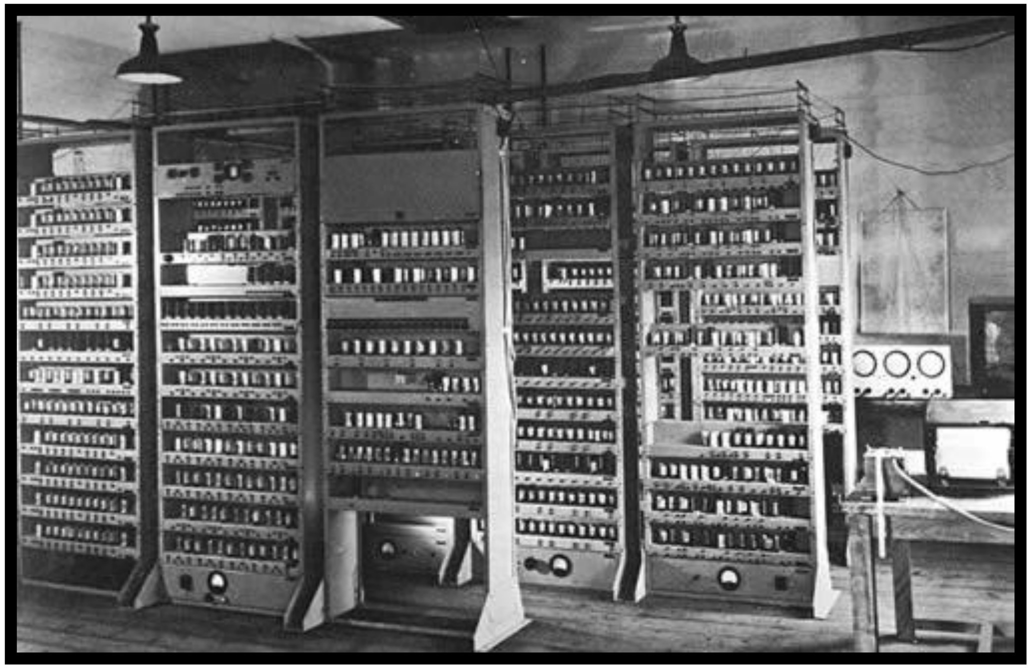
Introduction to Computer Architecture
ISE1 & EE2
Autumn 2010 - Dr. Tom Clarke

- EDSAC – first stored program computer
- Cambridge 1949
- 650 instructions per second.
- 1024 17-bit words of memory in mercury ultrasonic delay lines.
- 3000 valves, 12 kW power consumption, occupied a room 5m by 4m.
- Early use to solve problems in meteorology, genetics and X-ray crystallography.

60 years ago...



Electronic Delay Storage Automatic Computer

... and Now

- ARM7 core – up to 130 million instructions per second. 1995-2011.
- ARM7 core in many variations is most successful embedded processor today.
- Picture shows LPC2124 microcontroller which includes ARM7 core + RAM, ROM integrated peripherals.
  - The complete microcontroller is the square chip in the middle
  - 128K X 32 bit words flash RAM
  - 10mW/Mhz clock
- Original ARM design:
  - Steve Furber, Acorn Risc Machines, Cambridge, 1985

ARM7 CPU – LPC-2124 microcontroller

Why study this course?

- Computers are the single most significant technological advance of the 20th century
- The course makes a bridge between digital electronics (hardware) & high level languages (software)
- You will learn the basic concepts needed to understand how all computers work

ISE1 / EE2 Principles of Computers

- Course materials based on two books:
  - "Computer Organization & Design" 2nd edition, Patterson & Hennessy 1998 (around £30 new - £15 2nd hand via Amazon)
    - Covers most topics on this course
    - V. Useful for ISE – also used in 2nd Year.
  - "ARM System-on-Chip Architecture", Steve Furber, 2000 (around £25)
    - Best book on ARM processor
- Administration issues:
  - Around 15 lectures on this part of the course
  - Problem sheets for programming labs (EE) / tutorials (ISE)
  - **Coursework: will be programming exercises + on-line tests**
  - Course website: **www2.ee.ic.ac.uk/t.clarke/arch/**

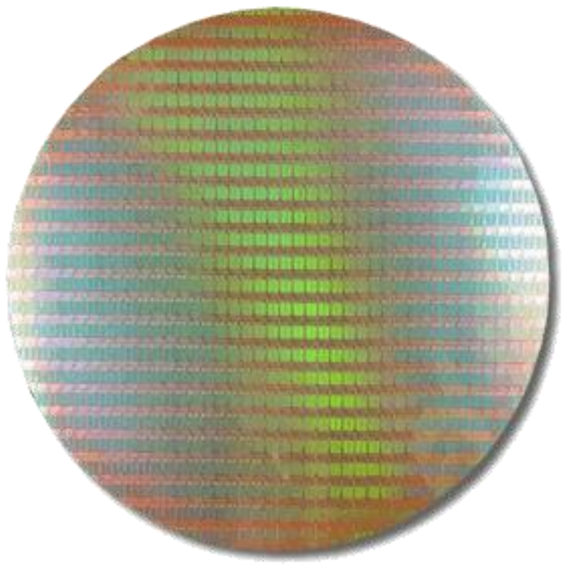
What will you be learning?

- **Part 1 - Introduction**
  - What is a computer?
  - What is **I**nstruction **S**et **A**rchitecture (ISA)
  - How is an ISA implemented in hardware? (at an abstract level)
  - MU0 – a simple computer
  - Instruction formats
  - Memory addressing
- **Part 2 – ARM Assembly Language**
  - **Assembly language** - large part of the course
  - Details of the ARM ISA
  - How to use arithmetic & logical data operations
  - How are machine instructions used to implement programs?
- **Part 3 – Hardware Topics**
  - Computer interfacing with peripherals
    - interrupts & DMA
  - ARM Organisation
    - Pipelining
    - Throughput
  - Memory hierarchy

