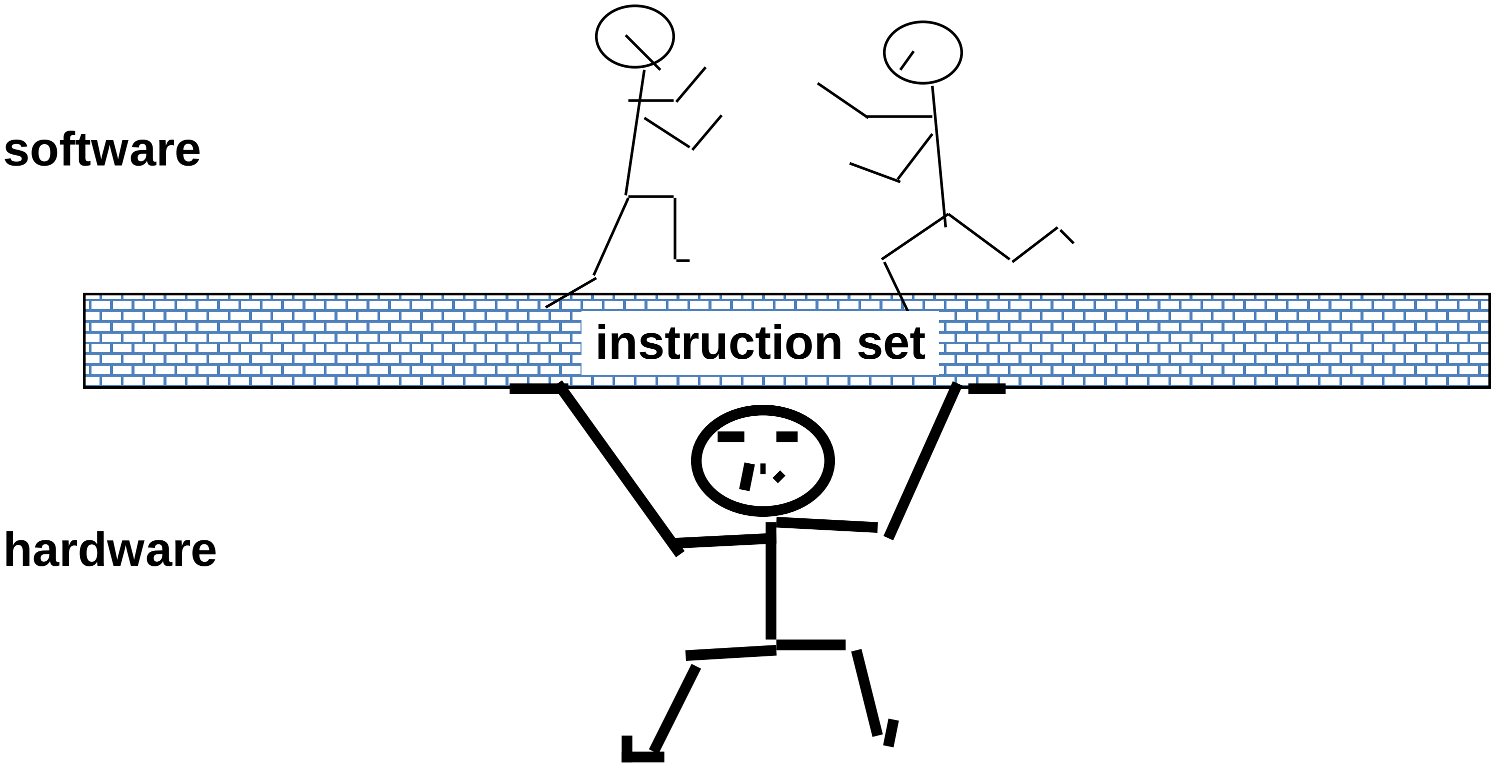
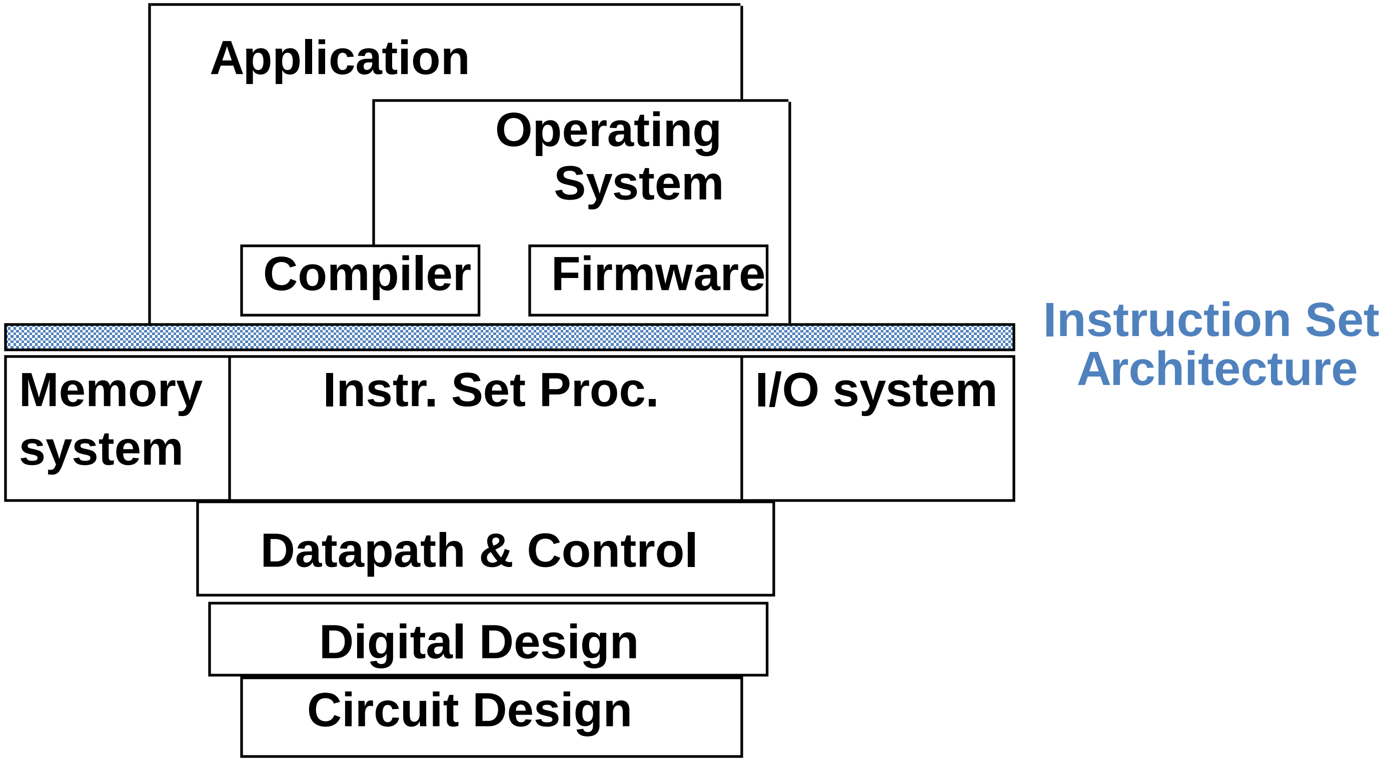
The Instruction Set: a Critical Interface



One of the most important abstractions is ISA

- A critical interface between HW and SW
- Example: MIPS (originally **Microprocessor without Interlocked Pipeline Stages)**

How Do the Pieces Fit Together?



- Coordination of many *levels of abstraction*
- Under a rapidly changing set of forces
- Design, measurement, *and* evaluation

Chapter 1

Computer Abstractions and Technology

The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore's Law
- Makes novel applications feasible
  - Computers in automobiles
  - Cell phones
  - Human genome project
  - World Wide Web
  - Search Engines
- Computers are pervasive

Classes of Computers

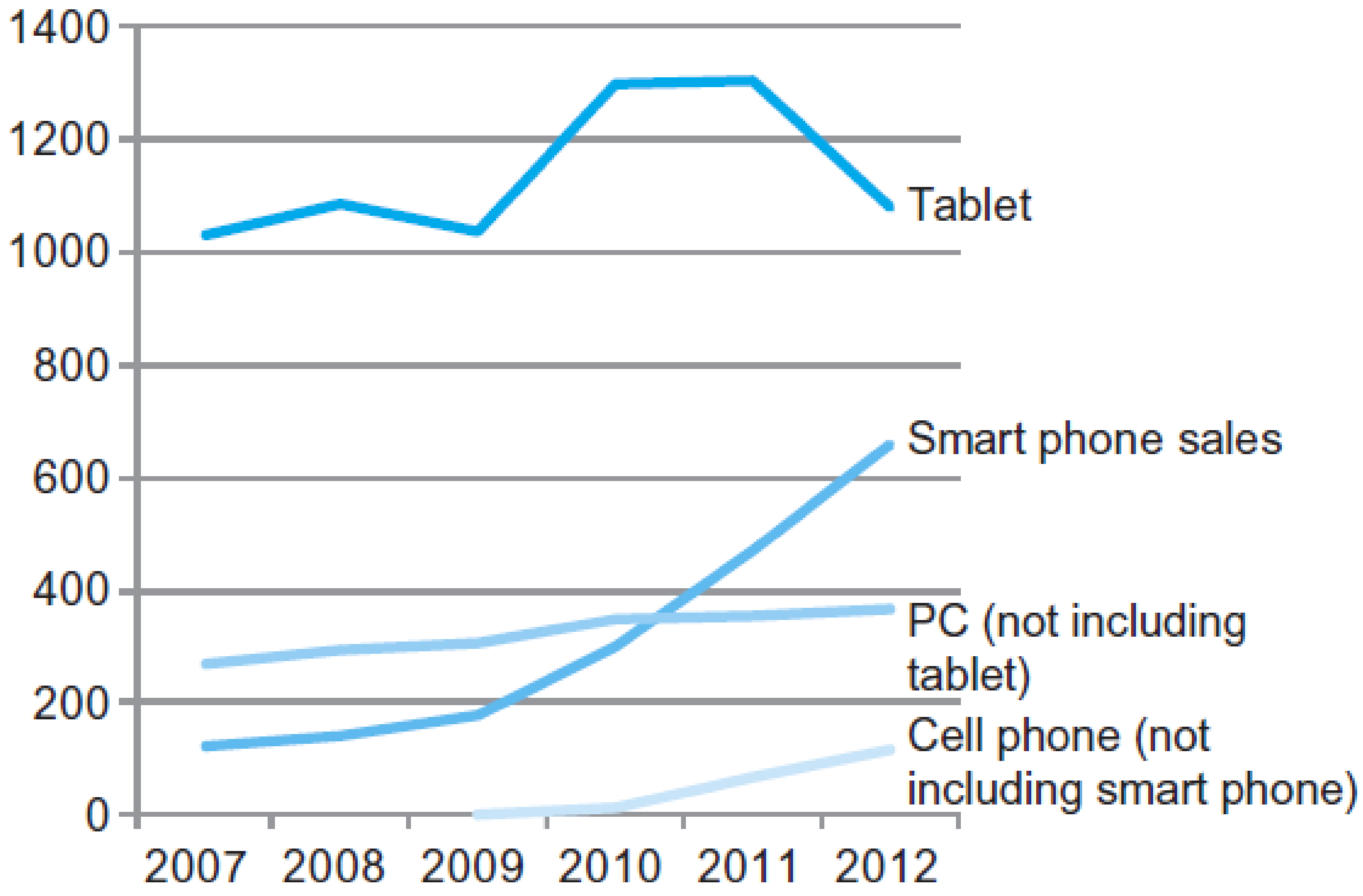
- Personal computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff

- Server computers
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized

Classes of Computers

- Supercomputers
  - High-end scientific and engineering calculations
  - Highest capability but represent a small fraction of the overall computer market
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

The PostPC Era



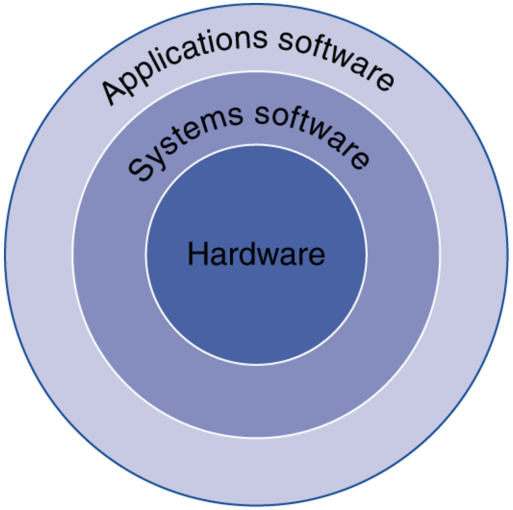
The PostPC Era

- Personal Mobile Device (PMD)
  - Battery operated
  - Connects to the Internet
  - Hundreds of dollars
  - Smart phones, tablets, electronic glasses
- Cloud computing
  - Warehouse Scale Computers (WSC)
  - Software as a Service (SaaS)
  - Portion of software run on a PMD and a portion run in the Cloud
  - Amazon and Google

Understanding Performance

- Algorithm
  - Determines number of operations executed
- Programming language, compiler, architecture
  - Determine number of machine instructions executed per operation
- Processor and memory system
  - Determine how fast instructions are executed
- I/O system (including OS)
  - Determines how fast I/O operations are executed

