CISC / RISC

Complex / Reduced

Instruction Set Computers

Instruction Usage

	Instruction Group	Average Usage
1	Data Movement	45.28%
2	Flow Control	28.73%
3	Arithmetic	10.75%
4	Compare	5.92%
5	Logical	3.91%
6	Shift	2.93%
7	Bit Manipulation	2.04%
8	I/O & Others	0.44%

Programming Observations

- 56% of constants ± 15 (5 bits)
- 98% of constants ± 511 (10 bits)
- 95% of subroutines require less than 6 parameters (arguments)

Instruction Usage and Programming Observations lead to a smaller, less complex, instruction set

RISC / CISC Comparison (1/2)

	Characteristics	RISC	CISC
1	On chip registers	Many ($>$30)	Few (2–16)
2	Registers per instruction	Three `ADD R1, R2, R3` $[R1] \leftarrow [R2] + [R3]$	Two `ADD R1, R2` $[R2] \leftarrow [R1] + [R2]$
3	Parameter Passing	Efficient on-chip registers	Inefficient off-chip memory
4	Flow Control Instructions	Optimised 20–30% of program	Not-Optimised

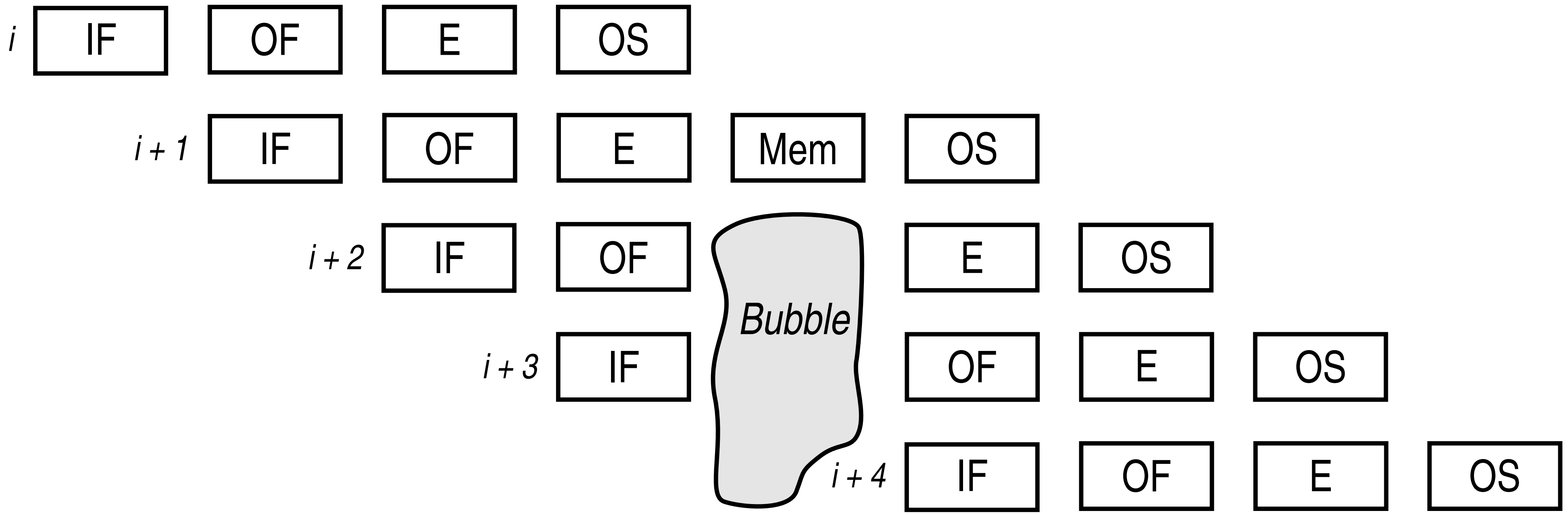
RISC / CISC Comparison (2/2)

	Characteristics	RISC	CISC
5	Operations Per Clock Cycle	One Instruction	One microcode (RTL) Instruction
6	Less used Instructions	Not Implemented	Full Implementation
7	Microcode	Not Implemented	All Instructions
8	Instruction format	Few (4 or 5) fixed length (32-bit)	Many (18) variable lengths (8–80 bits)

Pipelining

- Machine cycle:
  1. Fetch instruction into IR
  2. Decode op-code field
  3. Fetch Operands
  4. Execute operation
  5. Store Operands
- Different part of circuit for each step
- Run all steps in parallel
  (Fetch and decode next instruction at the same time
  as fetching the operands for current instruction. . .)
- An “air bubble” may occur
  (The pipeline is interrupted and a ‘do nothing’ step
  is introduced)

Pipeline “Bubble”



Bubbles: Branch Delay

Pipeline has already read the next instruction before reaching the execute phase of the branch instruction, thus we have to throw away the next instruction, causing a pipeline bubble.

- **Delay Jump** (aka **Delay Slot**)

  Always execute the instruction after the Branch

- **Branch Prediction**

  Cache both next and target instructions, so next instruction to be executed is on-chip and ready to load into the pipeline, loosing one clock-cycle, but splitting the cache.

- **Conditional Execution**

  Remove the need for most branch instructions by allowing instructions to be executed conditionally, wasting one clock-cycle for each non-executed instruction.

Bubbles: Dependency Delay

Next instruction relies on result of the current instruction

Line 1: x = a + b
Line 2: y = x + 2

Line 2 must wait for the Operand Store phase of line 1 to complete before it can perform it's Operand Fetch phase, causing a bubble, otherwise x will have the wrong value.

- **Internal forwarding** (*aka* **Instruction Scheduling**)

  Place another non-dependent instruction in between the two dependent instructions.

- **Load Delay**

  Allow the CPU to delay by one clock-cycle when a source register for the current instruction is the same as a destination register for the previous instruction, thus skipping over the bubble (allowing the bubble to occur).

Bubbles: Memory Access

There are over 20 CPU clock-cycles per memory clock-cycle. When accessing memory the CPU must slow down to the same cycle rate as the memory, causing a very large bubble.

- **Buffer Memory Access** (*aka* **Cache**)

  Use on-chip memory (cache) and a Memory Management Unit (MMU) to access off-chip memory while the CPU is executing at full speed.

  MMU attempts to predict memory access

  CPU will have to slow down when accessing memory not in cache (a cache *miss*).

- Produce program code in such a way as to reduce the number of external memory accesses required.

Cache Memory

- Frequently accessed memory (main or disk) is copied into cache memory
- Speeds up memory access
  Memory taken from on-chip cache (fast)
  rather than external off-chip memory (slow)
- Update Policy
  - ⇒ **Write Delayed** – Volatile
    Memory writes are stored in the cache
    Periodically write cache to external memory
  - ⇒ **Write Through** – Non-Volatile
    Write modifies external memory and cache
    Slow but always up to date

The Post RISC era

Intel	IA32	– CISC with 2 stage pipeline (*Pentium*)
	IA64	– RISC was Hewlett-Packard *Pyramid*
	XScale	– RISC was ARM’s StrongARM2
AMD	Athlon	– RISC based Pentium
	Opteron	– 64-Bit RISC with Athlon subset
Motorola		
	PowerPC	– RISC (PowerPC / PowerMac / . . .)
	DragonBall	– CISC (Early Palm’s and Mobile ’Phones)

ARM Advanced RISC Machines Ltd.
Has a nine stage pipeline
Low power, used in embedded systems
Palm Computing, G2.5 and G3 Mobile ’Phones, *etc*

SPARC Scalable Processor ARChitecture
A workstation level RISC processor
Developed by Sun, Texas Instruments and Fujitsu