Part 1

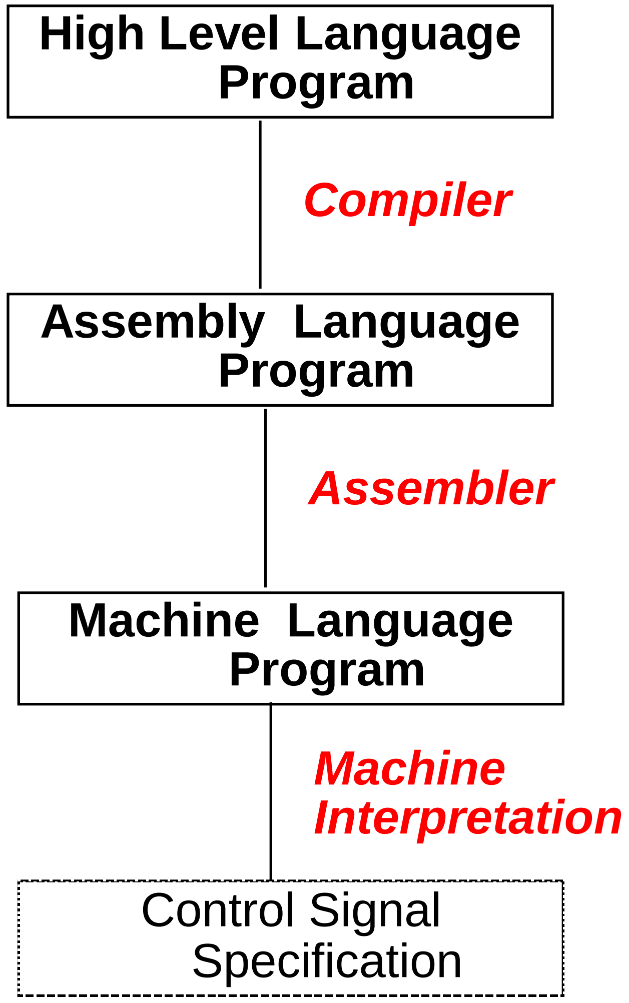
Introduction to Machine Architecture

EDSAC mercury delay line



300mm wafer, used for production of Intel's "Dothan" Pentium mobile CPUs

Levels of representation in computers

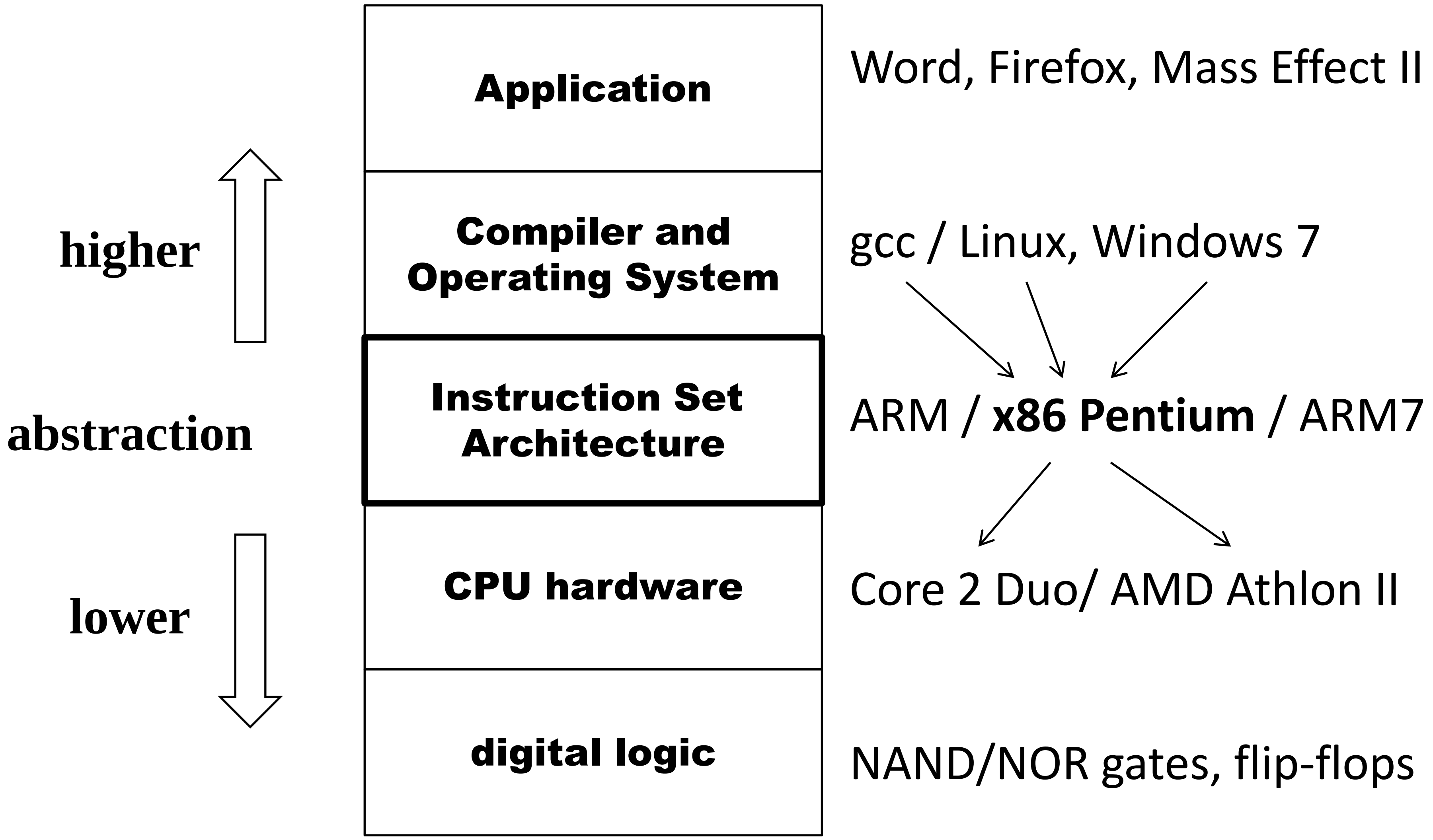

High Level Language Program
Compiler
Assembly Language Program
Assembler
Machine Language Program
Machine Interpretation
Control Signal Specification

```
temp = x;
x = y;
y = temp;
```

```
LDR     R0,  [PC, #0x0014]
LDR     R1,  [R0]
LDR     R2,  [R0,#0x0004]
STR     R2,  [R0]
STR     R1,  [R0,#0x0004]
```

```
1110 0101 1001 1111 0000 0000 0001 0100
1110 0101 1001 0000 0001 0000 0000 0000
1110 0101 1001 0000 0000 0010 0000 0100
1110 0101 1000 0000 0010 0000 0000 0000
1110 0101 1000 0000 0001 0000 0000 0100
```

What is "Computer Architecture" ?



higher
abstraction
lower
Application
Compiler and Operating System
Instruction Set Architecture
CPU hardware
digital logic
Word, Firefox, Mass Effect II
gcc / Linux, Windows 7
ARM / **x86 Pentium** / ARM7
Core 2 Duo/ AMD Athlon II
NAND/NOR gates, flip-flops

What is “Instruction Set Architecture (ISA)”?

- “. . . the attributes of a [computing] system **as seen by the programmer**, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation.”
  - Amdahl, Blaaw, and Brooks, 1964
- **What** the computer does, not **how** it does it!