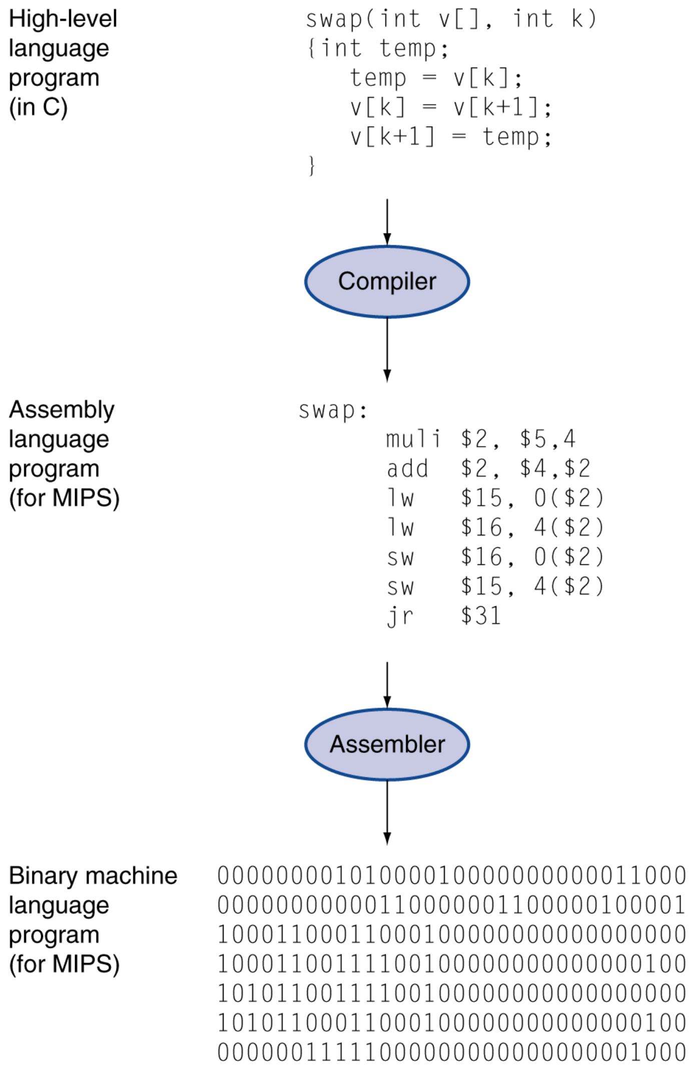
Below Your Program



- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

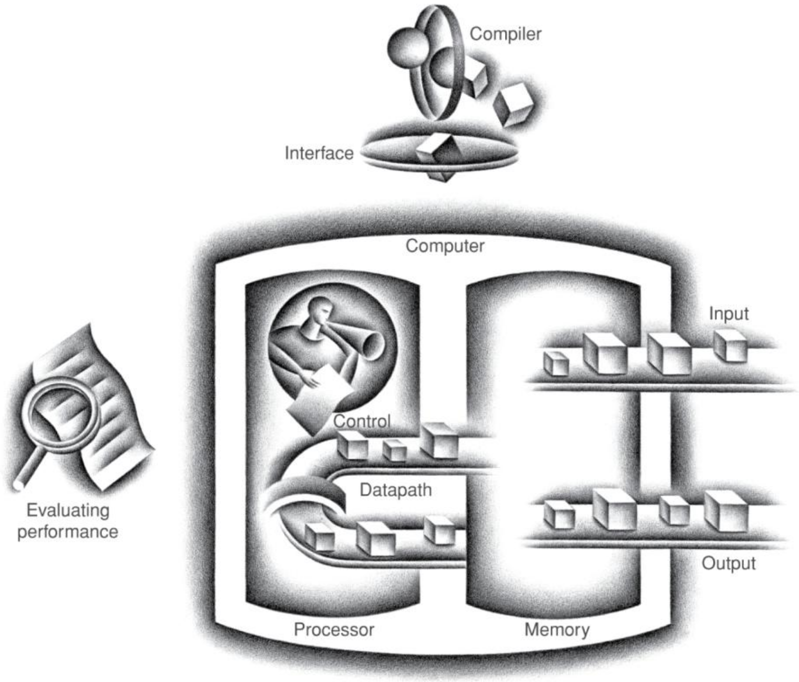
```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Components of a Computer

§1.4 Under the Covers

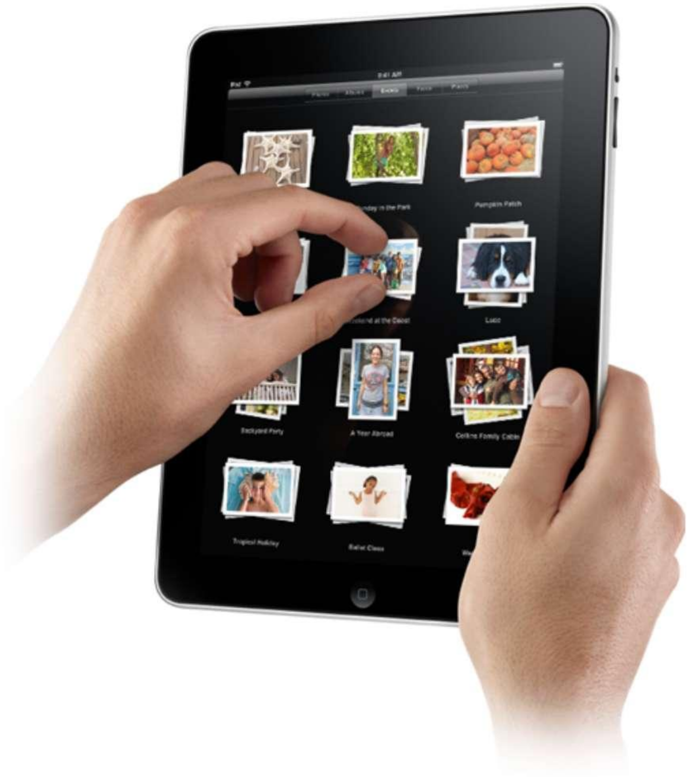
The BIG Picture



- Same components for all kinds of computer
  - Desktop, server, embedded
- Input/output includes
  - User-interface devices
    - Display, keyboard, mouse
  - Storage devices
    - Hard disk, CD/DVD, flash
  - Network adapters
    - For communicating with other computers

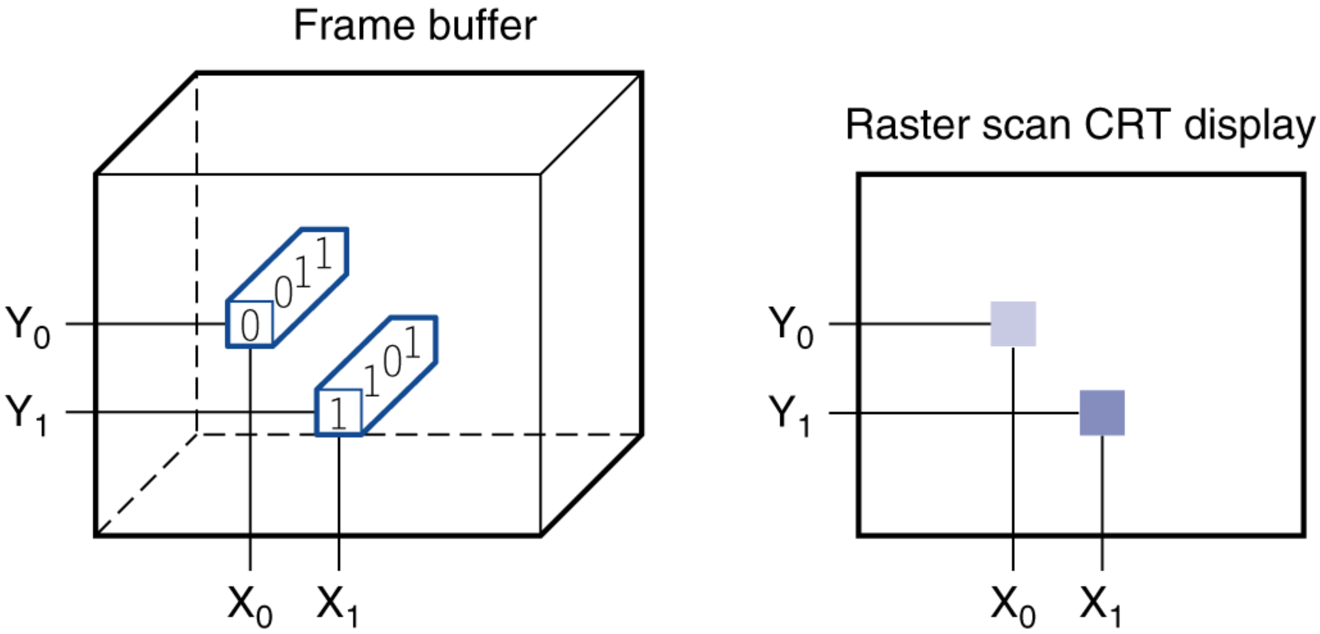
Touchscreen

- PostPC device
- Supersedes keyboard and mouse
- Resistive and Capacitive types
  - Most tablets, smart phones use capacitive
  - Capacitive allows multiple touches simultaneously

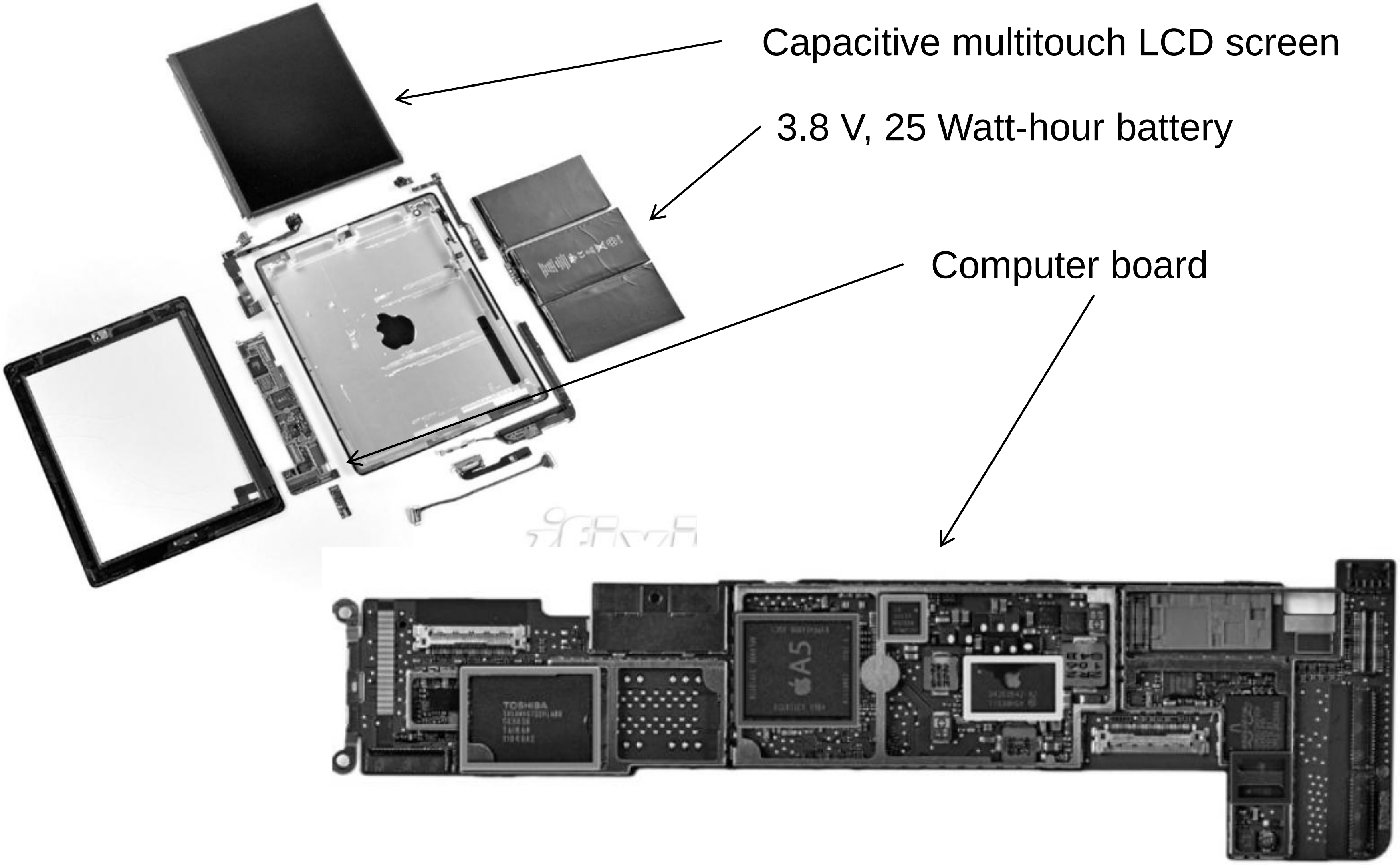


Through the Looking Glass

- LCD screen: picture elements (pixels)
  - Mirrors content of frame buffer memory



Opening the Box



Capacitive multitouch LCD screen

3.8 V, 25 Watt-hour battery

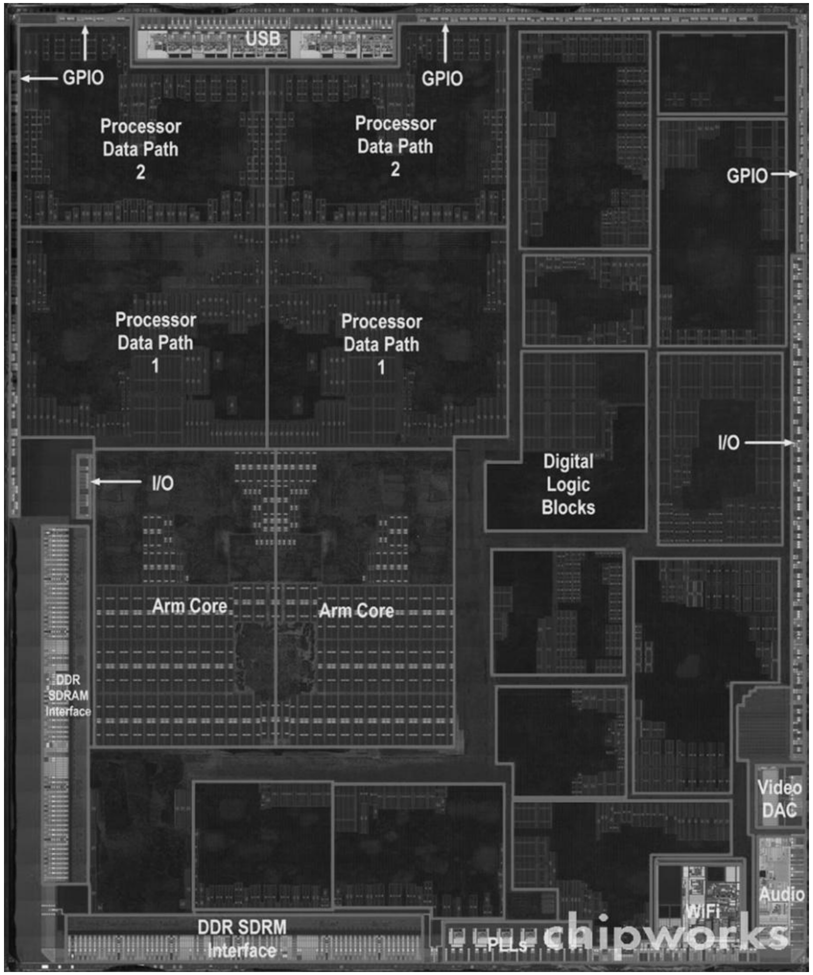
Computer board

Inside the Processor (CPU)

- Datapath: performs operations on data
- Control: sequences datapath, memory, ...
- Cache memory
  - Small fast SRAM memory for immediate access to data

Inside the Processor

- Apple A5



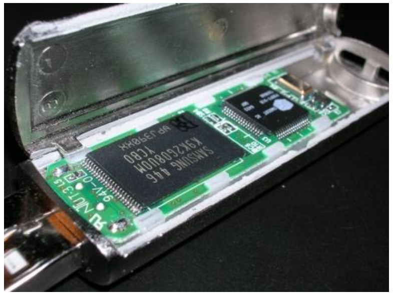
Abstractions

The BIG Picture

- Abstraction helps us deal with complexity
  - Hide lower-level detail
- Instruction set architecture (ISA)
  - The hardware/software interface
- Application binary interface
  - The ISA plus system software interface
- Implementation
  - The details underlying and interface

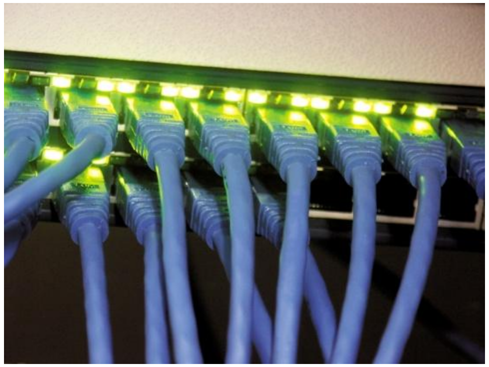
A Safe Place for Data

- Volatile main memory
  - Loses instructions and data when power off
- Non-volatile secondary memory
  - Magnetic disk
  - Flash memory
  - Optical disk (CDROM, DVD)



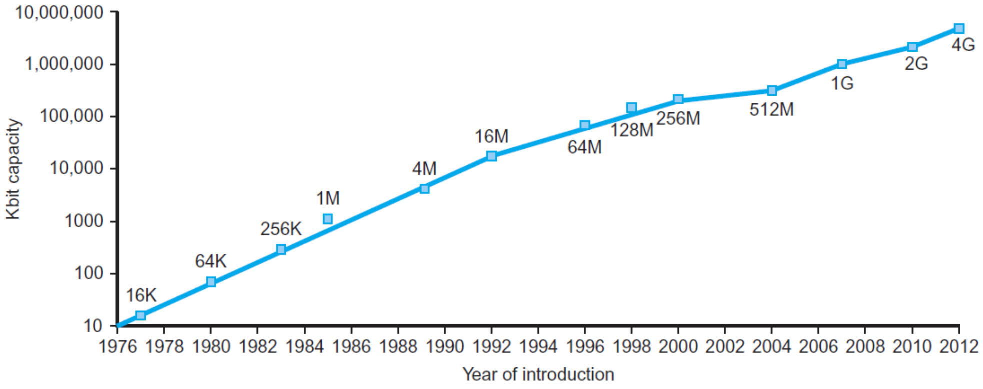
Networks

- Communication, resource sharing, nonlocal access
- Local area network (LAN): Ethernet
- Wide area network (WAN): the Internet
- Wireless network: WiFi, Bluetooth



Technology Trends

- Electronics technology continues to evolve
  - Increased capacity and performance
  - Reduced cost

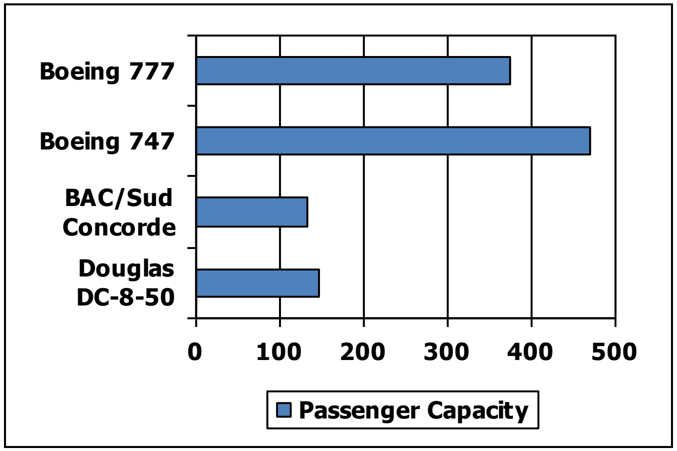


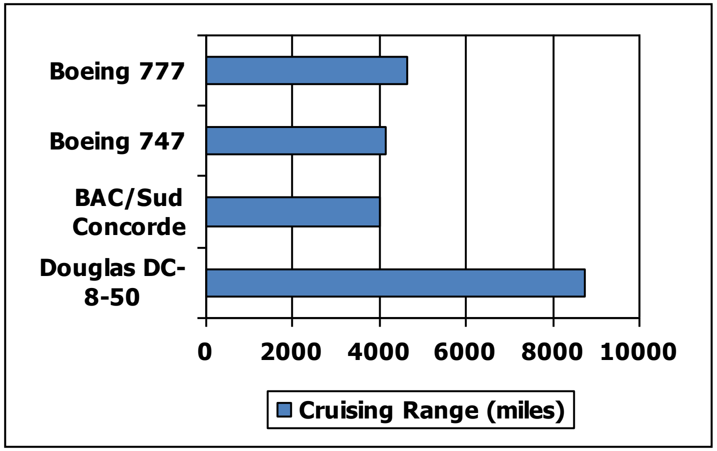
DRAM capacity

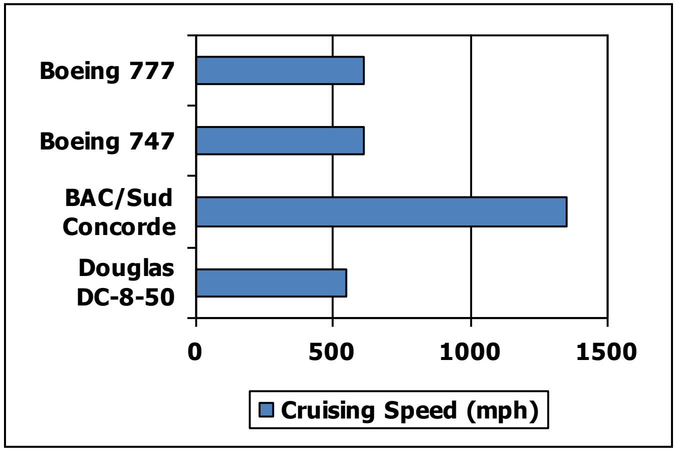
Year	Technology	Relative performance/cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit (IC)	900
1995	Very large scale IC (VLSI)	2,400,000
2013	Ultra large scale IC	250,000,000,000

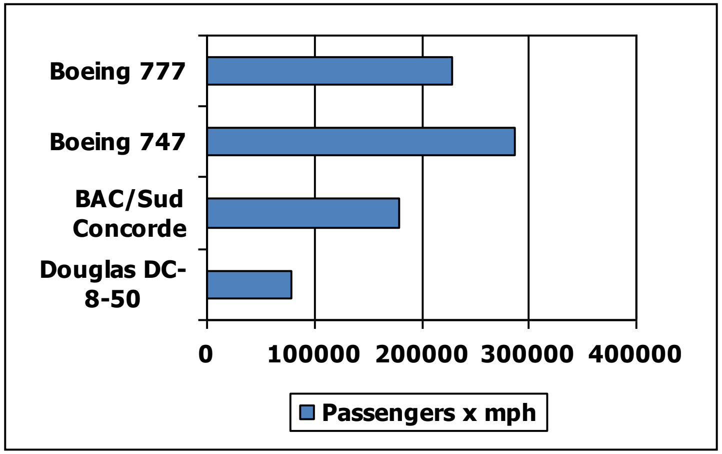
Defining Performance

- Which airplane has the best performance?









Response Time and Throughput

- Response time
  - How long it takes to do a task
- Throughput
  - Total work done per unit time
    - e.g., tasks/transactions/… per hour
- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?
- We’ll focus on response time for now…

Relative Performance

- Define Performance = 1/Execution Time
- "X is *n* time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

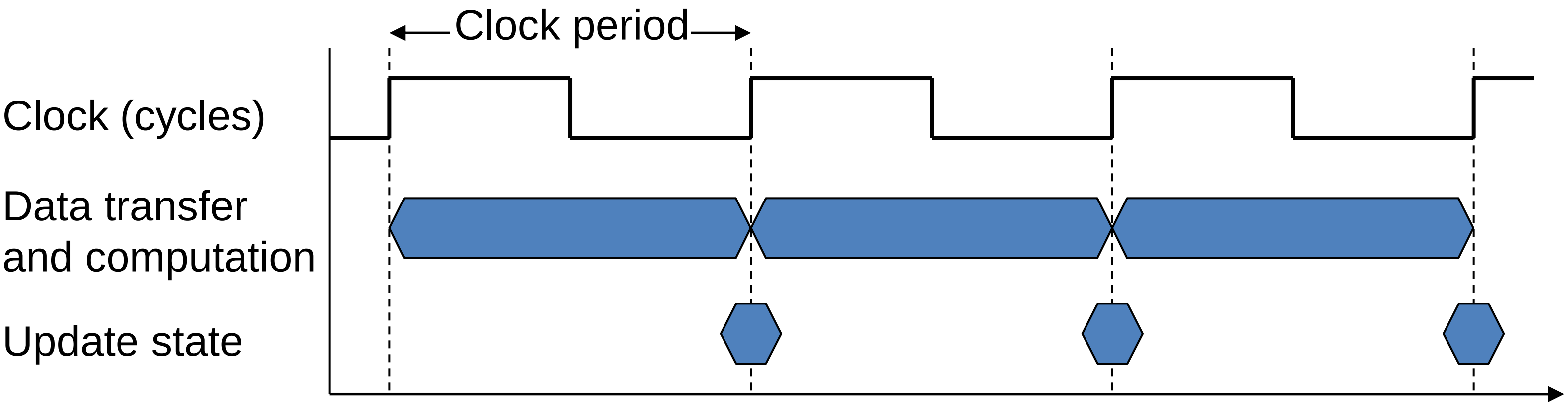
- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time$_B$ / Execution Time$_A$ = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Measuring Execution Time

- Elapsed time
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- CPU time
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprises user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): Inverse of the clock period (cycles per second)
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= \text{I} \times 2.0 \times 250\text{ps} = \text{I} \times 500\text{ps}\end{aligned}$$

A is faster…

$$\begin{aligned}\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= \text{I} \times 1.2 \times 500\text{ps} = \text{I} \times 600\text{ps}\end{aligned}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{\text{I} \times 600\text{ps}}{\text{I} \times 500\text{ps}} = 1.2$$

…by this much

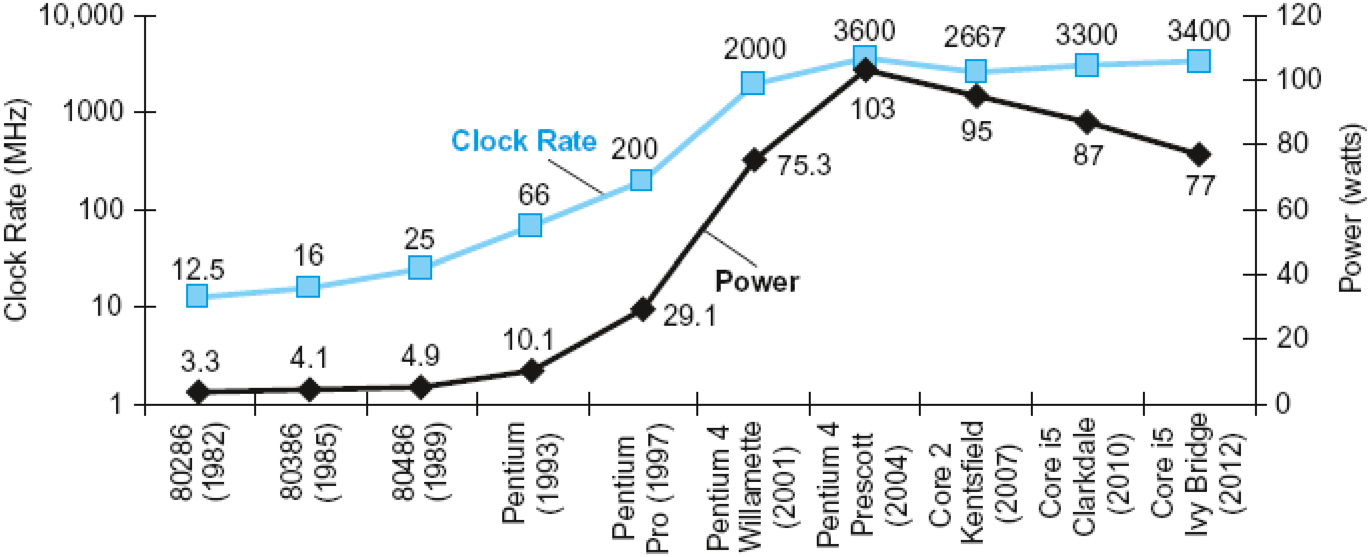
Performance Summary

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Power Trends



- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30

5V → 1V

×1000

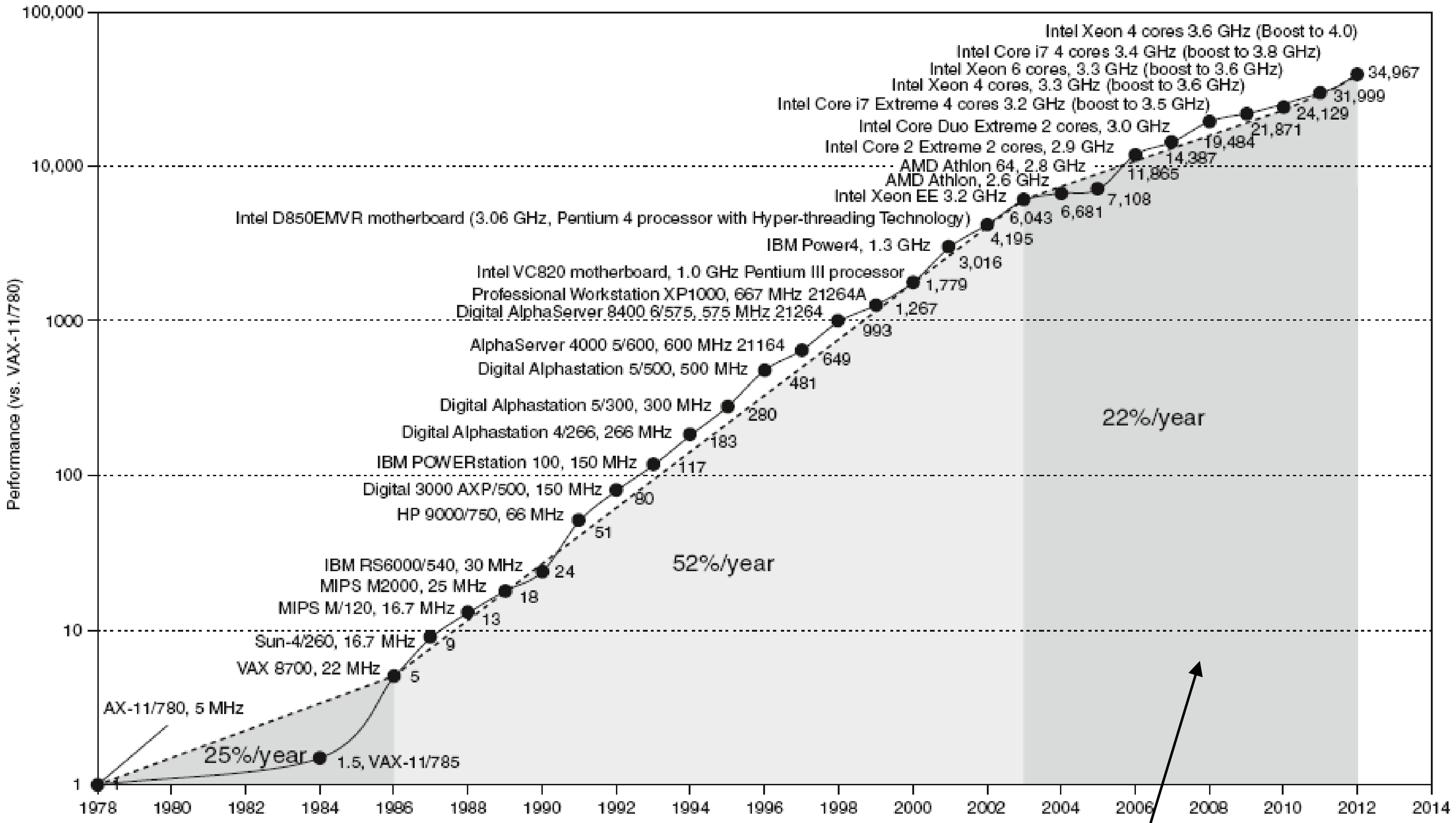
Reducing Power

- Suppose a new CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}{}^2 \times F_{old}} = 0.85^4 = 0.52$$

- The power wall
  - We can’t reduce voltage further
  - We can’t remove more heat
- How else can we improve performance?

Uniprocessor Performance



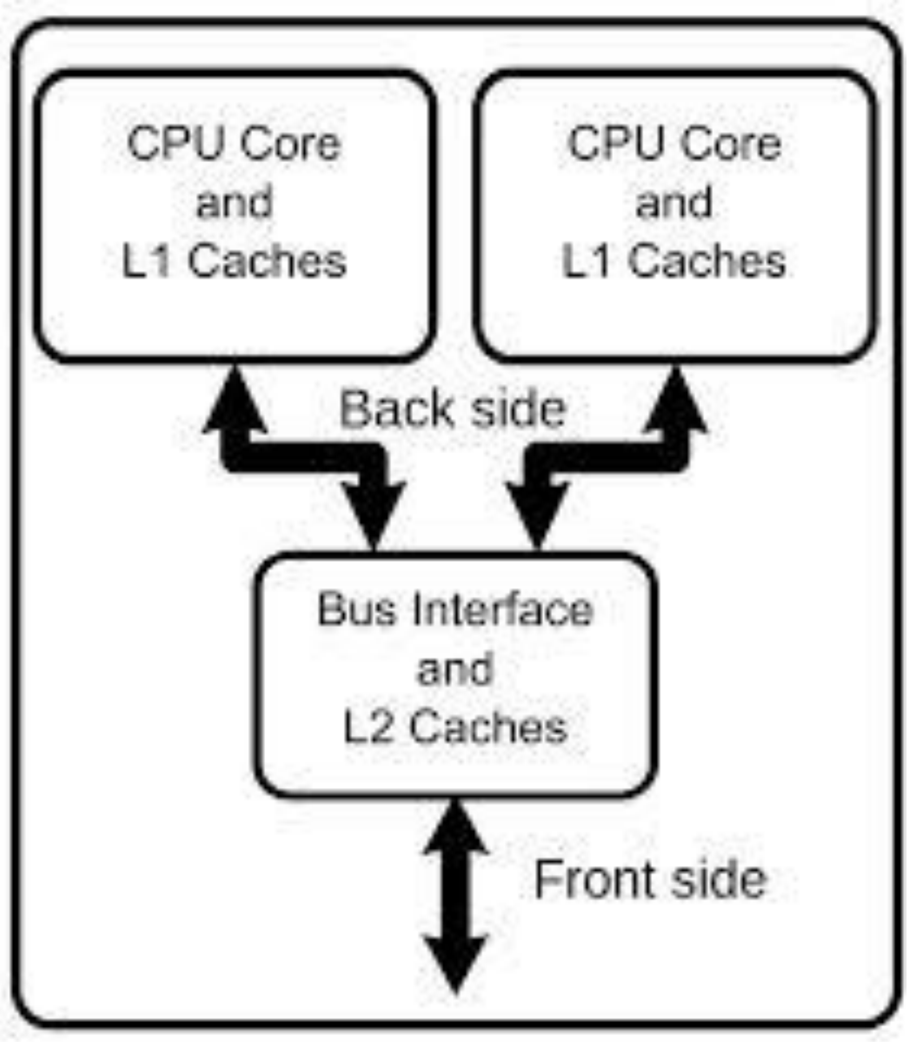
