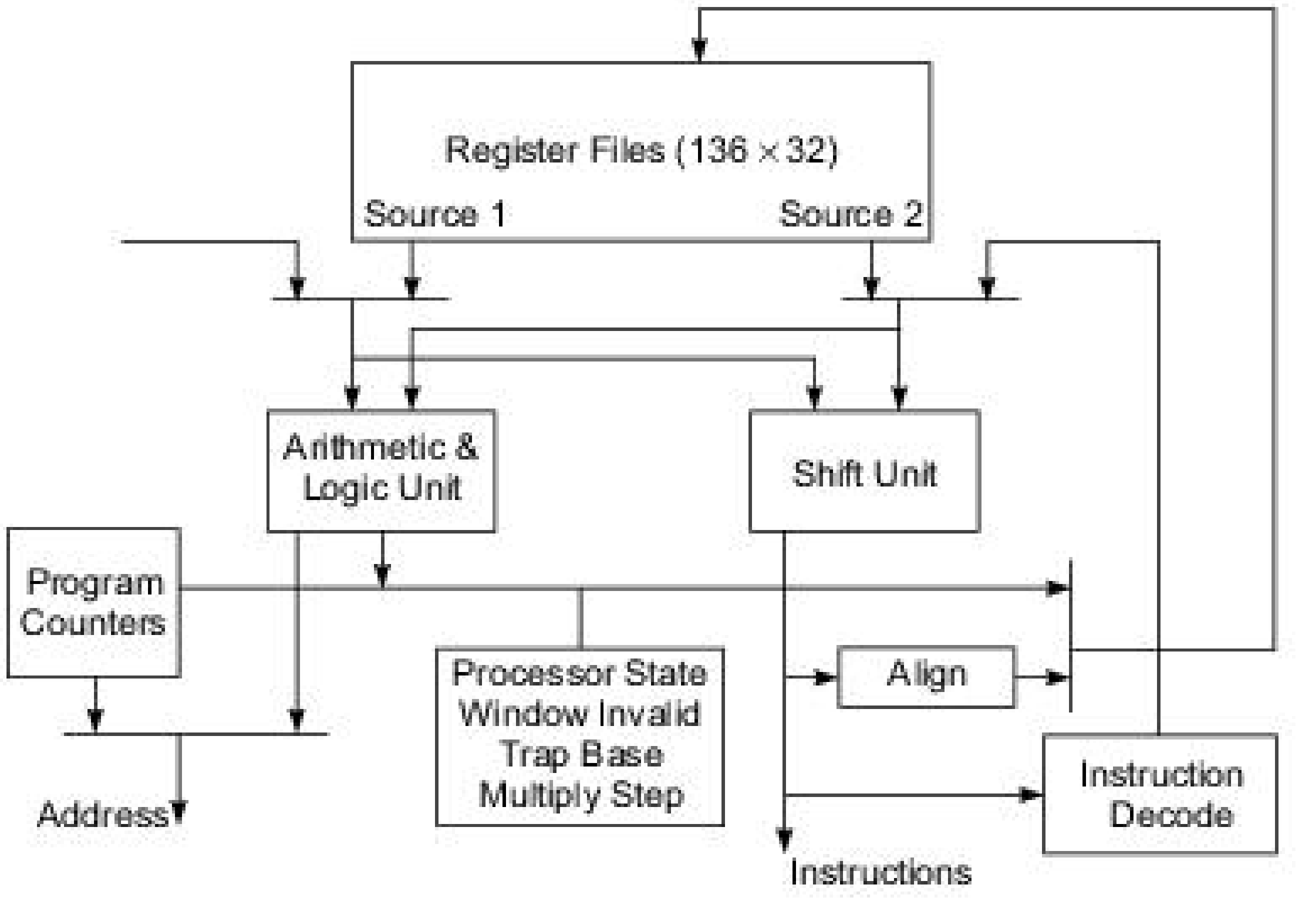


Instruction Size? Complexity?