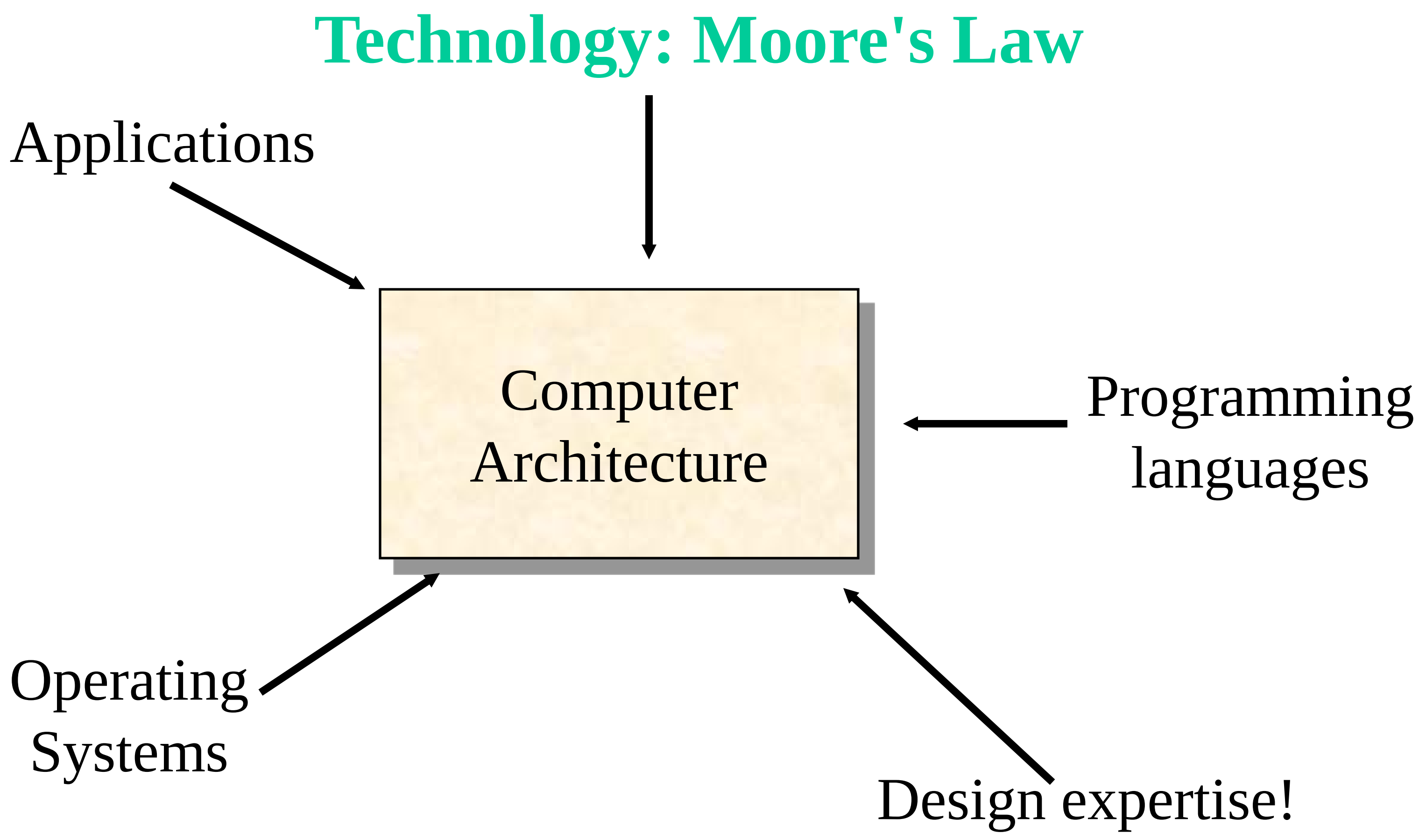
ISA includes:-

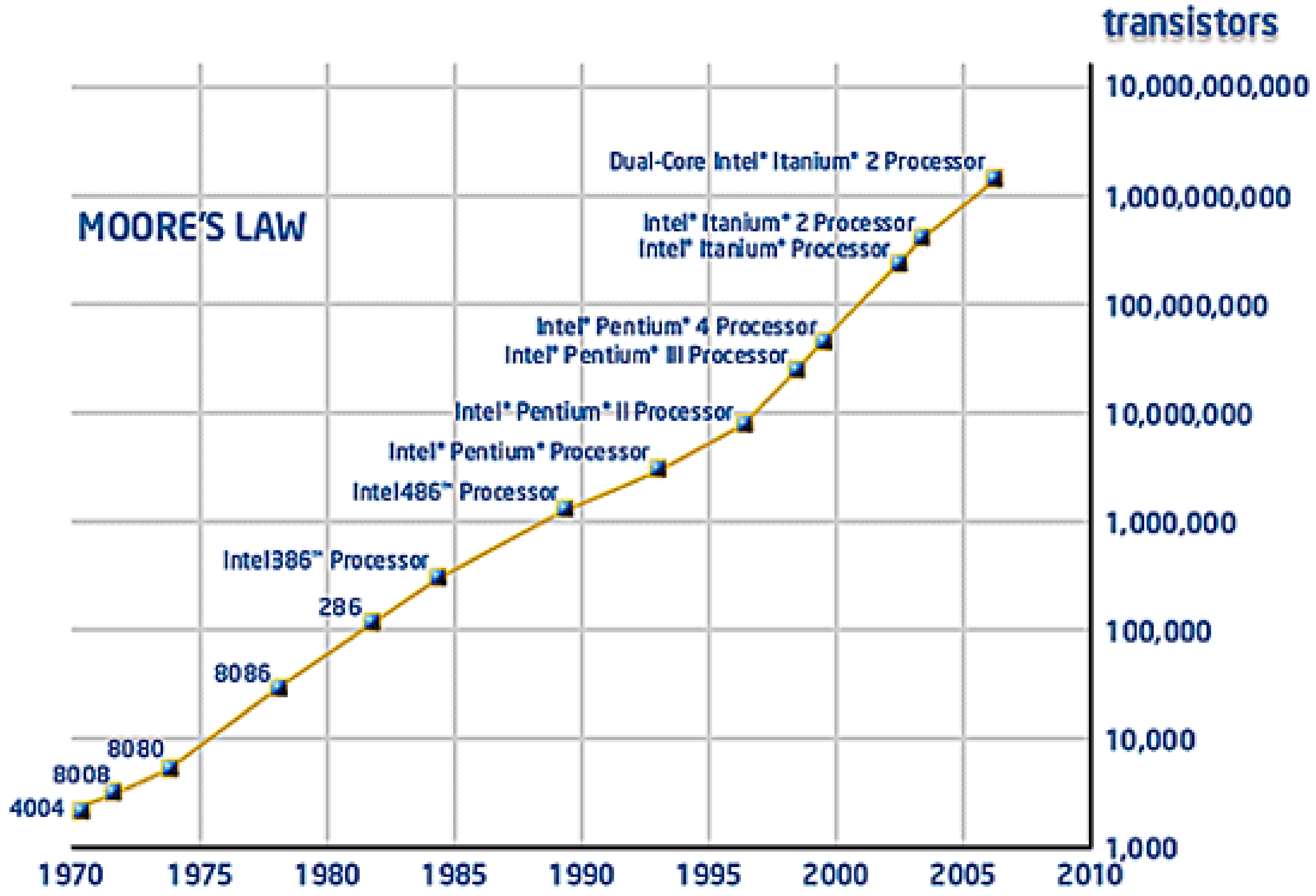
- **Instruction (or Operation Code) Set**
  - Data Types & Data Structures: Encodings & Representations
  - Instruction Formats
- Organization of Programmable Storage (main memory etc)
- Modes of Addressing and Accessing **Data Items** and **Instructions**
- Behaviour on Exceptional Conditions (e.g. hardware divide by 0)

Why is ISA important

- 8086/pentium ISA
  - Allows operating systems and applications to work seamlessly across different computer architectures:
    - Pentium
    - Core 2 duo
    - Athlon
    - Phenom
- ARM ISA
  - Supports highly optimising compiler & operating system software for embedded applications
  - ARM cores can be licensed & sold by different vendors
- Key advantage: **different implementations of the same ISA can all run identical software**

Factors influencing computer architectures

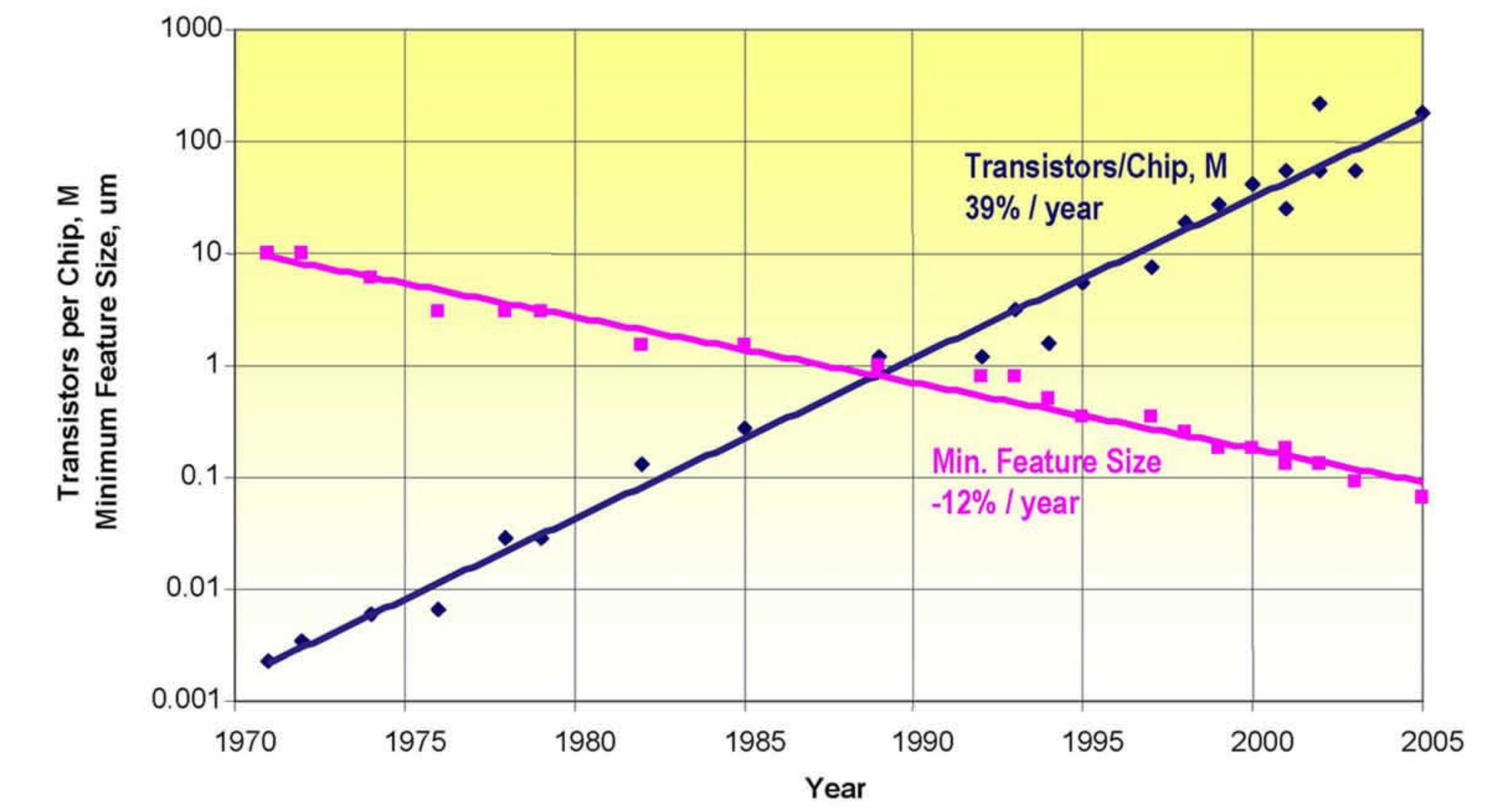


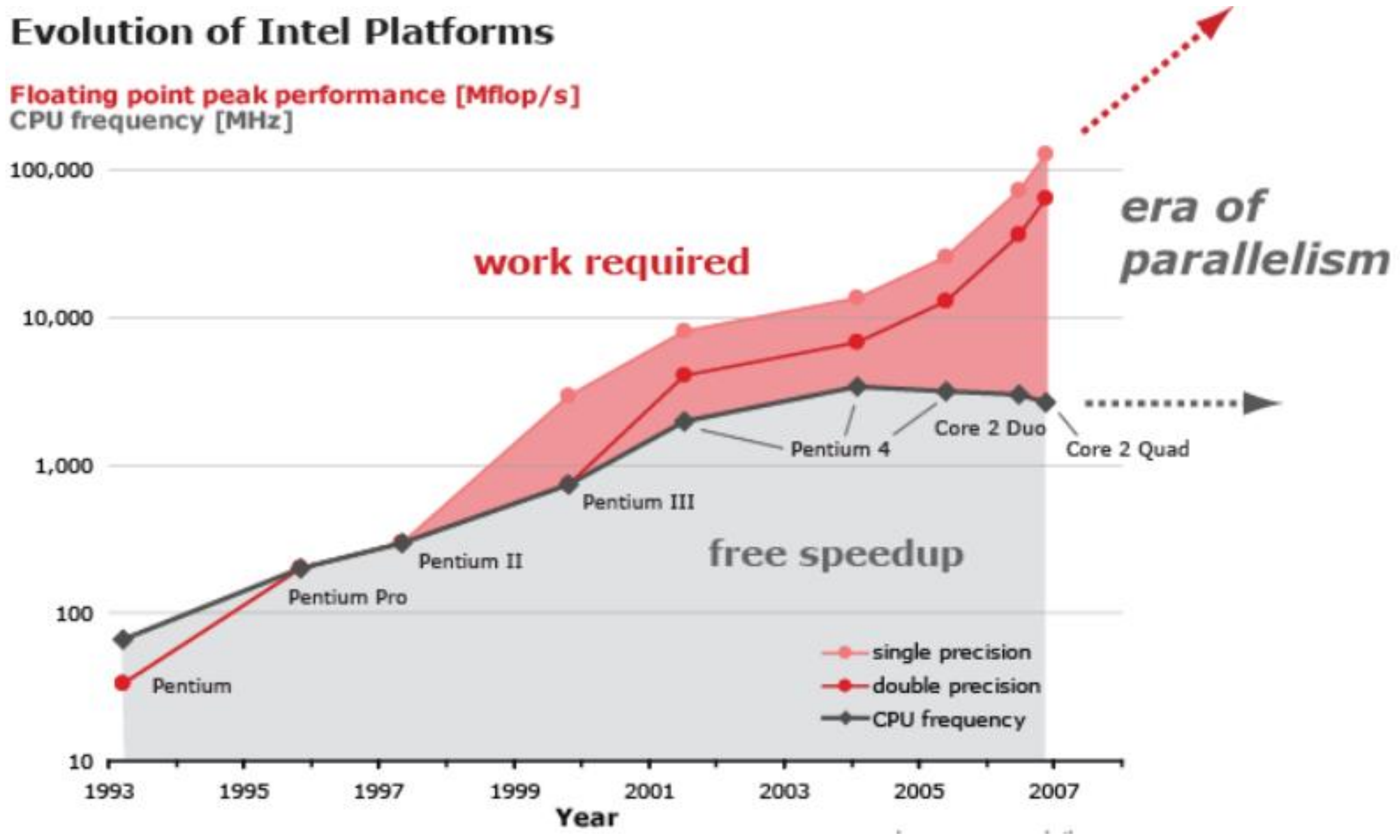

MOORE'S LAW
transistors
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
Dual-Core Intel® Itanium® 2 Processor
Intel® Itanium® 2 Processor
Intel® Itanium® Processor
Intel® Pentium® 4 Processor
Intel® Pentium® III Processor
Intel® Pentium® II Processor
Intel® Pentium® Processor
Intel486™ Processor
Intel386™ Processor
286
8086
8080
8008
4004
1970
1975
1980
1985
1990
1995
2000
2005
2010