Constrained by power, instruction-level parallelism, memory latency

§1.8 The Sea Change: The Switch to Multiprocessors

Multiprocessors

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

Dual Core Processor

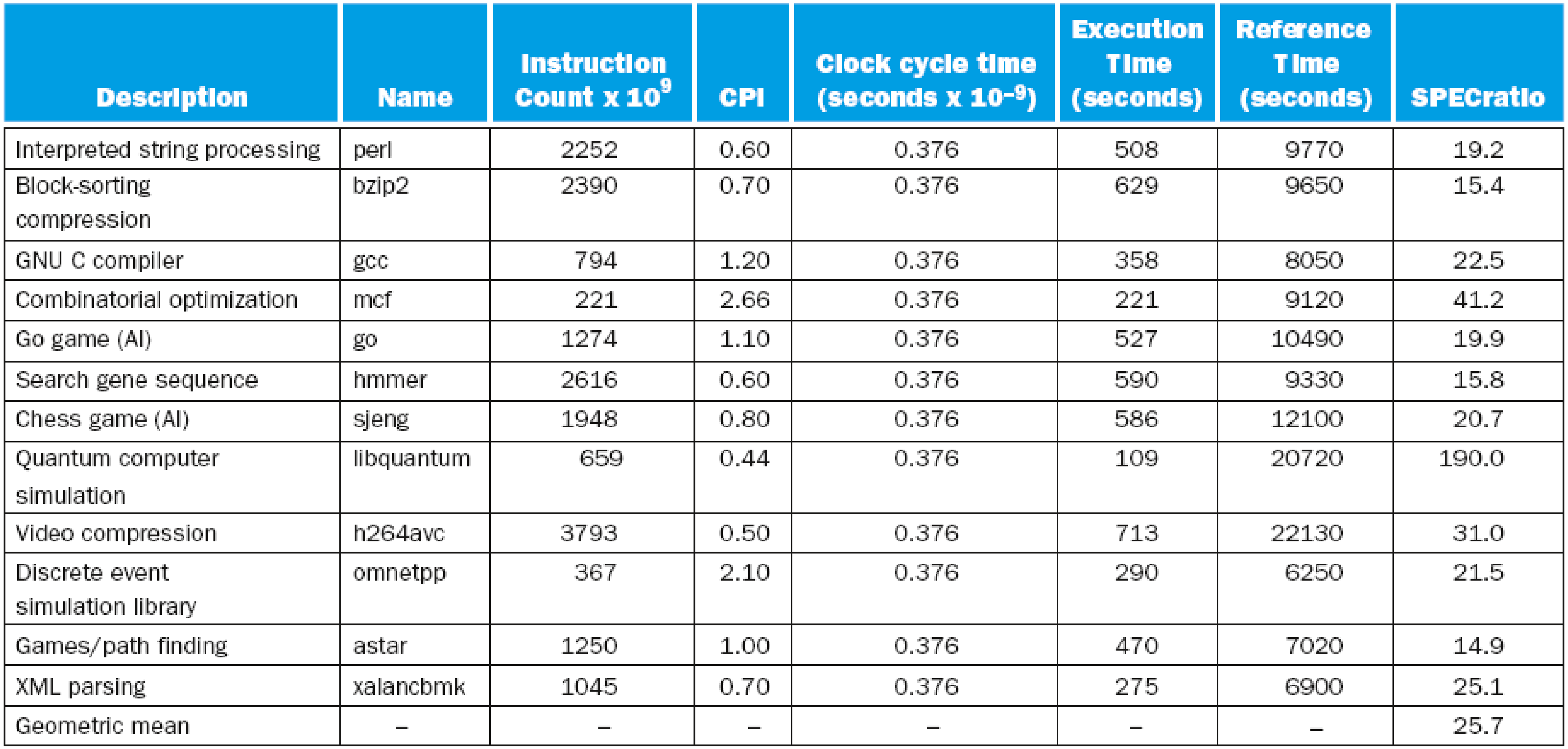


SPEC CPU Benchmark

- Programs used to measure performance
  - Supposedly typical of actual workload
- Standard Performance Evaluation Corp (SPEC)
  - Develops benchmarks for CPU, I/O, Web, …
- SPEC CPU2006
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalize relative to reference machine
  - Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

CINT2006 for Intel Core i7 920



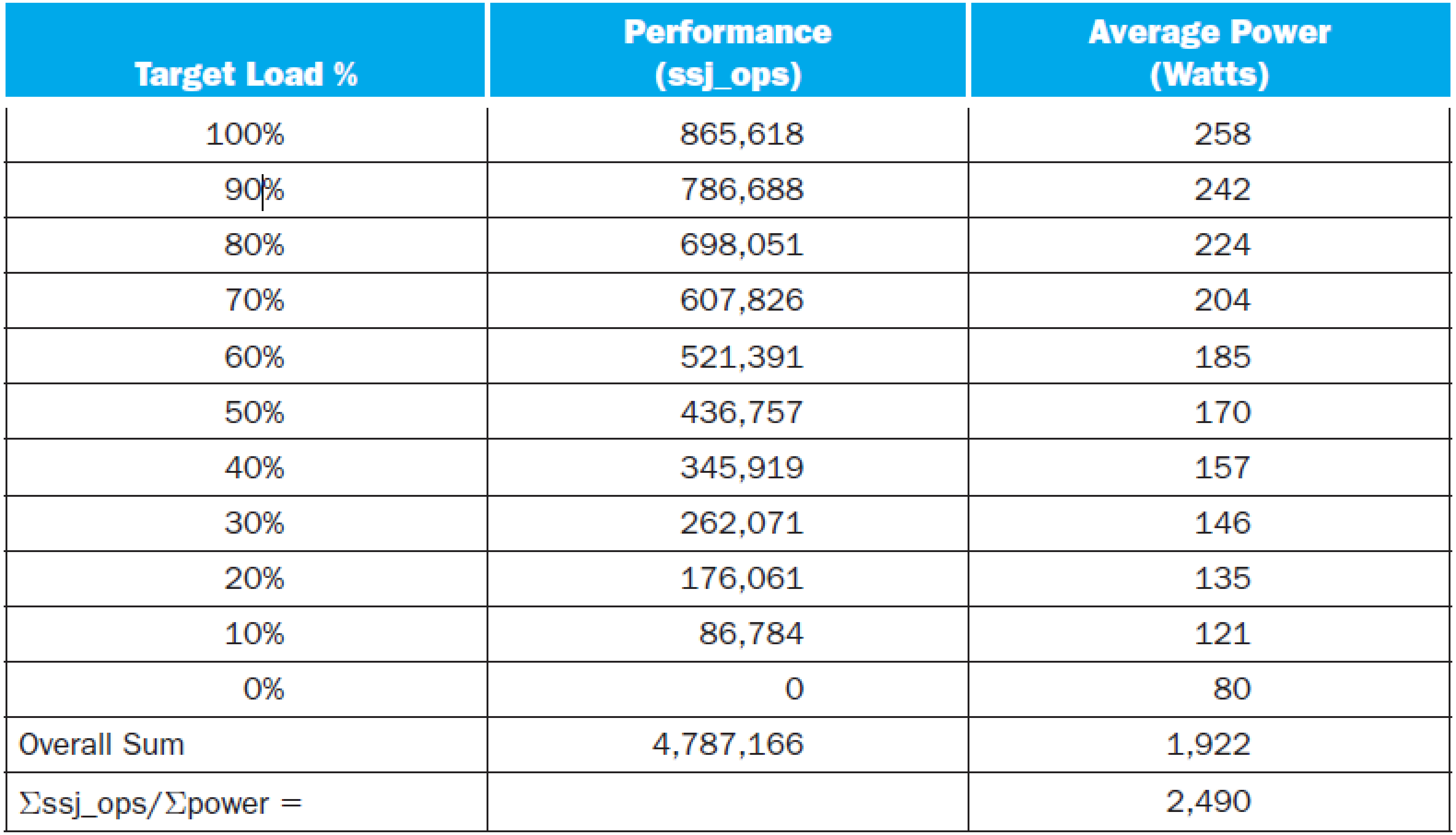
Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

SPEC Power Benchmark

- Power consumption of server at different workload levels
  - Performance: ssj_ops/sec
  - Power: Watts (Joules/sec)

$$\text{Overall ssj\_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i\right) \Big/ \left(\sum_{i=0}^{10} \text{power}_i\right)$$

SPECpower_ssj2008 for Xeon X5650



Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1,922
Σssj_ops/Σpower =		2,490

Pitfall: Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!
- Corollary: make the common case fast

Pitfall: MIPS as a Performance Metric

- MIPS: Millions of Instructions Per Second
  - Doesn’t account for
    - Differences in ISAs between computers
    - Differences in complexity between instructions

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$= \frac{\text{Instruction count}}{\dfrac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

- CPI varies between programs on a given CPU

Concluding Remarks

§1.9 Concluding Remarks

- Cost/performance is improving
  - Due to underlying technology development
- Hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure
- Power is a limiting factor
  - Use parallelism to improve performance

What is Computer Architecture

Computer Architecture =

Instruction Set Architecture +

Machine Organization

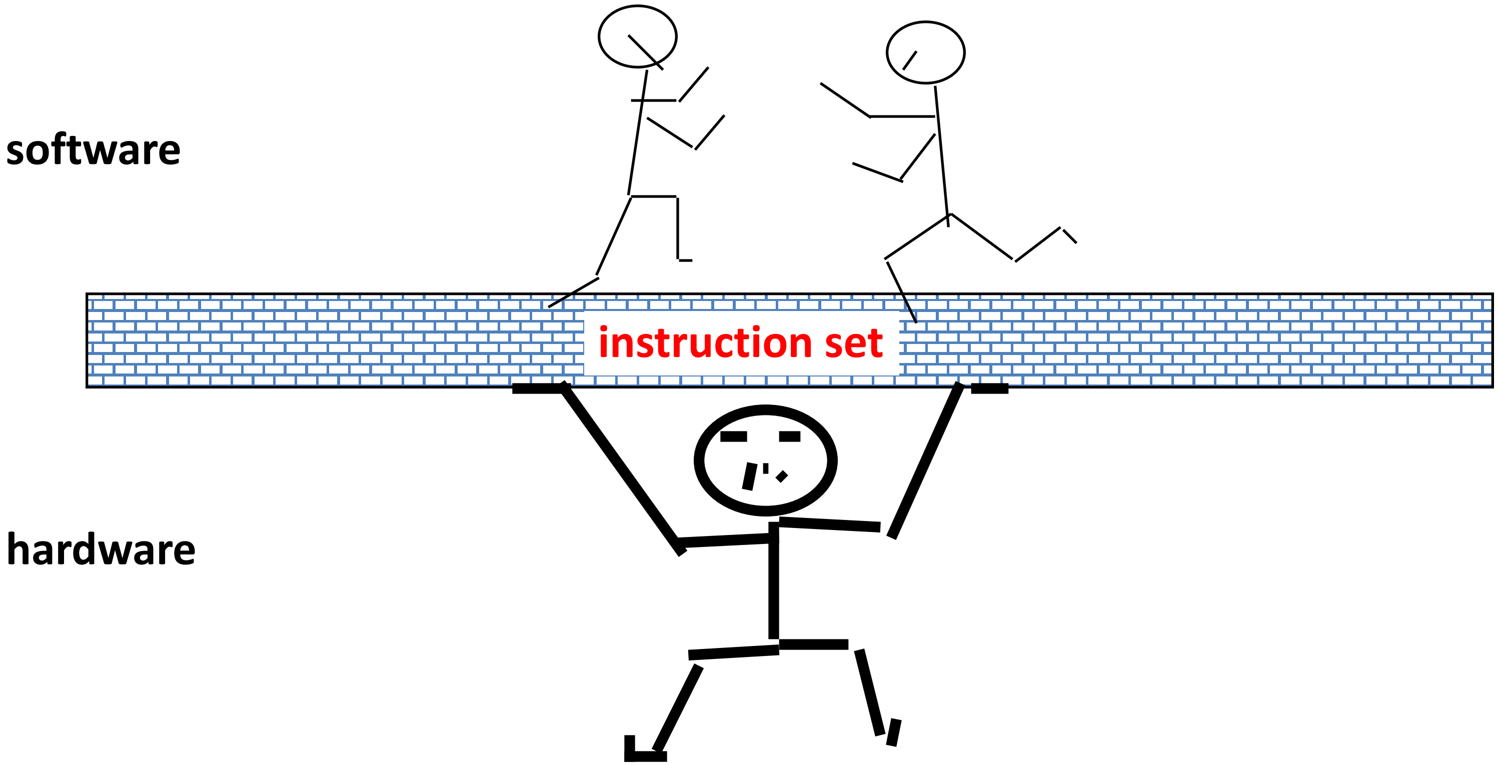
Instruction Set Architecture

- Defines any aspects of the processor that an assembly language programmer needs to know, in order to write a correct program
- Specifies:
  - The registers accessible to the programmer, their size and the instructions in the instructions set that can use each register
  - Information necessary to interact with the memory
    - Certain microprocessors require instructions to start only at specific memory locations; this alignment of the instructions will be part of the instruction architecture
  - How microprocessor reacts to interrupts
    - Some microprocessors have interrupts, that cause the processor to stop what is doing and perform some other preprogrammed functions (interrupt routines)

Instruction Set Architecture

- **Instruction Set**
- **Instruction Formats**
- **Modes of Addressing and Accessing Data**

The Instruction Set: a Critical Interface



software

instruction set

hardware

What is an Instruction Set?

- The complete collection of instructions that are understood by a CPU
- Machine language: binary representation of operations and (addresses of) arguments
- Assembly language: mnemonic representation for humans, e.g.,

Elements of an Instruction

- Operation code (opcode)
  - Do this: ADD, SUB, MUL, DIV, LOAD, STOR
- Source operand reference
  - To this: (address of) argument of op, e.g. register, memory location
- Result operand reference
  - Put the result here (as above)
- Next instruction reference (often implicit)
  - When you have done that, do this: BR

Instruction formats (1)

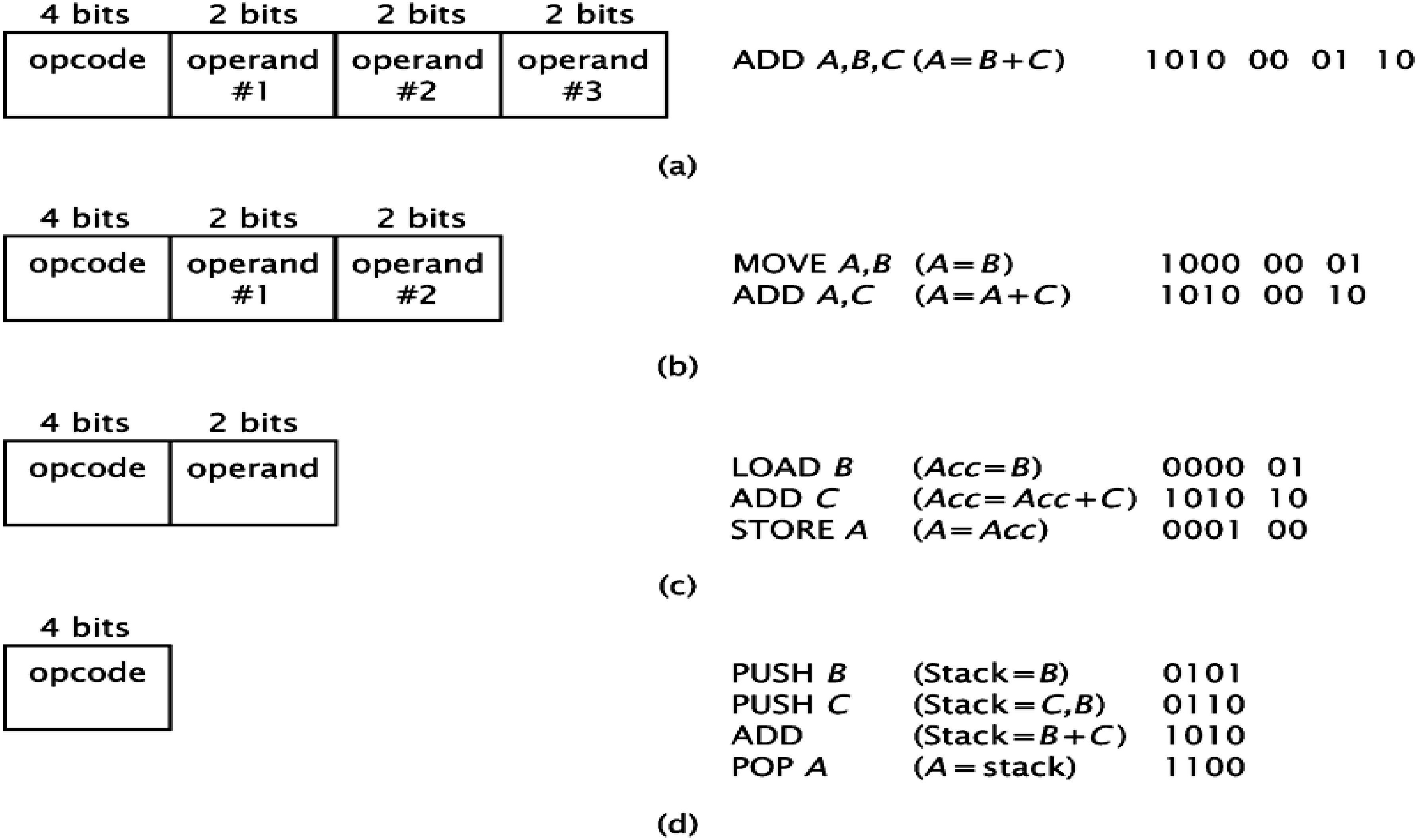
- An instruction is represented as a binary value with specific format, called the **instruction code**

- It is made out of different groups of bits, with different significations:
  - **Opcode** – represents the operation to be performed (it is the instruction identifier)
  - **Operands** – one, two or three represent the operands of the operation to be performed

- A microprocessor can have one format for all the instructions or can have several different formats

- An instruction is represented by a single instruction code

Instruction formats (2)



4 bits	2 bits	2 bits	2 bits
opcode	operand #1	operand #2	operand #3

ADD *A*,*B*,*C*	(*A*=*B*+*C*)	1010 00 01 10

(a)

4 bits	2 bits	2 bits
opcode	operand #1	operand #2

MOVE *A*,*B*	(*A*=*B*)	1000 00 01
ADD *A*,*C*	(*A*=*A*+*C*)	1010 00 10

(b)

4 bits	2 bits
opcode	operand

LOAD *B*	(*Acc*=*B*)	0000 01
ADD *C*	(*Acc*=*Acc*+*C*)	1010 10
STORE *A*	(*A*=*Acc*)	0001 00