Reduced Instruction Set Computer (RISC)

- RISC architectures represent an important innovation in the area of computer organization.
- The RISC architecture is an attempt to produce more CPU power by simplifying the instruction set of the CPU.
- The opposed trend to RISC is that of complex instruction set computers (CISC).
- Both RISCs and CISCs try to solve the same problem. CISCs are going the traditional way of implementing more and more complex instructions. RISCs try to simplify the instruction set.
- Innovations in RISC architectures are based on a close analysis of a large set of widely used programs.
- One of the main concerns of RISC designers was to maximize the efficiency of pipelining.
- Present architectures often include both RISC and CISC features.
- **Both RISC and CISC architectures have been developed as an attempt to cover the semantic gap.**

Typical RISC Processor Architecture



Main Characteristics of RISC Architectures:

- The instruction set is limited and includes only simple instructions.
- Only LOAD and STORE instructions reference data in memory.
- Instructions use only few addressing modes.
- Instructions are of fixed length and uniform format.
- A large number of registers is available.

Main Characteristics of RISC

A Small number of Simple Instructions:

- Simple and small decode and execution hardware is required.
- A hard-wired controller is needed, rather than using microprogramming.
- CPU takes less silicon area to implement, and run faster also.
- Execution of one machine instruction per clock cycle.
- Register-to-register operation.
- Simple addressing mode.
- Simple instruction format.

Main Characteristics of RISC

Try to achieve one instruction per clock

- Machine cycle is defined to be the time it takes two operands to fetch from registers, perform an ALU operation and store the result in a register.
- The instruction pipeline performs more efficiently due to simple instructions and simpler execution patterns.
- Complex operations are executed as a sequence of simple instructions. In the case of CISC, these operations are executed as one single or few complex instructions.

Example

An illustrative example with the following assumption:

- A program with 80% of executed instructions being simple and 20% complex.
- CISC: simple instructions takes 4 cycles, complex instructions take 8 cycles; cycle time is 100 ns.
- RISC: simple instructions are executed in one cycle; complex instructions are implemented as a sequence of instruction(let 14 instructions on average); cycle time is 75 ns.

How much time takes a program of 1000000 instructions:

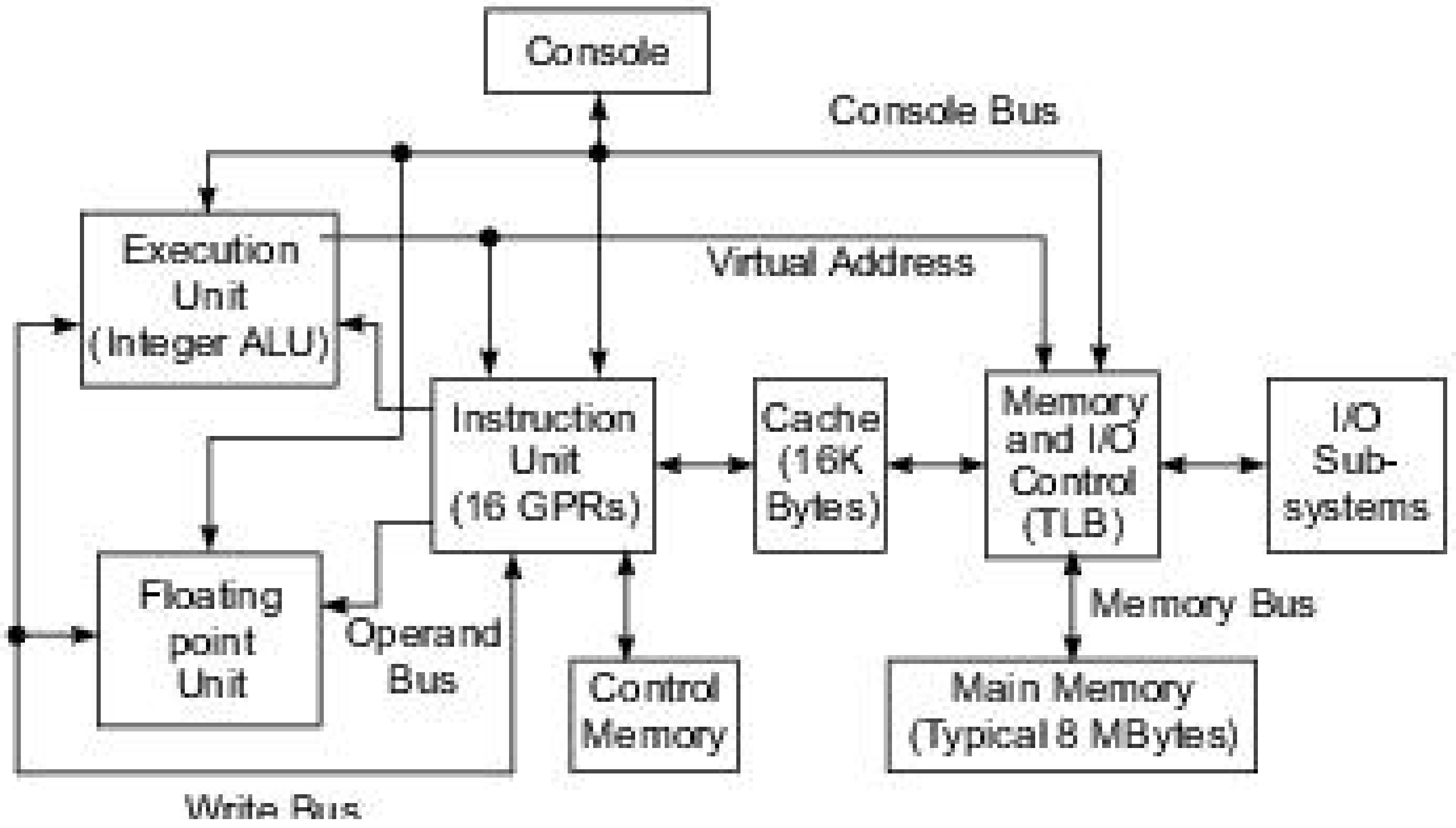
CISC: $(10^6 \text{ X } 0.80 \text{ X } 4 + 10^6 \text{ X } 0.20 \text{ X } 8) \text{ X } 10^{-7} = 0.48$ ns

CISC: $(10^6 \text{ X } 0.80 \text{ X } 1 + 10^6 \text{ X } 0.20 \text{ X } 14) \text{ X } .75 \text{ X } 10^{-7} = 0.48$ ns