Semiconductor feature size drives technology

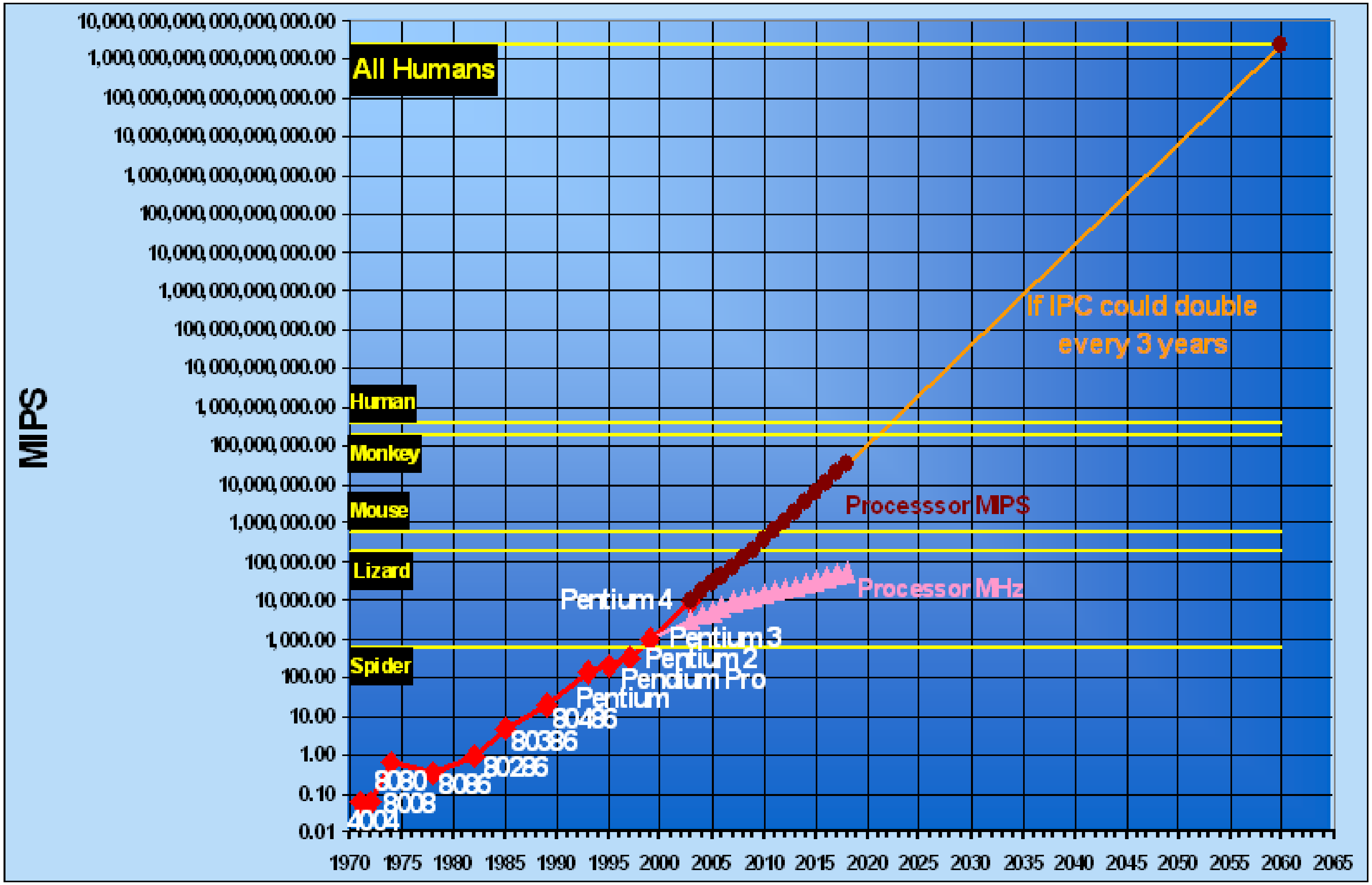
- **Transistors/cm^2** scales as (feature size)$^{-2}$
  - 2005 100nm
  - 2011 32nm
- **Speed** scales as (feature size)$^{-1/2}$
  - Other factors limit speed at very small sizes

Note:
Chips get bigger with time
10%/year of transistor increase is from larger chip size




Evolution of Intel Platforms
Floating point peak performance [Mflop/s]
CPU frequency [MHz]
100,000
10,000
1,000
100
10
1993
1995
1997
1999
2001
2003
2005
2007
Year
work required
era of parallelism
free speedup
Pentium
Pentium Pro
Pentium II
Pentium III
Pentium 4
Core 2 Duo
Core 2 Quad
single precision
double precision
CPU frequency

Logic & memory relative performance

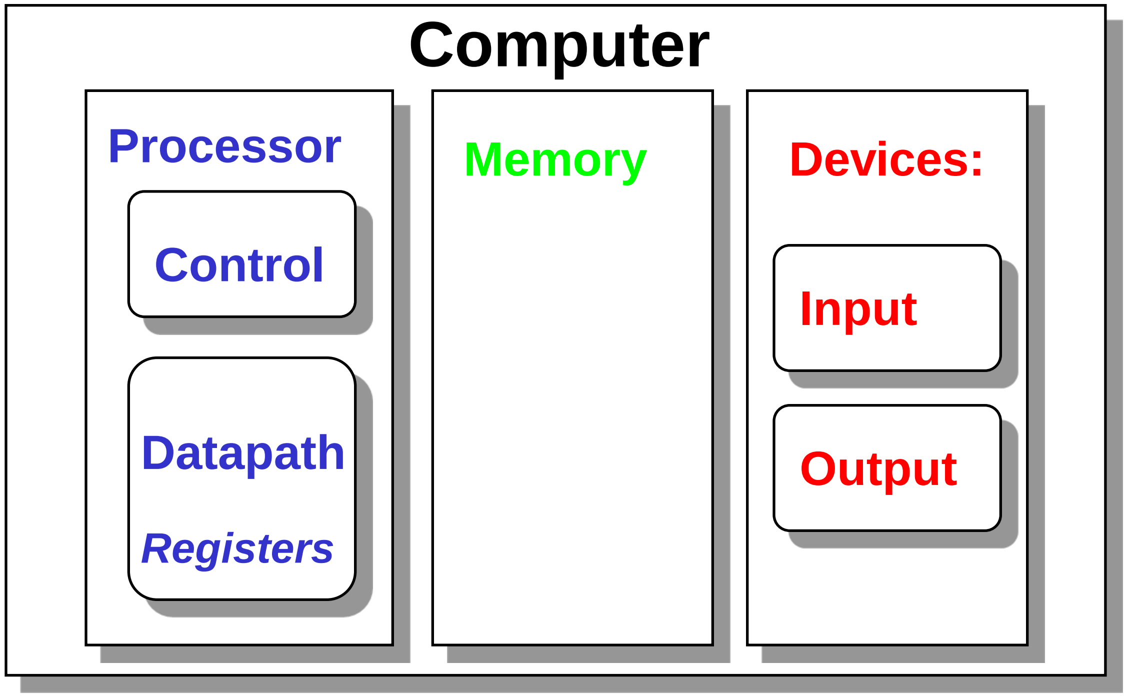


When will Moore's law end?