(c)

4 bits
opcode

PUSH *B*	(Stack=*B*)	0101
PUSH *C*	(Stack=*C*,*B*)	0110
ADD	(Stack=*B*+*C*)	1010
POP *A*	(*A*=stack)	1100

(d)

What Must an Instruction Specify?(I)

Data Flow

←

- Which operation to perform add r0, r1, r3
  - Op code: add, load, branch, etc.
- Where to find the operands: add r0, r1, r3
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result add r0, r1, r3
  - Again CPU register or memory cell

RISC vs. CISC (1)

- The believe that better performance would be obtained by reducing the number of instruction required to implement a program, lead to design of processors with very complex instructions (CISC)
  - CISC – ***C***omplex ***I***nstruction ***S***et ***C***omputers
- As compiler technologies improved, researchers started to wonder if CISC architectures really delivered better performances than architectures with simpler instruction set
  - RISC – ***R***educed ***I***nstruction ***S***et ***C***omputers

RISC vs. CISC (2)

- CISC
  - Fewer instructions to execute a given task than RISC
  - Programs for CISC take less storage space than programs for RISC
  - Arithmetic or other instructions may read their operand from memory and could write the result in memory
- RISC
  - Simpler instructions, faster execution speeds per instruction, more instructions executed in same amount of time than CISC
  - Cheaper to implement (simple instruction set results in simple implementation internal micro-architecture)
  - Load/Store architecture – only load and store are used to access the external memory

RISC vs. CISC (3)

RISC	CISC
LD R4, (R1) LD R5, (R2) ADD R6, R4, R5 ST (R3), R6	ADD (R3), (R2), (R1)

- Addition of two operands from memory, with result written in memory, in RISC and CISC architectures
- Having an operation broken into small instructions (RISC) allows the compiler to optimize the code
  - i.e. between the two LD instructions (memory is slow) the compiler can add some instructions that don’t need memory access
- The CISC instruction has no option but to wait for its operands to come from the memory, potentially delaying other instructions

Types of Instructions

- Data Transfer Instructions
- Arithmetic Instructions
- Logical Instructions
- Branching instructions
- Control instructions

Data Transfer

- Store, load, exchange, move, clear, set, push, pop
- Specifies: source and destination (memory, register, stack), amount of data
- May be different instructions for different (size, location) movements, e.g., MOV A,B

  LDA 4050H

Arithmetic

- Add, Subtract, Multiply, Divide for signed integer (+ floating point and packed decimal) – may involve data movement
- May include
  - Absolute (|a|)
  - Increment (a++)
  - Decrement (a--)
  - Negate (-a)

Logical

- Bitwise operations: AND, OR, NOT, XOR, TEST, CMP, SET
- Shifting and rotating functions, e.g.
  - logical right shift send 8-bit character from 16-bit word
  - arithmetic right shift: division and truncation for odd numbers
  - arithmetic left shift: multiplication without overflow

Transfer of Control

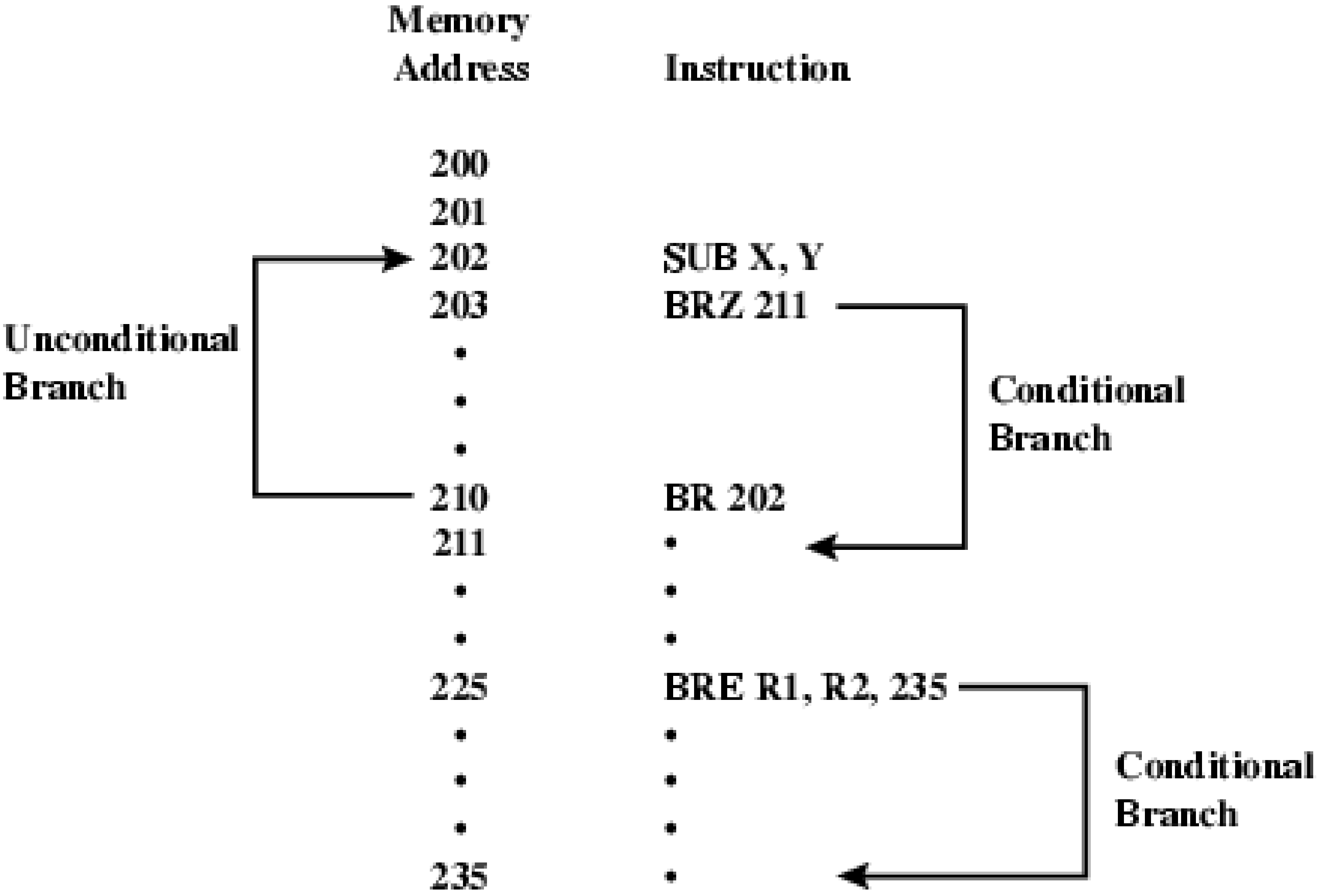
- Skip, e.g., increment and skip if zero:

  ISZ Reg1, cf. jumping out from loop

- Branch instructions: BRZ X (branch to X if result is zero), BRP X (positive), BRN X (negative), BRE X,R1,R2 (equal)

- Procedure (economy and modularity): call and return

Branch Instruction



Memory Address	Instruction
200	
201	
202	SUB X, Y
203	BRZ 211
•	
•	
•	
210	BR 202
211	•
•	•
•	•
225	BRE R1, R2, 235
•	•
•	•
•	•
235	•

Unconditional Branch

Conditional Branch

Conditional Branch

Input/Output

- May be specific instructions, e.g. INPUT, OUTPUT
- May be done using data movement instructions (memory mapped I/O)
- May be done by a separate controller (DMA): Start I/O, Test I/O

Systems Control

- Privileged instructions: accessing control registers or process table
- CPU needs to be in specific state
- For operating systems use
- For handling interrupts

CPU Elements

- **Program Counter** or PC contains the address of the instruction that will be executed next

- **Stack** – a data structure of last in first out type. A stack is described by a special register – **stack pointer**
  - It can be used explicitly to save/restore data
  - It is used implicitly by procedure call instructions (if available in the instruction set)
- **IR** – instruction register that holds the current instruction being processed by the microprocessor

Execution of an instruction

Stages

- fetch
- Decode
- execute

Machine cycles and Instruction cycles

- Opcode fetch
- Memory read
- Memory write
- I/O read