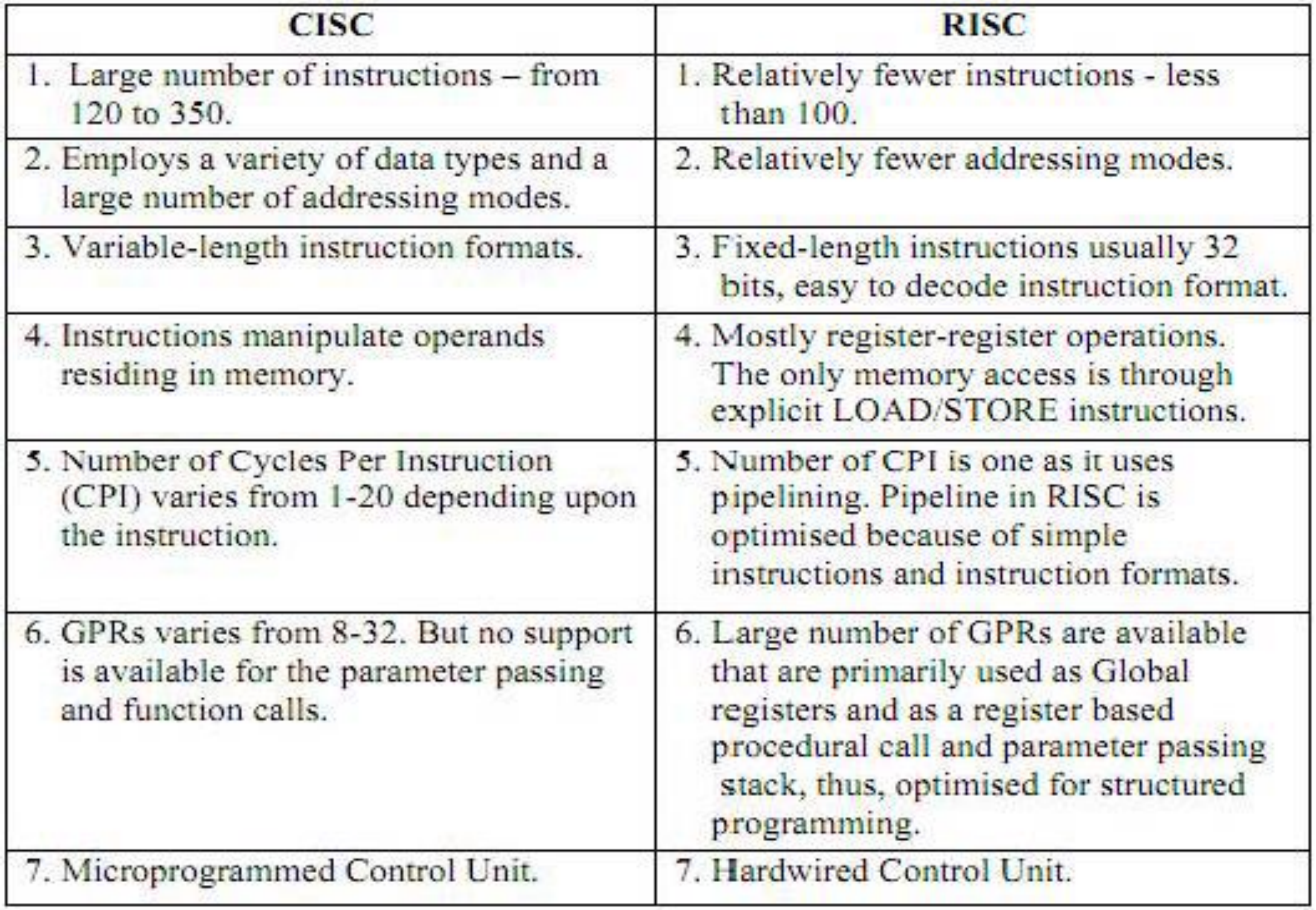
Main features of CISC

- A large number of instructions (> 200) and complex instructions and data types.
- Many and complex addressing modes.
- Direct hardware implementations of high-level language statements.
- Microprogramming techniques are used so that complicated instructions can be implemented.
- Memory bottleneck is a major problem, due to complex addressing modes and multiple memory accesses per instruction.

Typical CISC Processor Architecture



An overview



CISC	RISC
1. Large number of instructions – from 120 to 350.	1. Relatively fewer instructions - less than 100.
2. Employs a variety of data types and a large number of addressing modes.	2. Relatively fewer addressing modes.
3. Variable-length instruction formats.	3. Fixed-length instructions usually 32 bits, easy to decode instruction format.
4. Instructions manipulate operands residing in memory.	4. Mostly register-register operations. The only memory access is through explicit LOAD/STORE instructions.
5. Number of Cycles Per Instruction (CPI) varies from 1-20 depending upon the instruction.	5. Number of CPI is one as it uses pipelining. Pipeline in RISC is optimised because of simple instructions and instruction formats.
6. GPRs varies from 8-32. But no support is available for the parameter passing and function calls.	6. Large number of GPRs are available that are primarily used as Global registers and as a register based procedural call and parameter passing stack, thus, optimised for structured programming.
7. Microprogrammed Control Unit.	7. Hardwired Control Unit.