- **Technology**
  - Gordon Moore coined Moore's law in 1965
  - Number of transistors per chip roughly doubles every 2-3 years
  - In 2007 Moore said it could last for another 10-15 years
  - The technological limits on making transistors smaller are the size of atoms - in 2007 transistor gates were 5 atoms thick!
  - Use of high-k Hafnium in gates has pushed capability beyond previous 45nm limits
- **Economics**
  - Current generation CPUs (2011): 32nm
  - Globalfoundries' "Fab 8" in New York, 2012, 28nm, 5 years to build,
  - Intel's "Fab 42" Arizona, 2013, 14nm
  - Len Jelinek (chief analyst for at isuppli): 2014, around 20nm, will be end of economic return from new fabs. Moore's Law will stop.
  - Semiconductor manufacturers expect continued technological improvement in specific markets, e.g. non-volatile memory to replace hard disks

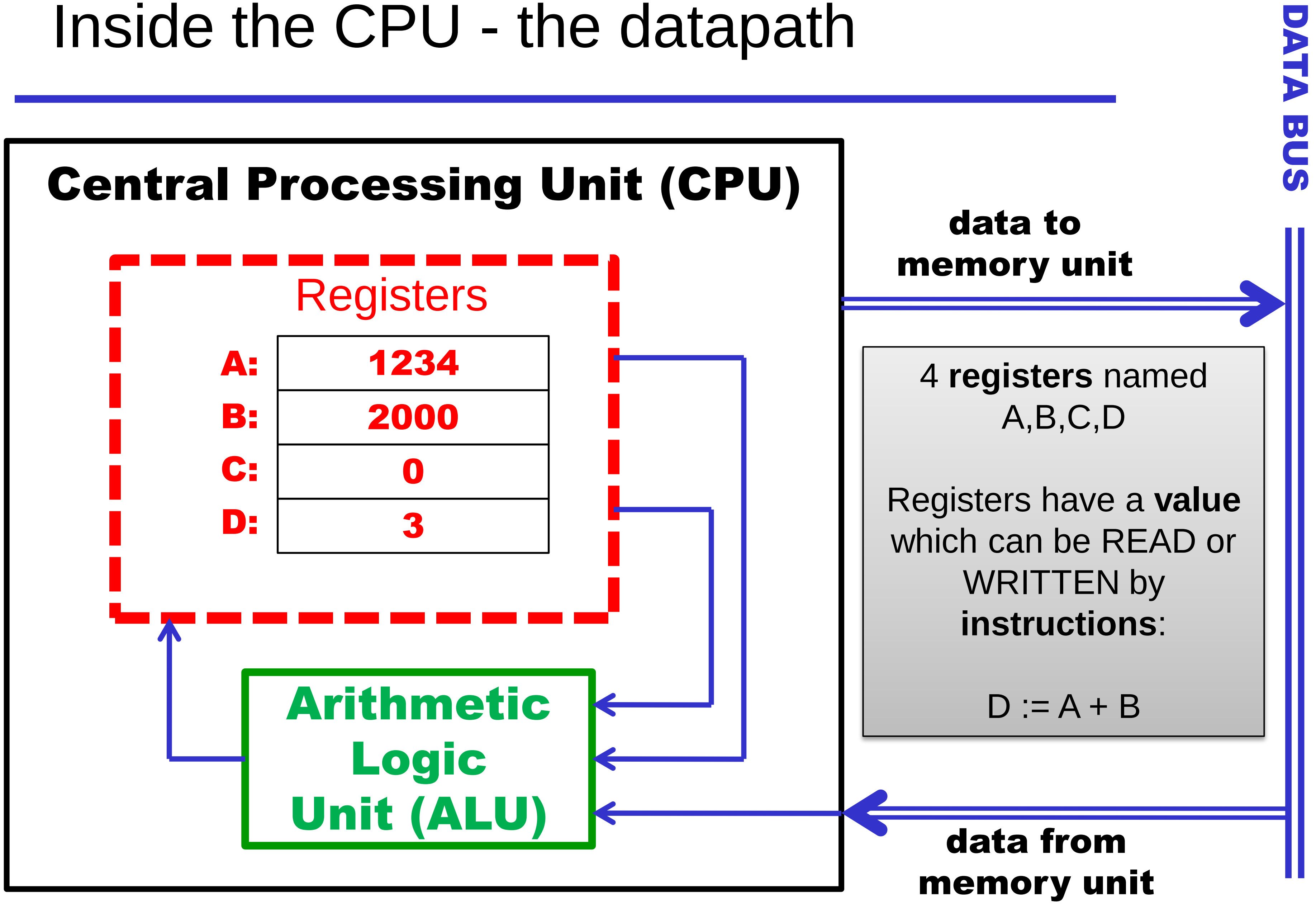
Internal Organisation of Computer

Processor aka CPU (Central Processing Unit)



- Major components of Typical Computer System
- Data is mostly stored in the computer memory separate from the Processor, however **registers** in the processor datapath can also store small amounts of data

Inside the CPU - the datapath



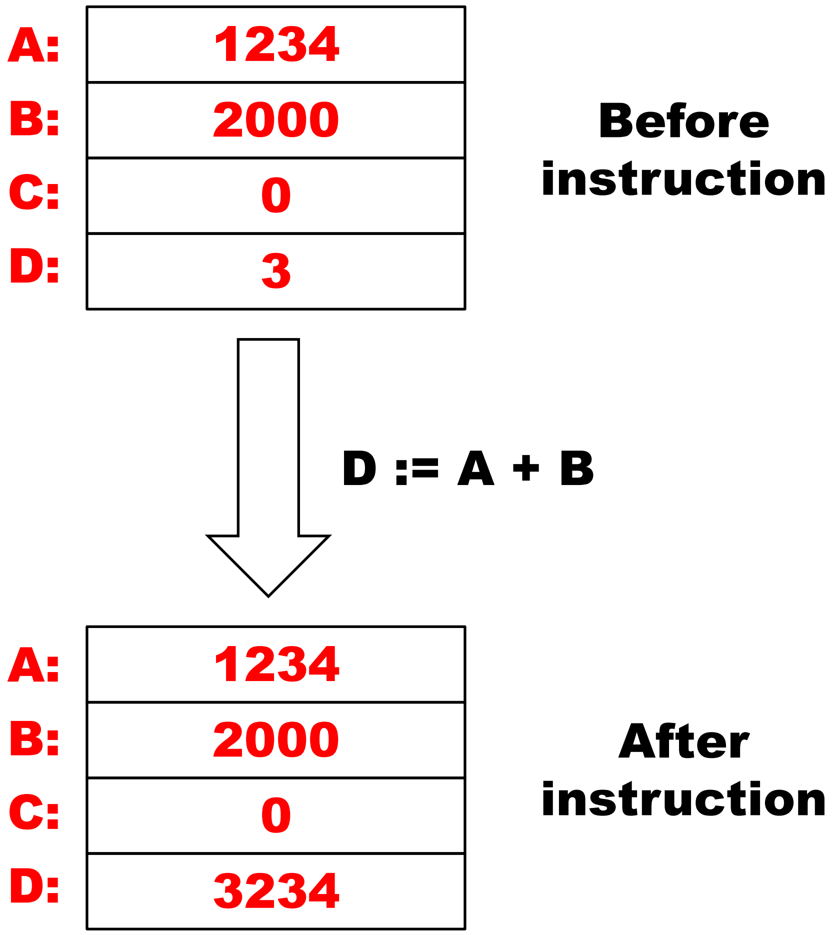
4 **registers** named A,B,C,D

Registers have a **value** which can be READ or WRITTEN by **instructions**:

D := A + B

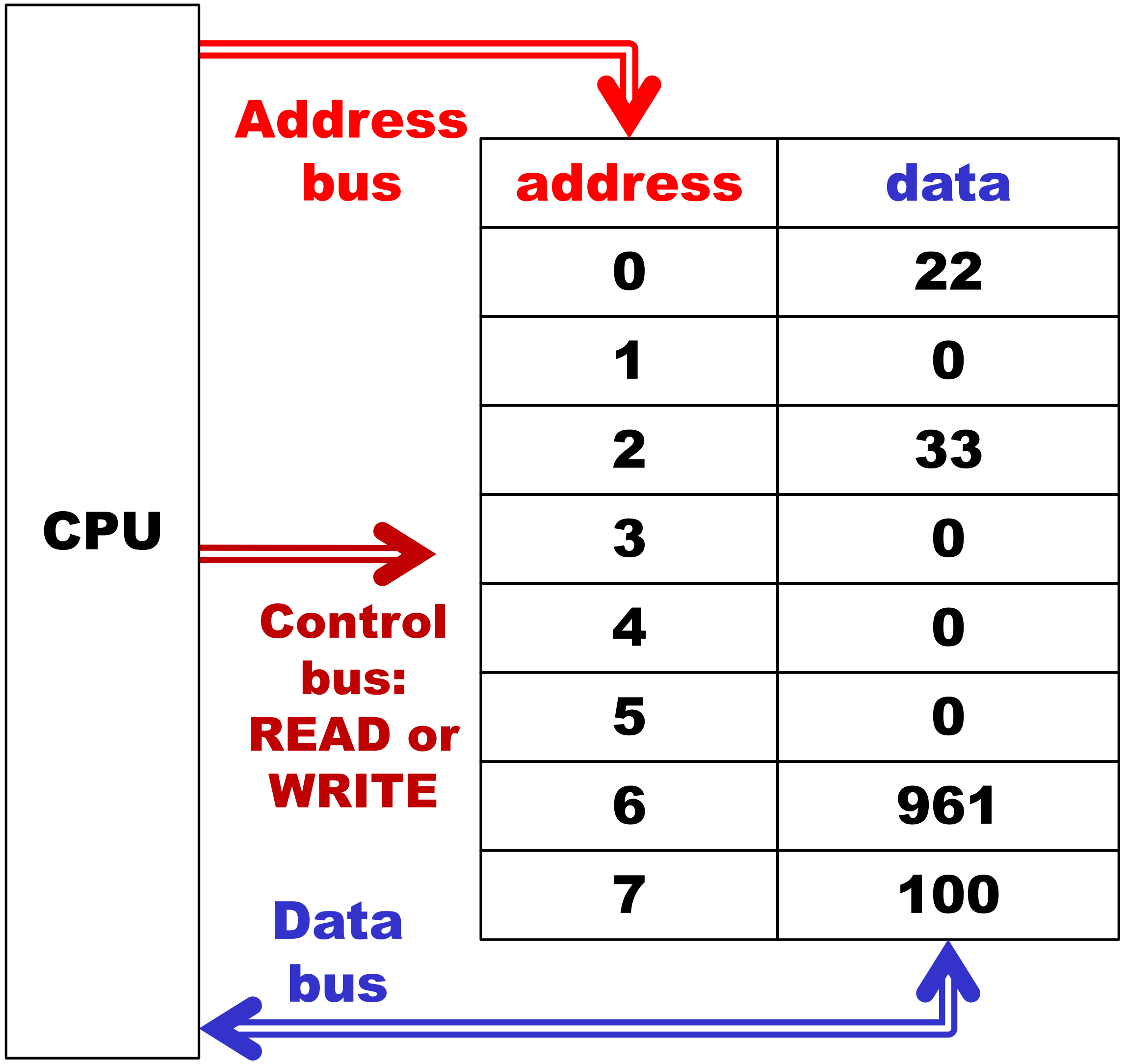
Instructions can move data between CPU registers