- I/O write
- Interrupt acnowledge

Addressing Modes

The way by which the location of the operand is specified

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement (Indexed)
- Stack

Immediate Addressing

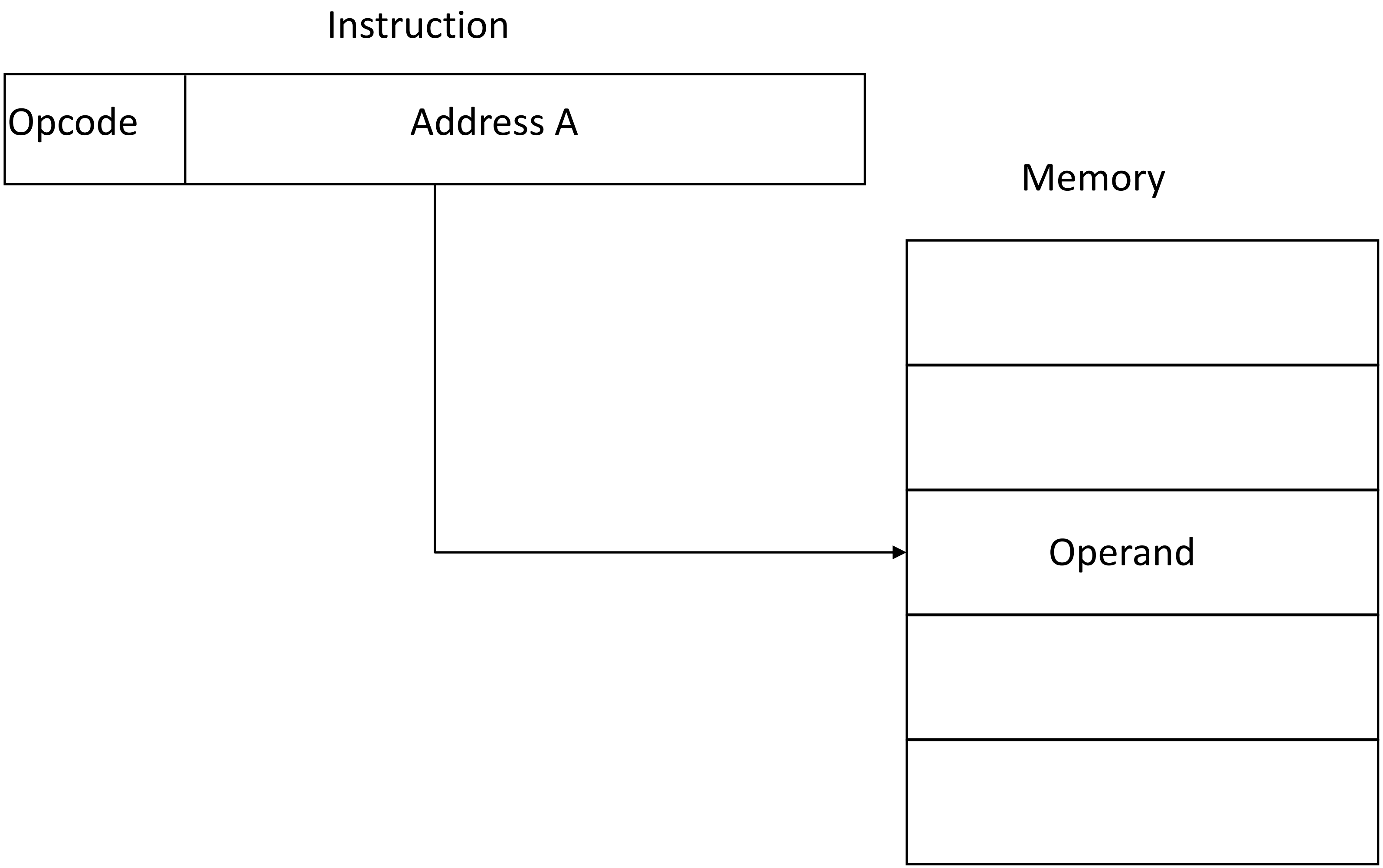
- Operand is part of instruction
- Operand = address field
- e.g., ADD #5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g., LDA 4050H

- Single memory reference to access data
- No additional calculations needed to work out effective address
- Limited address space (length of address field)

Direct Addressing Diagram



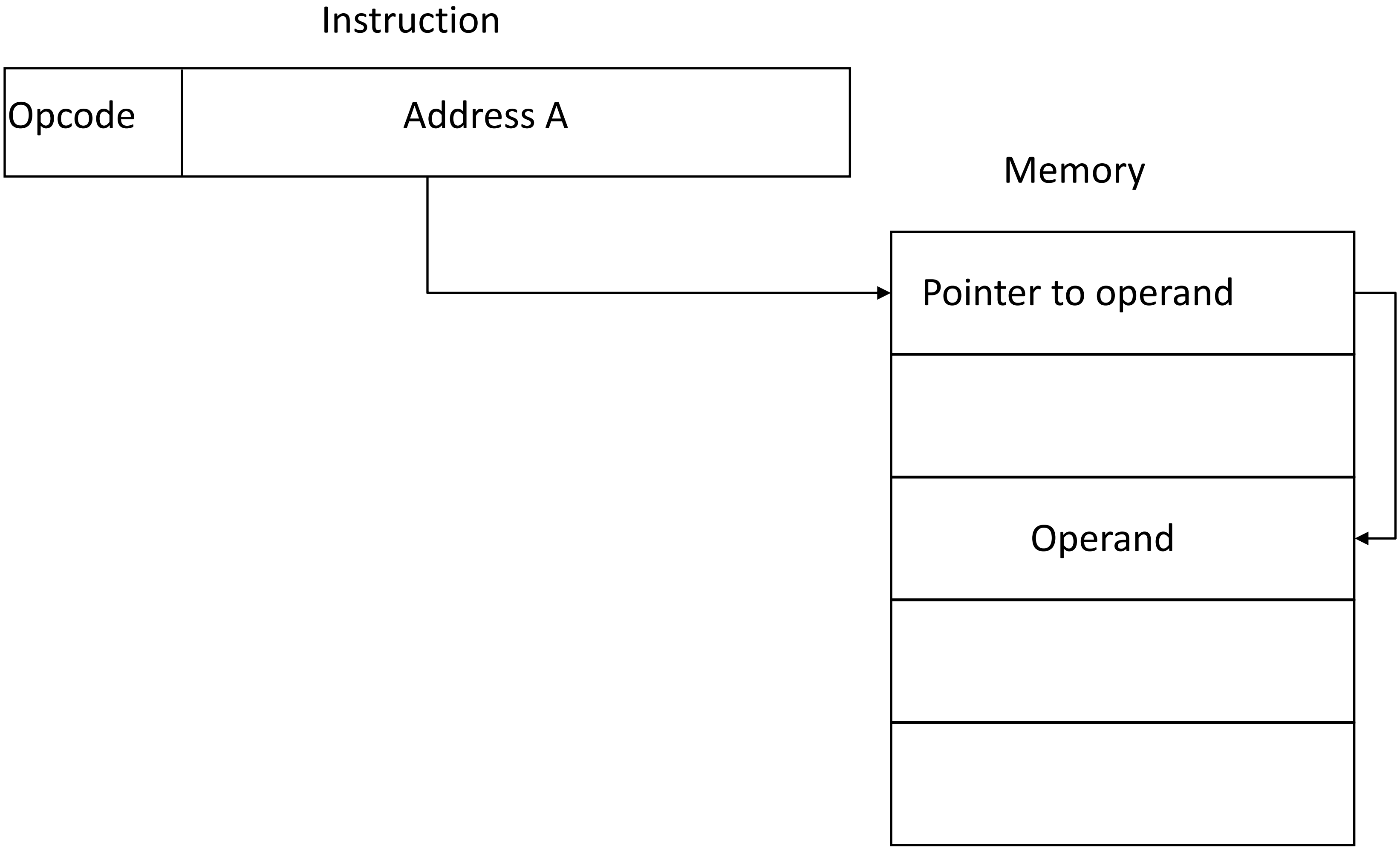
Indirect Addressing (1)

- 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 (2)

- Large address space
- 2^n where n = word length
- May be nested, multilevel, cascaded
  - e.g. EA = (((A)))
- Multiple memory accesses to find operand
- Hence slower

Indirect Addressing Diagram



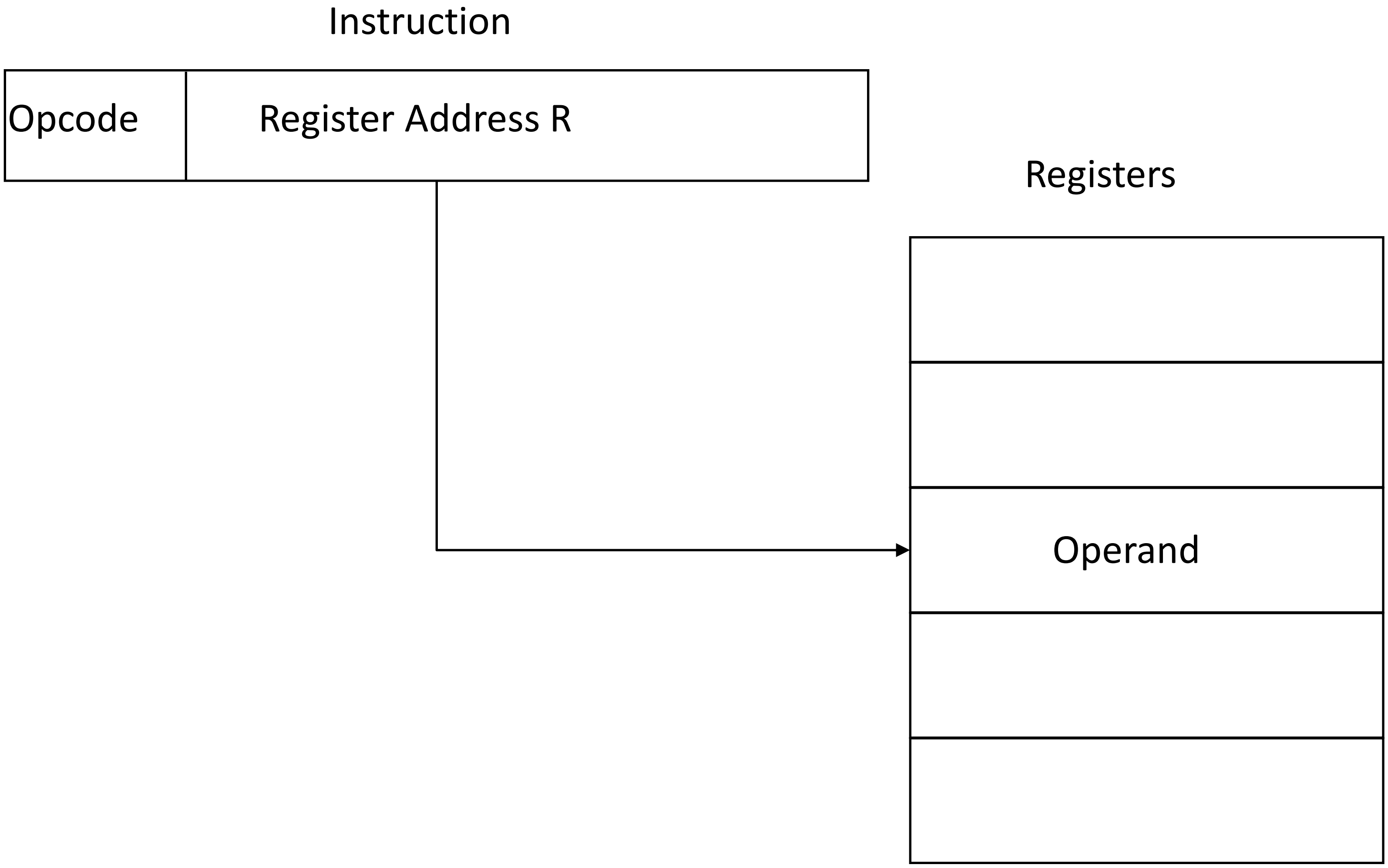
Register Addressing (1)

- 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 (2)

- No memory access
- Very fast execution
- Very limited address space
- Multiple registers helps performance
  - Requires good assembly programming or compiler writing – see register renaming

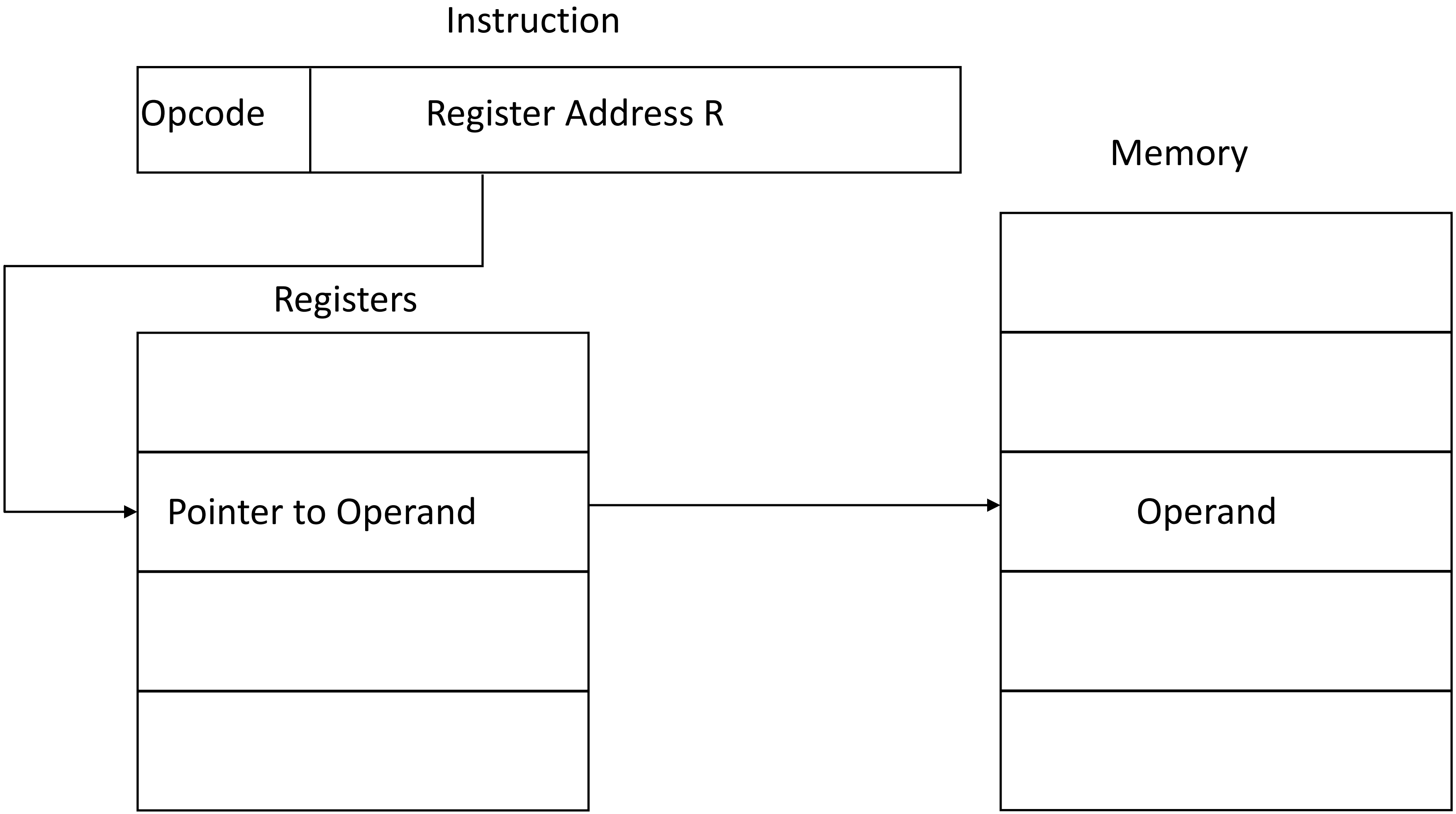
Register Addressing Diagram



Register Indirect Addressing

- EA = (R)
- Operand is in memory cell pointed to by contents of register R
- Large address space (2^n)
- One fewer memory access than indirect addressing

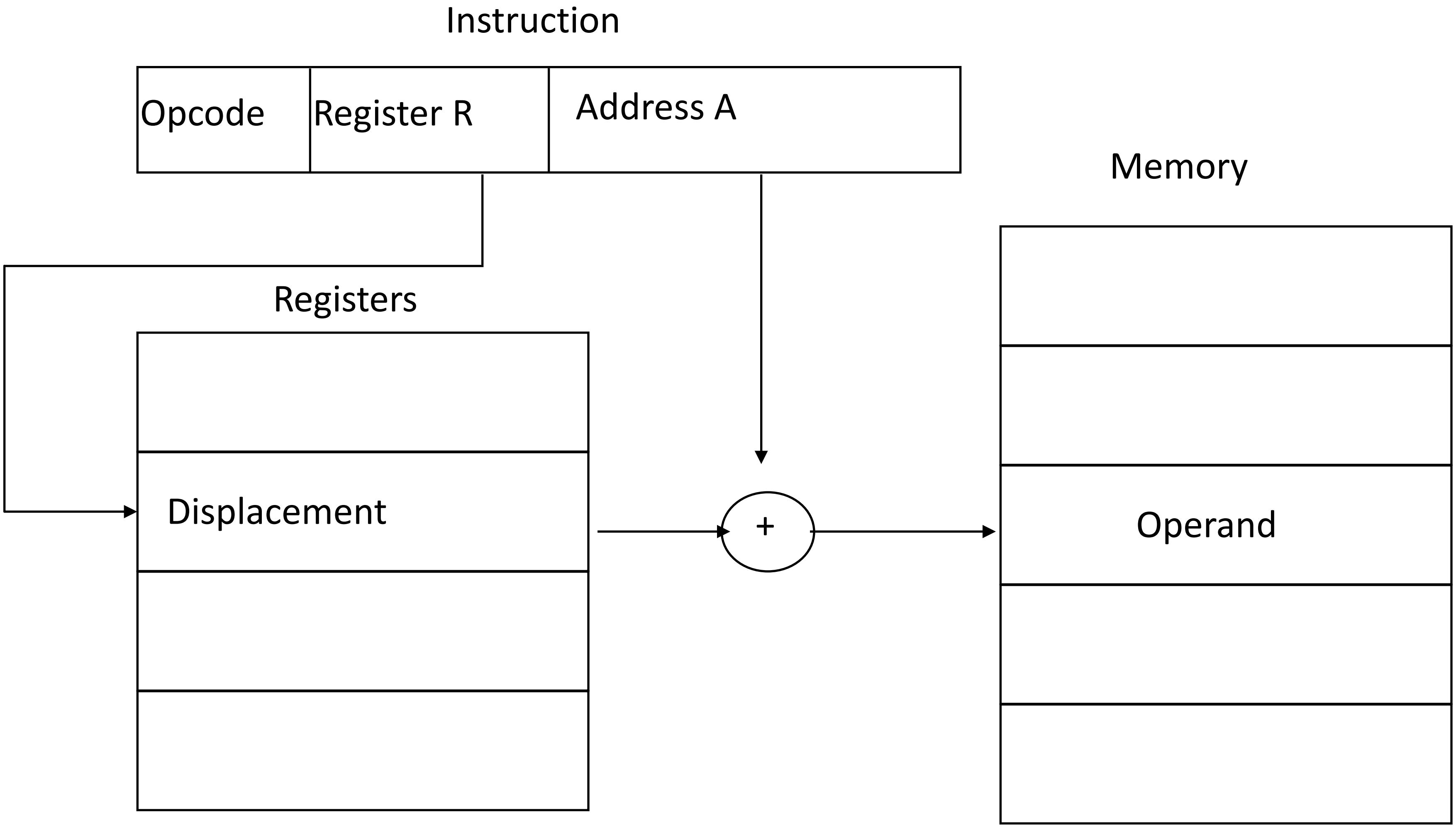
Register Indirect Addressing Diagram



Displacement Addressing

- EA = A + (R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa
- See segmentation

Displacement Addressing Diagram



Relative Addressing

- A version of displacement addressing
- R = Program counter, PC
- EA = A + (PC)
- i.e., get operand from A cells away from current location pointed to by PC

Base-Register Addressing

- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
- e.g., segment registers in 80x86

Stack Addressing

- Operand is (implicitly) on top of stack
- e.g.
  - ADD Pop top two items from stack and add and push result on top

Understanding ARM Instruction Set Architecture (ISA) using ARM Simulator

Importance of Instruction Set

- *Computer Architecture* and *Computer Organization* are two related courses, however, their objectives are different
- *Computer Organization* presents how various components (Memory, ALU, I/O Ports, Bus, etc) of a Microprocessor can be designed and developed
  - *Computer Organization* course describes the design and development of individual components of a processor.
- *Computer Architecture* course on the other hand, describes various interfaces within and to a Microprocessor
  - Describes the interfaces between various components within a Microprocessor
  - Describes programmers’ interface to a Microprocessor using Instruction Set Architecture (ISA)

The ARM Instruction Set Architecture

Advanced RISC Machine (ARM)

- ARM stands for *Advanced RISC Machine*
- ARM family includes 8/16/32/64 bits processors
- Different versions of ARM processors share the same machine Instruction Set
- We will be studying 32-bit ARM processor using a ARM Simulator application, **ARMSim#**
- ARMSim# © R. N. Horspool, W. D. Lyons, M. Serra Department of Computer Science, University of Victoria

Main features of the ARM Instruction Set

- All *instructions are 32* bits long. ARM processor supports 8/16/32/64 bits data values
- ARM processor executes instructions in a 3-Stage pipeline.
- Most of the ARM instructions executed in a single cycle.
- Every instruction can be conditionally executed.
- ARM uses load/store architecture
  - Data processing instructions act only on registers (Register-Register Architecture)
    - Three operand format
    - Combined ALU and barrel shifter for high speed bit manipulation
  - Specific memory access instructions with powerful auto-indexing addressing modes.
    - 32 bit and 8 bit data types and also 16 bit data types on ARM Architecture v4.
    - Flexible multiple register load and store instructions

Processor Modes

- The ARM has six operating *modes*:
  - *User* (unprivileged mode under which most tasks run)
  - *FIQ* (entered when a high priority (fast) interrupt is raised)
  - *IRQ* (entered when a low priority (normal) interrupt is raised)
  - *Supervisor* (entered *on reset* and when a *Software Interrupt instruction is executed*)
  - *Abort* (used to handle memory access violations)
  - *Undef* (used to handle undefined instructions)