- **D := A + B**
- **READ** A (1234)
- **READ** B (2000)
  - NB order of reading A,B does not matter
- **WRITE** D (3234)



Inside the memory unit

- Memory contains 8 **locations** named 0-7
- Memory location numbers are called **addresses**
- A Location's **data** can be READ or WRITTEN
- **Address** is a number which specifies location to be READ or WRITTEN with data



address	data
0	22
1	0
2	33
3	0
4	0
5	0
6	961
7	100

Instructions can move data between memory & registers

Memory location with address 12 is called

mem[12]

A:	**1234=>961**
B:	**2000**

Instruction:
A := mem[6]
W R

READ mem[6] (961)
WRITE A (961)

Instruction:
mem[0] := B
W R

READ B (2000)
WRITE mem[0] (2000)

address	data
0	22=>2000
1	0
2	33
3	0
4	0
5	0
6	961
7	100

Summary

- Speed and density of computers is increasing exponentially with time: speed is near limit, density will reach limit some time. (15 years?).
- All computers consist of five components
  - Processor (CPU): (1) datapath and (2) control
  - (3) Memory
  - (4) Input devices and (5) Output devices
- This course will concentrate on Instruction Set Architectures and their use through *assembly language programming*
  - Assembly language is a human-readable version of machine code
  - Think of this as being like learning arithmetic – calculators are better for big calculations – but you can’t understand maths without doing it for yourself as well.
- We will use the ARM processor as our main example

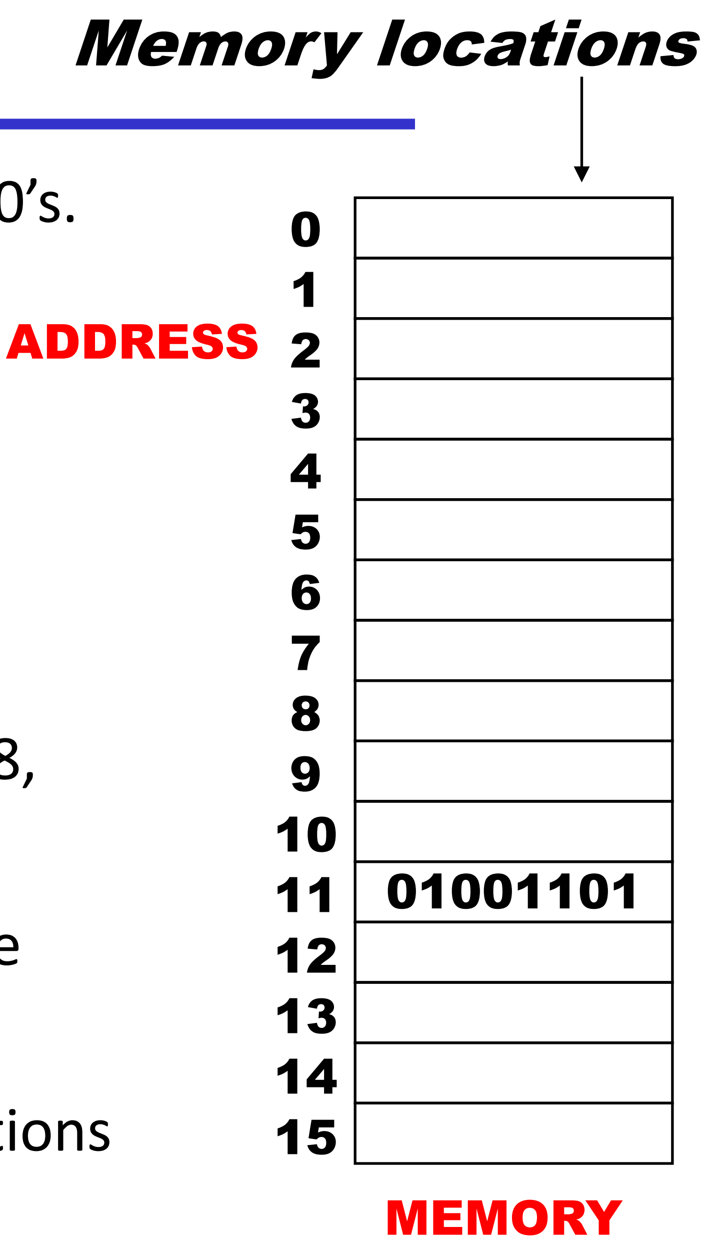
Lecture 2 A Very Simple Processor

The point of philosophy is to start with something so simple as not to seem worth stating, and to end with something so paradoxical that no one will believe it."
Bertrand Russell

- In this lecture we will show how the **control** part of a CPU works.
- We describe the operation, in detail, of a very simple processor called MU0.
- Although this is a toy example it illustrates important concepts we use later on:
  - Machine instructions written in **assembler**, and codes in machine words
  - Sequential execution of instructions

Data in Computer Memory

- Look into memory and you'll see '1's and '0's. Meaning depends on context: could be program, could be numeric data, etc
- Data & programs are stored in memory
- Each memory location has fixed width: number of bits (binary digits) of data
- The **width** of the memory unit is typically 8, 16, 32, 64 bits etc.
- The width of the **address** a depends on the number of locations N:
  - Thus 8 bits can address up to 256 locations



Digital representation of numbers

- We will say more about this in part 2
- For now, a number stored in a computer will have a fixed number of bits (binary digits) usually 8, 16 , or 32.
  - N bits can store whole unsigned numbers in range $0 \leq x \leq 2^N - 1$
- Each storage location or register in a computer which stores data has a fixed size in bits, and thus can store non-negative whole numbers up to a given maximum size.

2^4 = 16 numbers

0000	**0**
0001	**1**
0010	**2**
0011	**3**
0100	**4**
0101	**5**
0110	**6**
0111	**7**
1000	**8**
1001	**9**
1010	**10**
1011	**11**
1100	**12**
1101	**13**
1110	**14**
1111	**15**

How to write numbers

16 bit memory location or register

0	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**

What is stored in hardware:	**0000001100011010**
Human readable representations:	
binary number (base 2) 0 or 1	1100011010
decimal number (base 10) 0-9	794
hexadecimal number (base 16) 0-9, A-F	31A
as above, but with leading zeros	031A

Use whichever representation is easiest - the data is the same

More on Binary Notation – numbers as 1s and 0s

- Digital computers use a binary number system where the *base* (or *radix*) is 2. The radix determines the weight given to each digit:

$$DIGIT * 2^{POSITION\,\#}$$

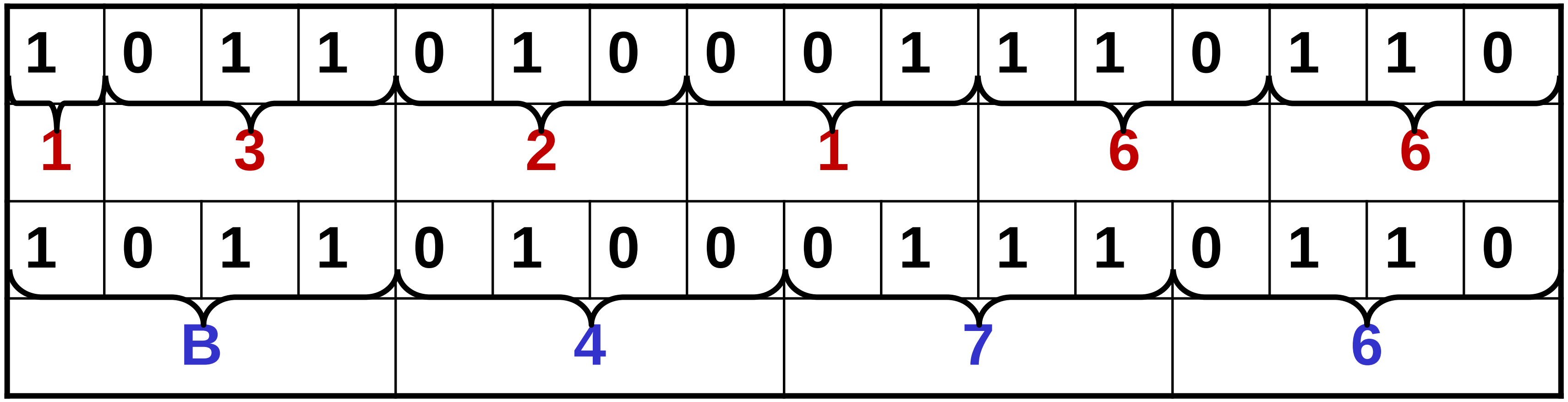
Fours	Twos	Ones		Halves	Fourths
2^2	2^1	2^0		2^{-1}	2^{-2}
1	1	0	.	1	1