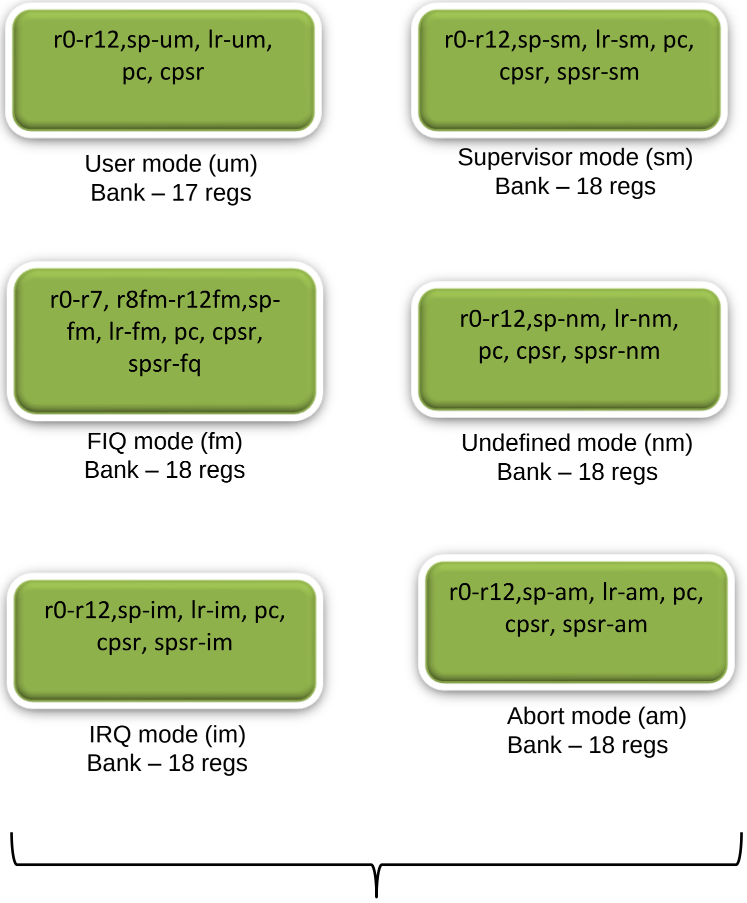
Addressing

- ARM includes capability to address byte, half word, and full word (32 bits)
- ARM instructions are always 32 bits in size, data can be of 8/16/32 bits
- This means, byte addresses of instructions start from 0 and increase in steps 4: 0, 4, 8, and so on
- Program Counter is initialized to 0 when the processor is reset, and increases in steps of 4
- This means, the least significant 2 bits of PC are always 0
- This reduces the number of bits required to store address and address offset of instructions with in programs; does not work for data

Registers and Register Banks

37 Registers

1. 18 General Purpose Registers: r0 – r12, r8fm-r12fm (fm stands of FIQ Mode)
2. 6 Stack Pointer (sp): sp-xx
3. 6 Link Register (lr): lr-xx
4. 1 Program Counter: pc
5. 1 Current Program Status Register: cpsr
6. 5 Saved Program Status Register: spsr-xx



r0-r12,sp-um, lr-um, pc, cpsr

User mode (um)
Bank – 17 regs

r0-r12,sp-sm, lr-sm, pc, cpsr, spsr-sm

Supervisor mode (sm)
Bank – 18 regs

r0-r7, r8fm-r12fm,sp-fm, lr-fm, pc, cpsr, spsr-fq

FIQ mode (fm)
Bank – 18 regs

r0-r12,sp-nm, lr-nm, pc, cpsr, spsr-nm

Undefined mode (nm)
Bank – 18 regs

r0-r12,sp-im, lr-im, pc, cpsr, spsr-im

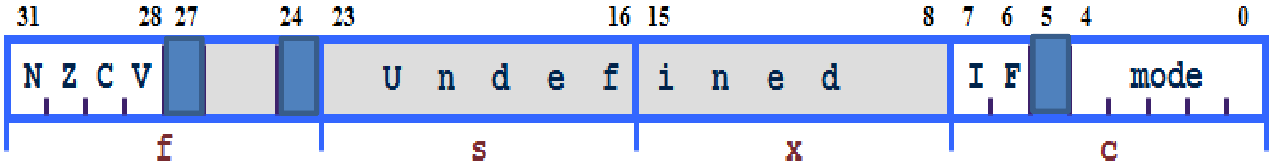
IRQ mode (im)
Bank – 18 regs

r0-r12,sp-am, lr-am, pc, cpsr, spsr-am

Abort mode (am)
Bank – 18 regs

6 Register Banks

Program Status Registers



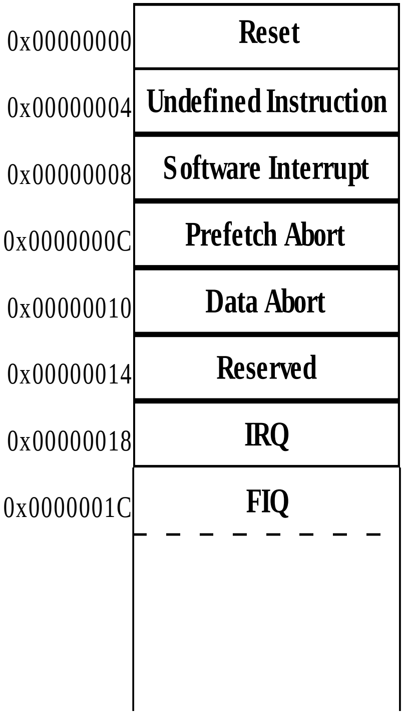
- Condition code flags
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed

- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- Mode bits
  - Specify the processor mode

Mode Encoding	
Value	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10100	Abort
10101	Undef
11111	System

Exception Handling and the Vector Table

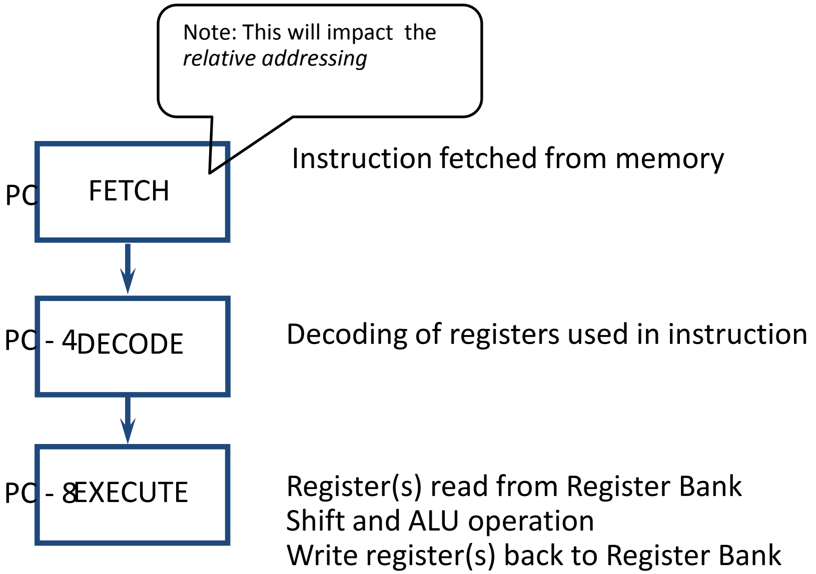
- When an exception occurs, the core:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
  - Maps in appropriate banked registers
  - Stores the “*return address*” in LR_<mode>
  - Sets PC to vector address
    - For instance for *Undefined instruction* PC is set to 0x00000008
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>



0x00000000	Reset
0x00000004	Undefined Instruction
0x00000008	Software Interrupt
0x0000000C	Prefetch Abort
0x00000010	Data Abort
0x00000014	Reserved
0x00000018	IRQ
0x0000001C	FIQ

The Instruction Pipeline

- In order to increase the *instruction throughput* ARM uses a 3-stage pipeline



Note: This will impact the *relative addressing*
PC
FETCH
Instruction fetched from memory
PC - 4
DECODE
Decoding of registers used in instruction
PC - 8
EXECUTE
Register(s) read from Register Bank
Shift and ALU operation
Write register(s) back to Register Bank

- PC points to the instruction being fetched, rather than pointing to the instruction being executed.

Branching Instruction with Offset

- The ARM supports B(ranch) instructions with *offset address* (rather than *absolute address*)
- The *target address* to branch is computed at execution using an *offset (actual offset/4)* & *Program Counter (PC) Value:*

  target address

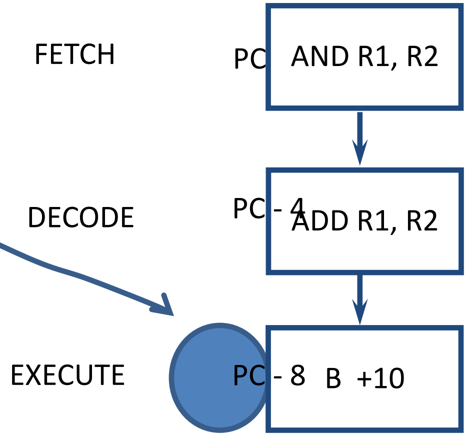
  = PC + *offset *4;*

  = *(branch instruction address* + 8) + (*offset* << 2)
- Similarly the *offset address* can be computed from *target* and *branch instruction addresses* as follows:

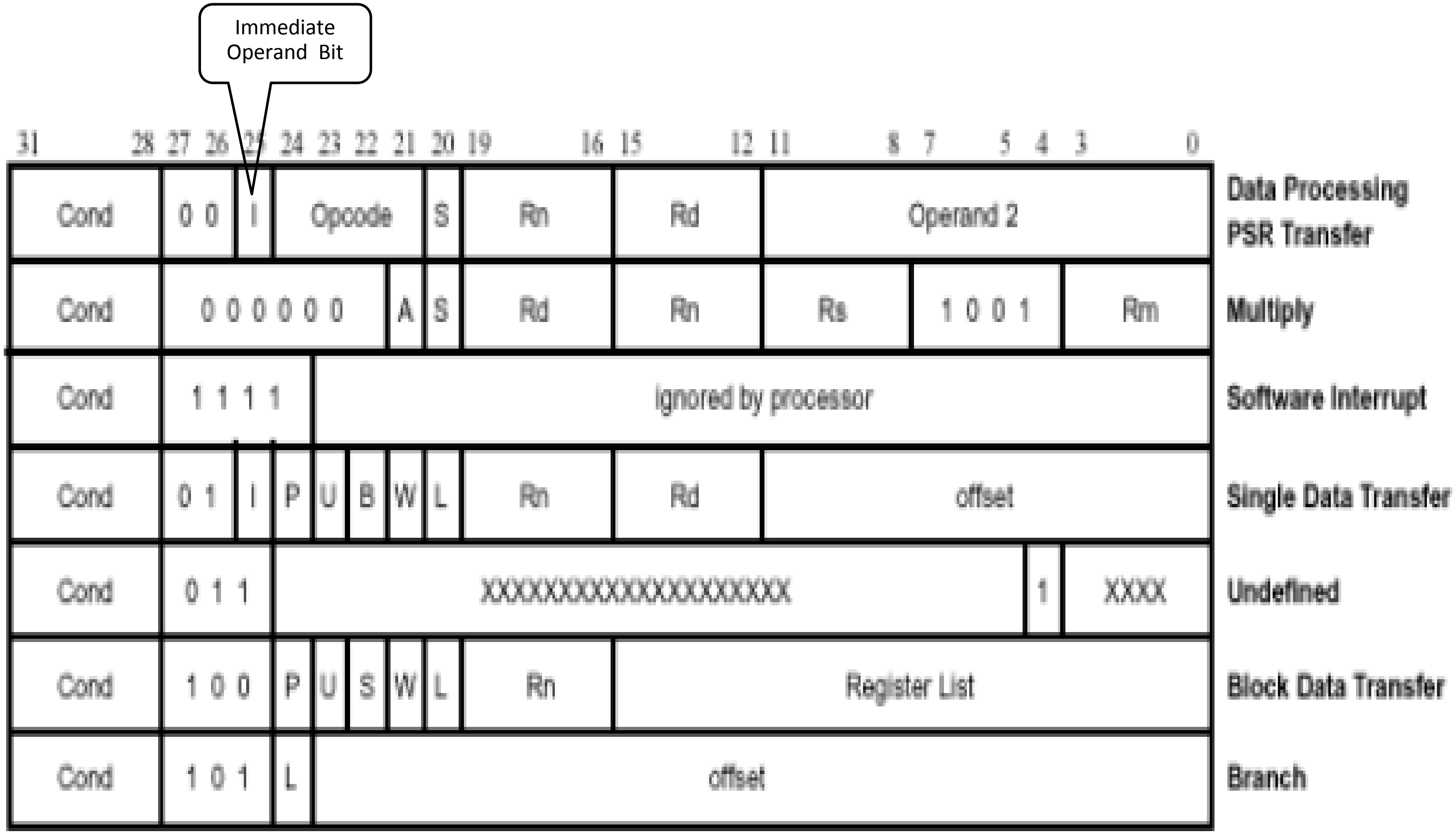
  Offset

  = (*target address* – PC) << 2

  = (*target address* - (*branch instruction address* +8))
  >> 2



Instruction Encoding for Some ARM Instruction Types



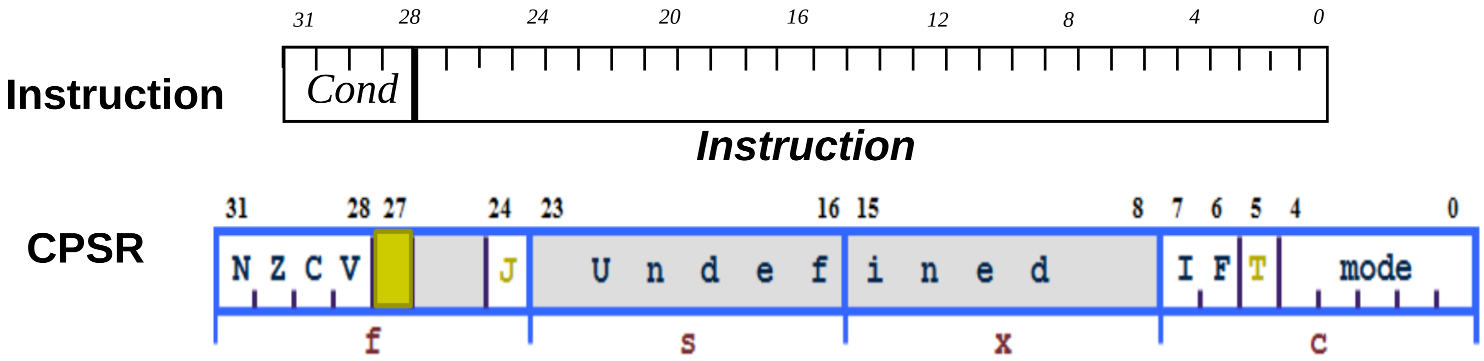
Immediate Operand Bit

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 8	7 5	4	3 0	
Cond	0 0	I	Opcode				S	Rn	Rd	Operand 2				Data Processing PSR Transfer
Cond	0 0 0 0 0 0					A	S	Rd	Rn	Rs	1 0 0 1		Rm	Multiply
Cond	1 1 1 1			ignored by processor										Software Interrupt
Cond	0 1	I	P	U	B	W	L	Rn	Rd	offset				Single Data Transfer
Cond	0 1 1		XXXXXXXXXXXXXXXXXXXX									1	XXXX	Undefined
Cond	1 0 0		P	U	S	W	L	Rn	Register List					Block Data Transfer
Cond	1 0 1		L	offset										Branch

Conditional Execution

- Most processors allow only branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - All instructions contain a condition field which determines whether the CPU will execute them.
  - Non-executed instructions soak up 1 cycle.
    - Still have to complete cycle so as to allow fetching and decoding of following instructions.
- This removes the need for many branches, which stall the pipeline (3 cycles to refill).
  - Allows very dense in-line code, without branches.
  - The Time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

The Condition Field



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear or N clear and V set (<, or =)

1110 = AL - always

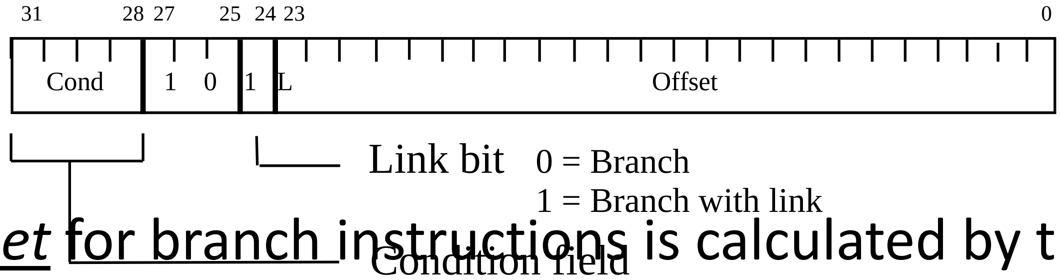
1111 = NV - reserved.

Using and updating the Condition Field

- To execute an instruction conditionally, simply postfix it with the appropriate condition:
  - For example an add instruction takes the form:
    - ADD r0,r1,r2   ; r0 = r1 + r2 (ADD**AL**)
  - To execute this only if the zero flag is set:
    - ADD**EQ** r0,r1,r2   ; If zero flag set then r0 = r1 + r2
- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect).
- To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
  - For example to add two numbers and set the condition flags:
    - ADD**S** r0,r1,r2   ;
    - r0 = r1 + r2 ; ... and set flags

Branch instruction

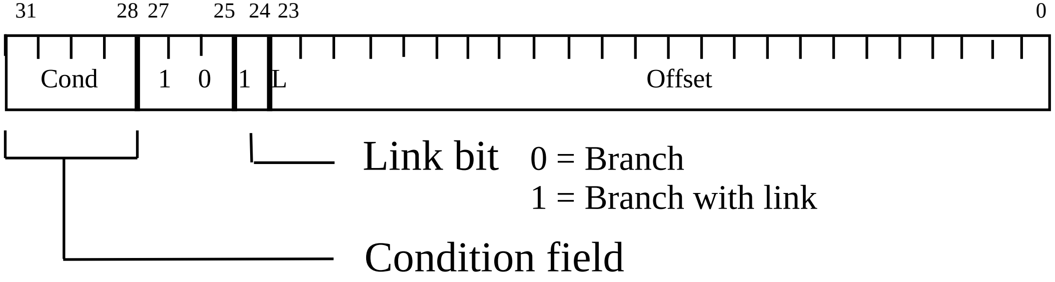
- `B{<cond>} label ;`   <u>`versus Branch with Link`</u>

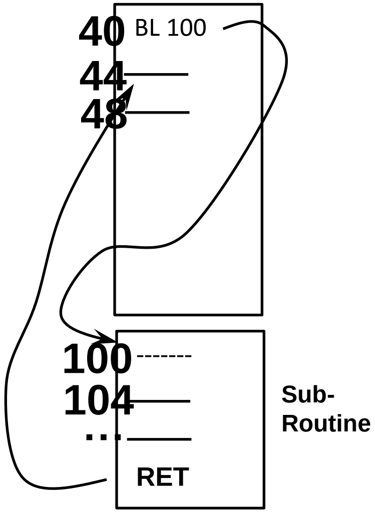


- The <u>*offset*</u> for branch instructions is calculated by the assembler:
  - By taking the <u>difference between the target instruction and the branch instruction address minus 8</u> (to allow for the pipeline). Then right shift the result by 2 positions.
  - This gives a range of ± 32 Mbytes.

Branch instruction with Link

- Branch with Link : `BL{<cond>} sub_routine_label`





Data processing Instructions

- Largest family of ARM instructions, all sharing the same instruction format.
- Contains:
  - Arithmetic operations
  - Comparisons (no results - just set condition codes)
  - Logical operations
  - Data movement between registers
- Remember, this is a load / store architecture
  - These instruction only work on registers, ***NOT*** memory.
- They each perform a specific operation on one or two operands.
  - First operand always a register - Rn
  - Second operand sent to the ALU via barrel shifter.
- We will examine the barrel shifter shortly.

Arithmetic Operations

- Operations are:
  - ADD operand1 + operand2 ; operand1 is Rn
  - ADC operand1 + operand2 + carry
  - SUB operand1 - operand2
  - SBC operand1 - operand2 + carry -1
  - RSB operand2 - operand1
  - RSC operand2 - operand1 + carry - 1
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2 ; Rd is destination register
- Examples
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Comparisons

- The only effect of the comparisons is to
  - ***UPDATE THE CONDITION FLAGS*****.** Thus no need to set S bit.
- Operations are:
  - CMP operand1 - operand2, but result not written
  - CMN operand1 + operand2, but result not written
  - TST operand1 AND operand2, but result not written
  - TEQ operand1 EOR operand2, but result not written
- Syntax:
  - <Operation>{<cond>} Rn, Operand2
- Examples:
  - CMP r0, r1
  - TSTEQ r2, #5

Logical Operations

- Operations are:
  - AND operand1 AND operand2
  - EOR operand1 EOR operand2
  - ORR operand1 OR operand2
  - BIC operand1 AND NOT operand2 [ie bit clear]
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Examples:
  - AND r0, r1, r2
  - BICEQ r2, r3, #7
  - EORS r1,r3,r0

Data Movement

- Operations are:
  - MOV operand2
  - MVN NOT operand2

  Note that these make no use of operand1.
- Syntax:
  - <Operation>{<cond>}{S} Rd, Operand2
- Examples:
  - MOV r0, r1
  - MOVS r2, #10
  - MVNEQ r1,#0

The Barrel Shifter

- The ARM doesn’t have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.
- Barrel shifter performs shift operations faster than regular shifter
- So what operations does the barrel shifter support?

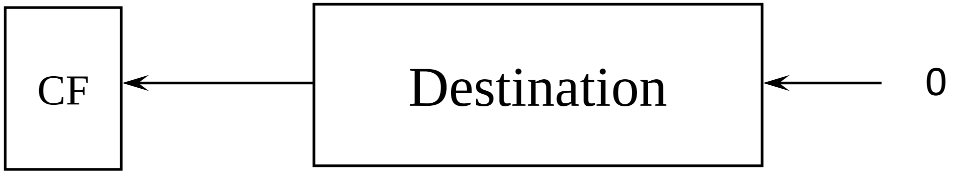
Barrel Shifter - Left Shift

- Shifts left by the specified amount (multiplies by powers of two) e.g.

  LSL #5 = multiply by 32

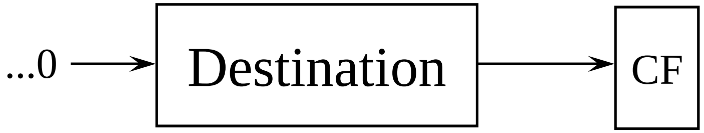
- CF is Carry Flag bit

Logical Shift Left (LSL)



Barrel Shifter - Right Shifts

Logical Shift Right

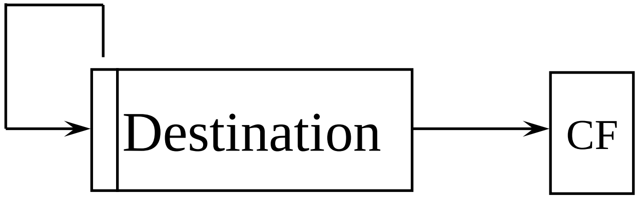


...0

Destination

CF

Arithmetic Shift Right



Destination

CF

Sign bit shifted in

Logical Shift Right

- Shifts right by the specified amount (divides by powers of two) e.g.

  LSR #5 = divide by 32

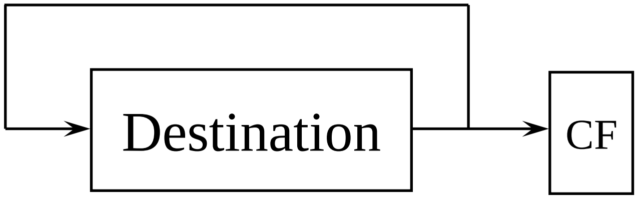
Arithmetic Shift Right

- Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g.

  ASR #5 = divide by 32

Barrel Shifter - Rotations

Rotate Right



Rotate Right (ROR)

- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

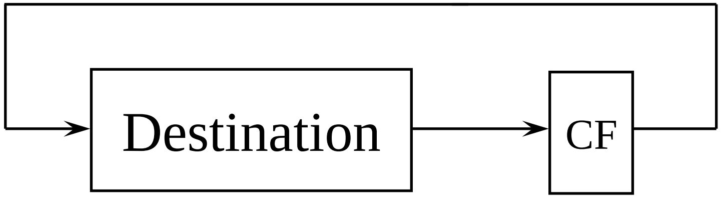
  e.g. ROR #4

- Note the last bit rotated is also used as the Carry Out.

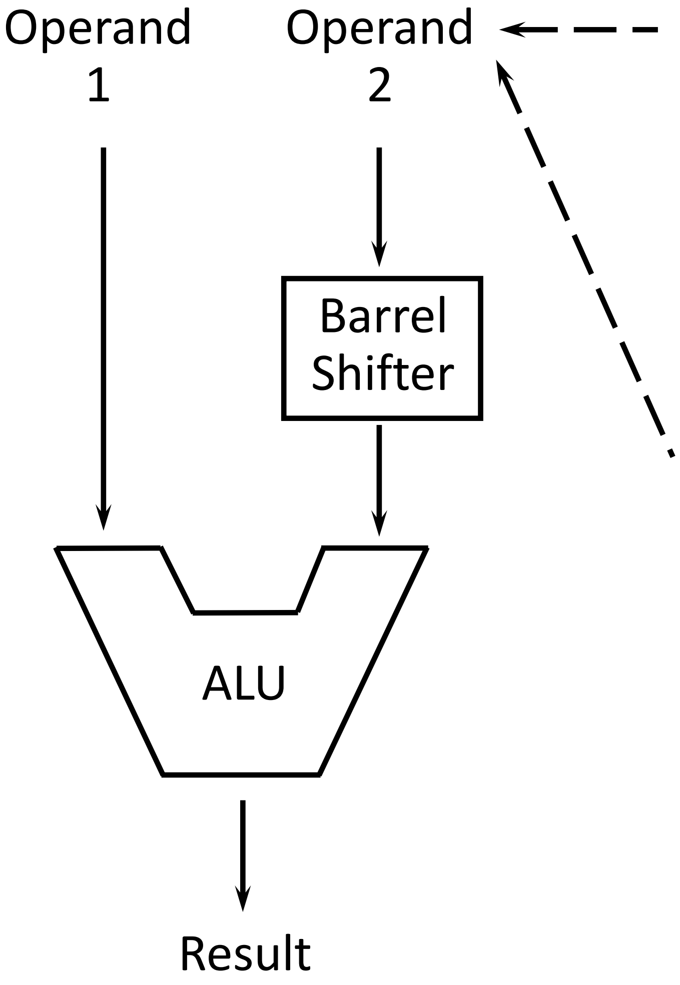
Rotate Right Extended (RRX)