- For example, the value of this binary number is as on left:

$1*2^2$	=	1*4	=	4.
$1*2^1$	=	1*2	=	2.
$0*2^0$	=	0*1	=	0.
$1*2^{-1}$	=	1*.5	=	0.5
$1*2^{-2}$	=	1*.25	=	+ 0.25
				6.75

- NOTATION:
  - $12_{(10)}$ 12 decimal
  - $12_{(8)}$ 12 octal = $10_{(10)}$
  - $1000_{(2)}$ 1000 binary = $8_{(16)}$

Octal and hexadecimal number systems: easy to translate to/from binary



Extra hex digits

1	0	1	0	**10**	**A**
1	0	1	1	**11**	**B**
1	1	0	0	**12**	**C**
1	1	0	1	**13**	**D**
1	1	1	0	**14**	**E**
1	1	1	1	**15**	**F**

$132166_{(8)} = B476_{(16)}$

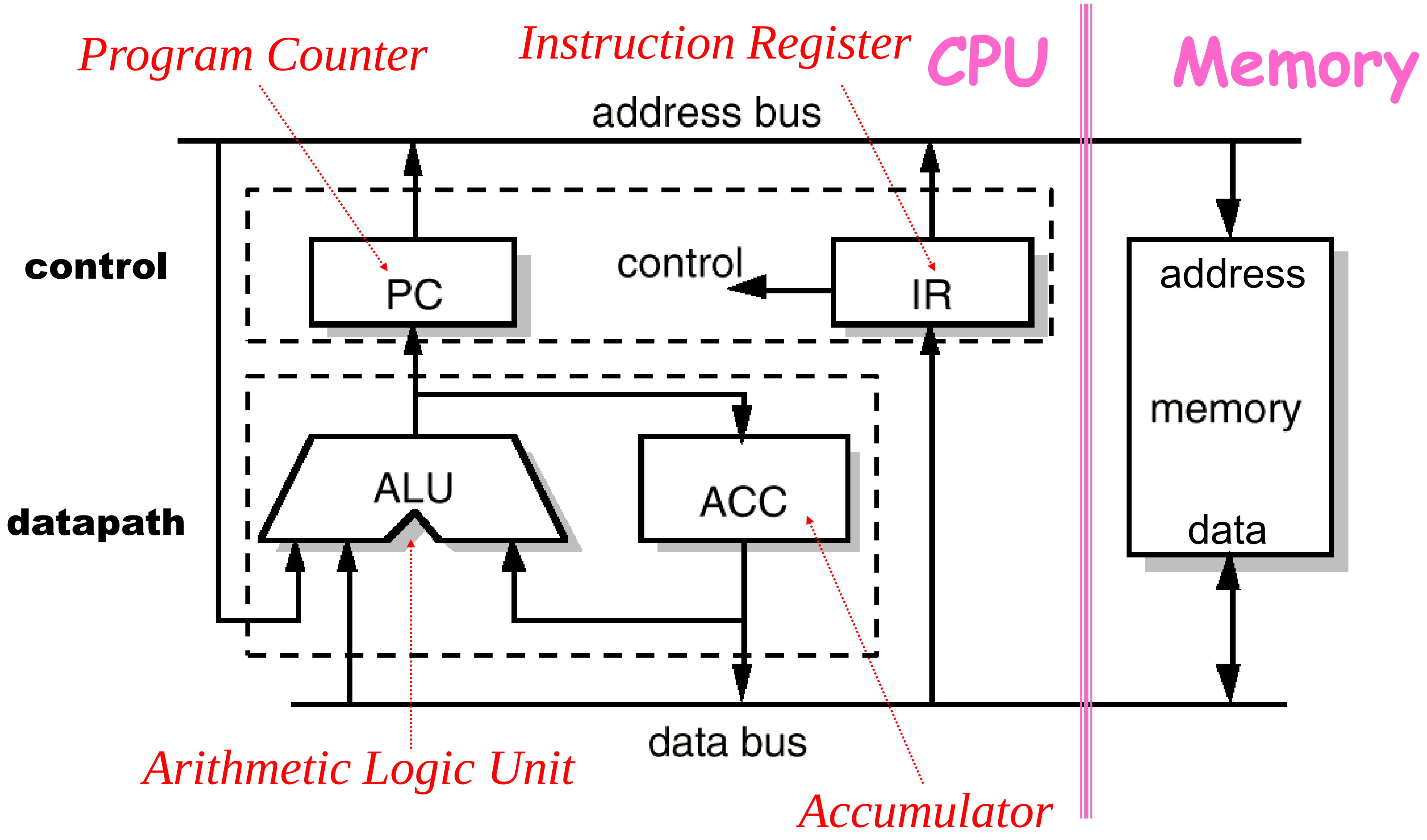
What is this number in decimal?

(((**11***16+**4**)*16+**7**)*16)+**6**

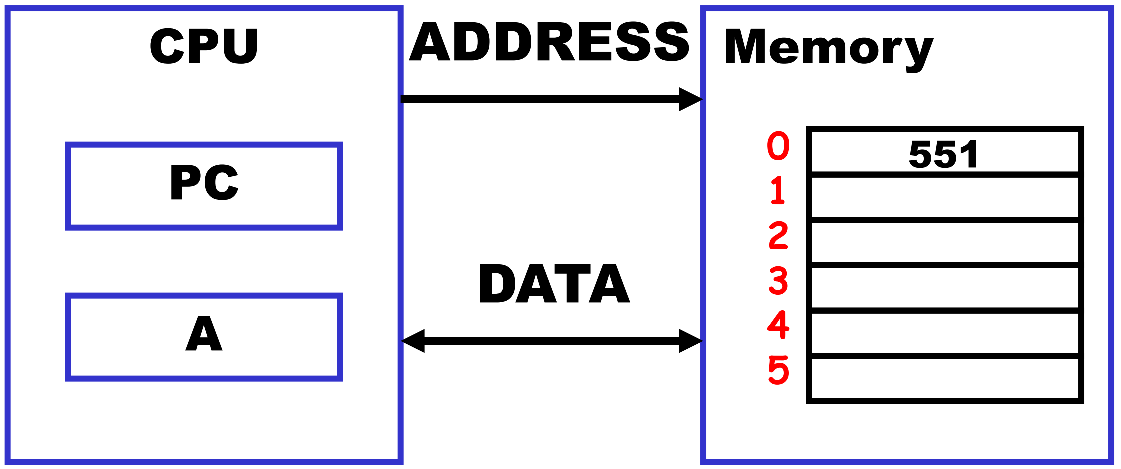
Hexadecimal examples

- You should remember these:
  - $10_{(16)} = 16$, $F_{(16)} = 15$
  - $100_{(16)} = 256$, $FF_{(16)} = 255$
  - $1000_{(16)} = 4096$, $FFF_{(16)} = 4095$
  - $10000_{(16)} = 65536$, $FFFF_{(16)} = 65535$
- For convenience 1024 = 1K, so 4096=4K, 65536=64K etc.
- What is $2013_{(16)}$?
  - In binary?
  - In decimal?
- Can you work out $(1000_{(16)} – 8)$ in hexadecimal (borrow is 16, not 10)?
  - $FF8_{(16)}$

MU0 - A Very Simple Processor



Logical (programmer's) view of MU0



Memory location with *address* 0 is storing data 551

Registers:
Each can store one 16 bit number
A: Accumulator
PC: Program Counter

(NB IR is not visible to programmer)

Memory Locations:
Each can store one 16 bit number

MU0 Design

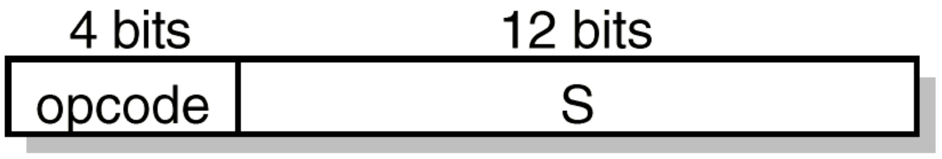
- Let us design a simple processor MU0 with 16-bit instruction and data bus and minimal hardware:-
  - Program Counter (PC) - holds address of the next instruction to execute (a register)
  - One register: accumulator (A) - holds data being processed
  - Instruction Register (IR) - holds current instruction code being executed
  - Arithmetic Logic Unit (ALU) - performs operations on data
- We will only design 8 instructions, but to leave room for expansion, we will allow capacity for 16 instructions
  - so we need 4 bits to identify an instruction: the *opcode*

MU0 Design (2)

- Let us further assume that the memory is word-addressable (forget bytes)
  - each 16-bit word has its own location: word 0, word 1, etc.
    - Mu0 can't address individual bytes!

address	data
0	$0123_{(16)}$
1	$7777_{(16)}$

- The 16-bit instruction code (machine code) has a format:



- Note top 4 bits define the operation code (opcode) and the bottom 12 bits define the memory address of the data (the operand)
- This machine can address up to 2^{12} = 4k words = 8k bytes of data

MU0 Instruction Set

- **mem[S]** – contents of memory location with *address* **S**
- [Think of memory locations as being an array – here S is the array index]
- A is the single 16 bit CPU register
- S is a number from instruction in range 0-4095 (000$_{(16)}$-FFF$_{(16)}$)