- This operation uses the CPSR C flag as a 33rd bit.
- Rotates right by 1 bit. Encoded as ROR #0.

Rotate Right through Carry



Using the Barrel Shifter: The Second Operand



- Register, optionally with shift operation applied.
- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.

* Immediate value

- 8 bit number
- Can be rotated right through an even number of positions.
- Assembler will calculate rotate for you from constant.

Second Operand: Shifted Register

- The amount by which the register is to be shifted is contained in either:
  - the immediate 5-bit field in the instruction
    - *NO OVERHEAD*
    - Shift is done for free - executes in single cycle.
  - the bottom byte of a register (not PC)
    - Then takes extra cycle to execute
    - ARM doesn’t have enough read ports to read 3 registers at once.
    - Then same as on other processors where shift is separate instruction.
- If no shift is specified then a default shift is applied: LSL #0
  - i.e. barrel shifter has no effect on value in register.

Load / Store Instructions

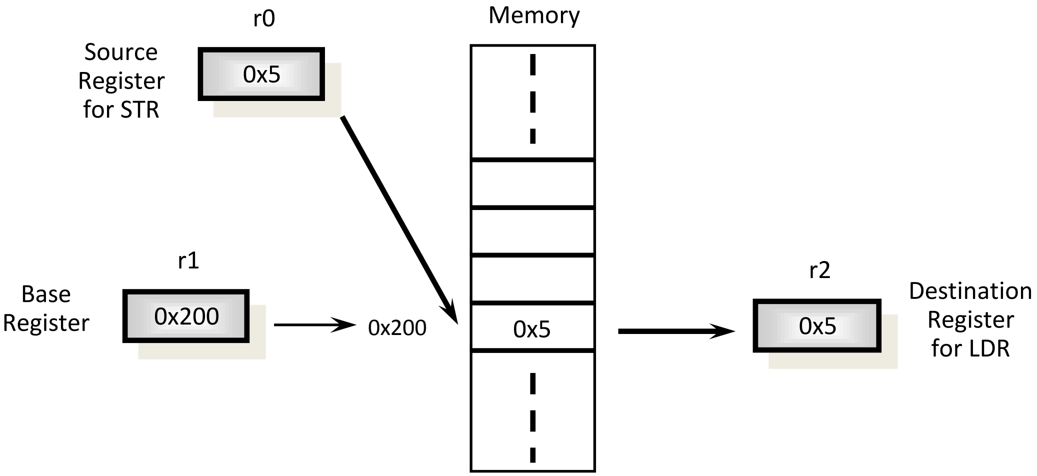
- The ARM is a Load / Store Architecture:
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
- This might sound inefficient, but in practice isn't:
  - Load data values from memory into registers.
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
  - Store results from registers out to memory.
- The ARM has three sets of instructions which interact with main memory. These are:
  - Single register data transfer (LDR / STR).
  - Block data transfer (LDM/STM).
  - Single Data Swap (SWP).

Single register data transfer

- The basic load and store instructions are:
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- ARM Architecture Version 4 also adds support for halfwords and signed data.
  - Load and Store Halfword
    - LDRH / STRH
  - Load Signed Byte or Halfword - load value and sign extend it to 32 bits.
    - LDRSB / LDRSH
- All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.
  - e.g. LDREQB
- Syntax:
  - <LDR|STR>{<cond>}{<size>} Rd, <address>

Load and Store

- The memory location to be accessed is held in a base register
  - STR r0, [r1] ; Store contents of r0 to location pointed to
    ; by contents of r1.
  - LDR r2, [r1] ; Load r2 with contents of memory location
    ; pointed to by contents of r1.

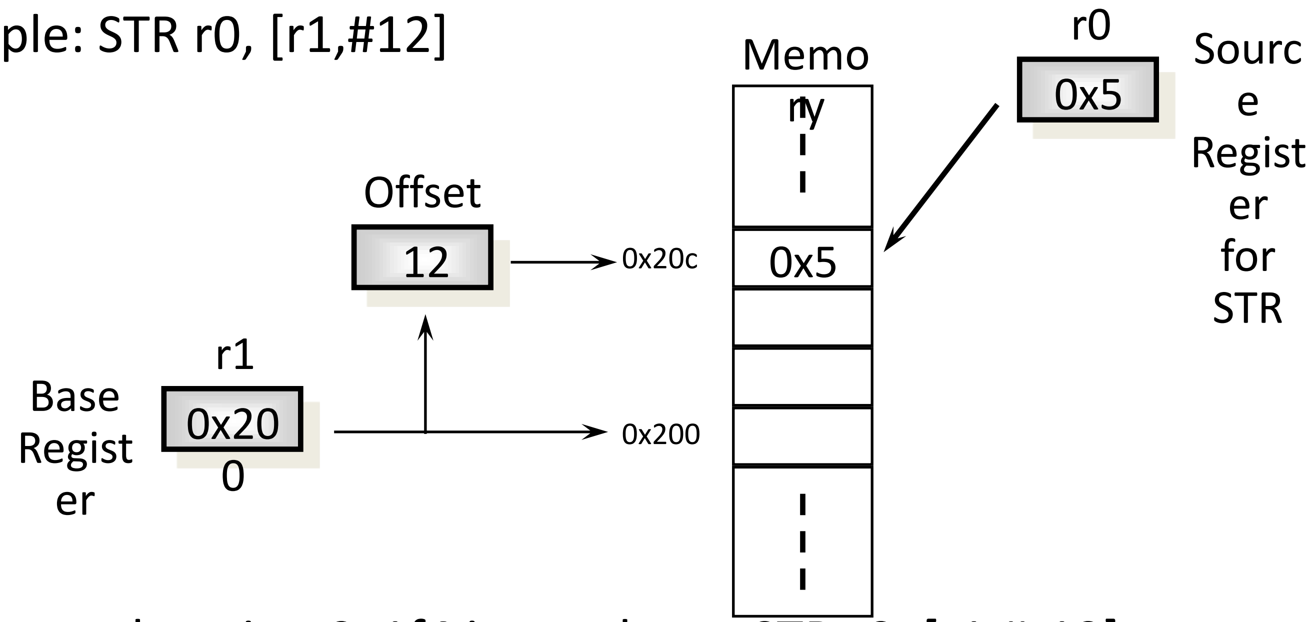


Load and Store with Indexing

- As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.
- This offset can be
  - An unsigned 12bit immediate value (ie 0 - 4095 bytes).
  - A register, optionally shifted by an immediate value
- This can be either added or subtracted from the base register:
  - Prefix the offset value or register with '**+**' (default) or '**-**'.
- This offset can be applied:
  - before the transfer is made: ***Pre-indexed addressing***
    - optionally *auto-incrementing* the base register, by postfixing the instruction with an '**!**'.
  - after the transfer is made: ***Post-indexed addressing***
    - causing the base register to be *auto-incremented*.

Load and Store Word Pre-indexed Addressing

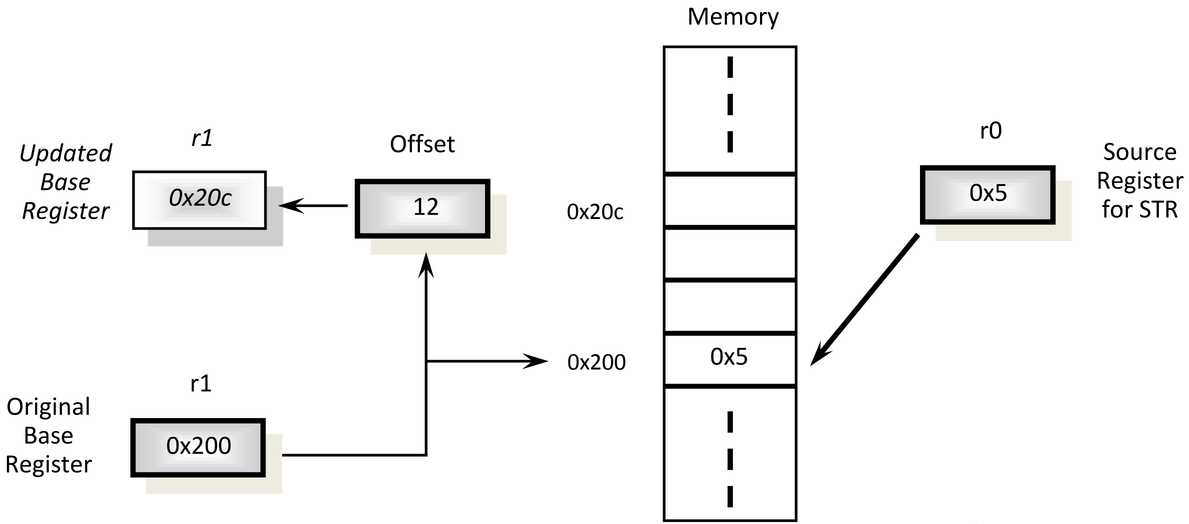
- Example: STR r0, [r1,#12]



- To store to location 0x1f4 instead use: STR r0, [r1,#-12]
- To auto-increment base pointer to 0x20c use: STR r0, [r1, #4]!
- If r2 contains 3, access 0x20c by multiplying this by 4:
  - STR r0, [r1, r2, LSL #2]

Load and Store Word Post-indexed Addressing

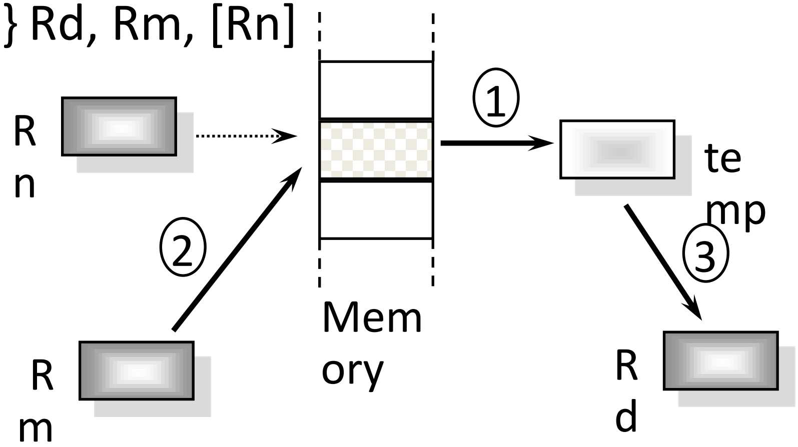
- Example: STR r0, [r1], #12



- To auto-increment the base register to location 0x1f4 instead use:
  - STR r0, [r1], #-12
- If r2 contains 3, auto-increment base register to 0x20c by multiplying this by 4:
  - STR r0, [r1], r2, LSL #2

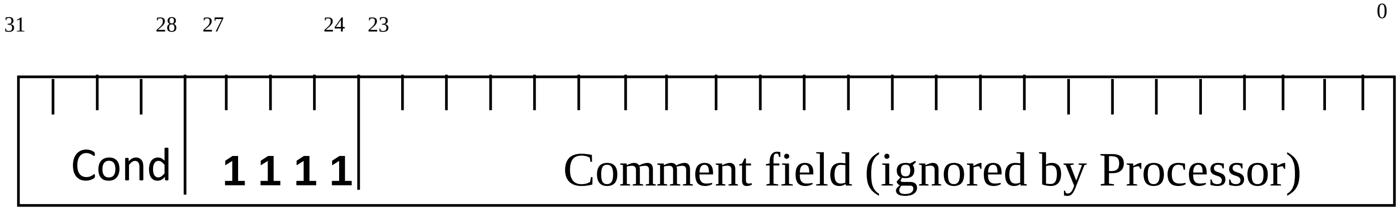
Swap Instructions

- Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.
- Syntax:
  - – SWP{<cond>}{B} Rd, Rm, [Rn]



- Thus to implement an actual swap of contents make Rd = Rm.
- The compiler cannot produce this instruction.

Software Interrupt (SWI)



31 28 27 24 23 0

Cond 1 1 1 1 Comment field (ignored by Processor)

- In effect, a SWI is a user-defined instruction.
- It causes an exception trap to the SWI hardware vector (thus causing a change to supervisor mode, plus the associated state saving), thus causing the SWI exception handler to be called.
- The handler can then examine the comment field of the instruction to decide what operation has been requested.
- By making use of the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- See Exception Handling Module for further details.