	Instruction	**Opcode (hex)**	**Effect**
LoaD A	**LDA S**	**0000** (0)	**A := mem[S]**
Store A	**STA S**	**0001** (1)	**mem[S] := A**
ADD to A	**ADD S**	**0010** (2)	**A := A + mem[S]**
SUBtract from A	**SUB S**	**0011** (3)	**A := A – mem[S]**
JuMP	**JMP S**	**0100** (4)	**PC := S**
Jump if Gt Equal	**JGE S**	**0101** (5)	**if A ≥ 0, PC := S**
Jump if Not Equal	**JNE S**	**0110** (6)	**if A ≠ 0, PC := S**
SToP	**STP**	**0111** (7)	**stop**

Our First Program

- The simplest use of our microprocessor: add two numbers
  - Let’s assume these numbers are stored at **two consecutive locations in memory**, with addresses 2E and 2F
  - Let’s assume we wish to store the result back to memory location [with address] 30
- We need to load the accumulator with one value, add the other, and then store the result back into memory

Instructions execute in sequence

```
LDA  02E              002E
ADD  02F      ->      202F
STA  030              1030
STP                   7???
```

Note – we follow tradition and use Hex notation for addresses and data

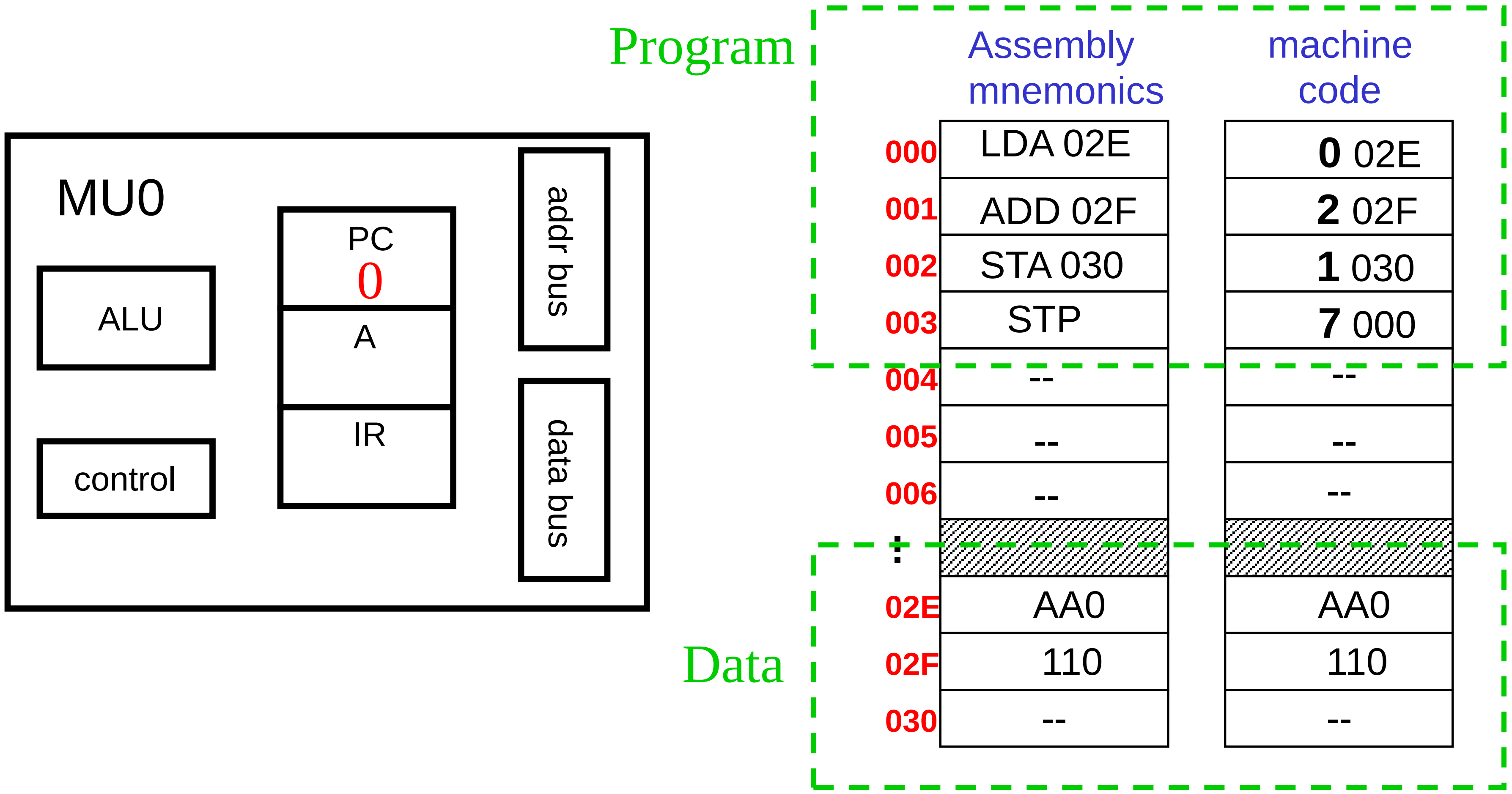
Human readable (mnemonic) assembly code

Machine Code

Walkthrough of MU0 execution

- Common technique used to debug computer programs
- Write down state of all internal registers, memory, etc
- Trace through instructions making the necessary changes at each step
- In machine code the PC register determines which instruction will next be executed.
  - Normally PC is incremented at end of each instruction to get to the next instruction in the sequence
- Each machine instruction is implemented in hardware during one or more clock cycles in which things happen. We will trace through each cycle
  - This technique is invaluable when debugging or learning initially!

Caught in the Act!

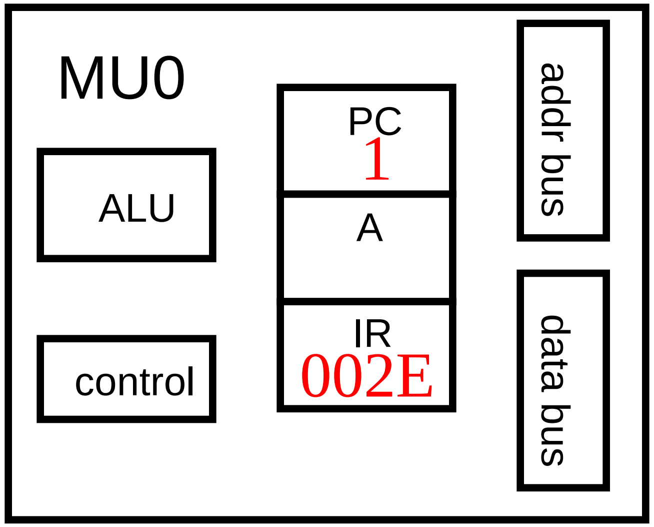


- Initially, we assume PC = 0, data and instructions are loaded in memory as shown, other CPU registers are undefined.

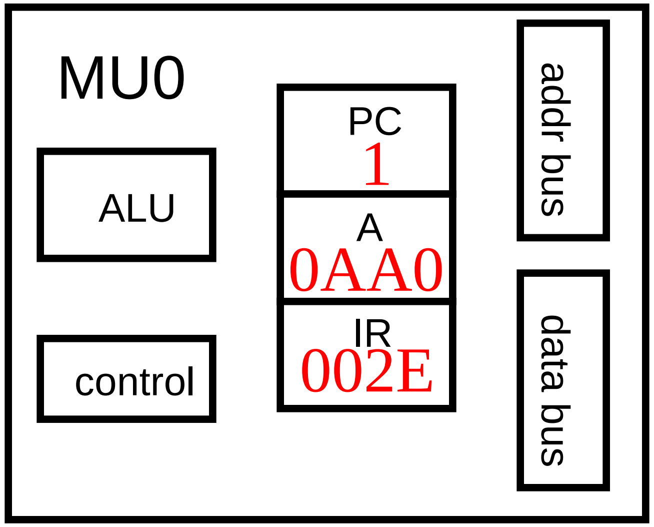
Instruction 1: LDA 02E

NB – data shown is after each cycle has completed – so PC is one more than PC used to fetch instruction

Cycle 1
(fetch instr and increment PC)



Cycle 2
(execute instruction)

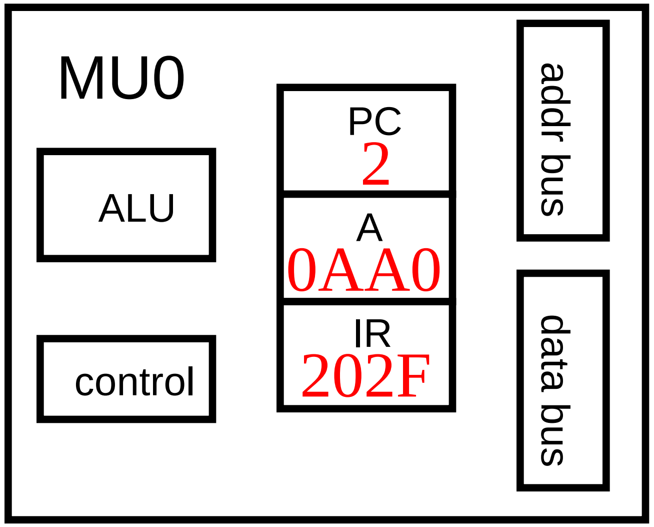


machine code

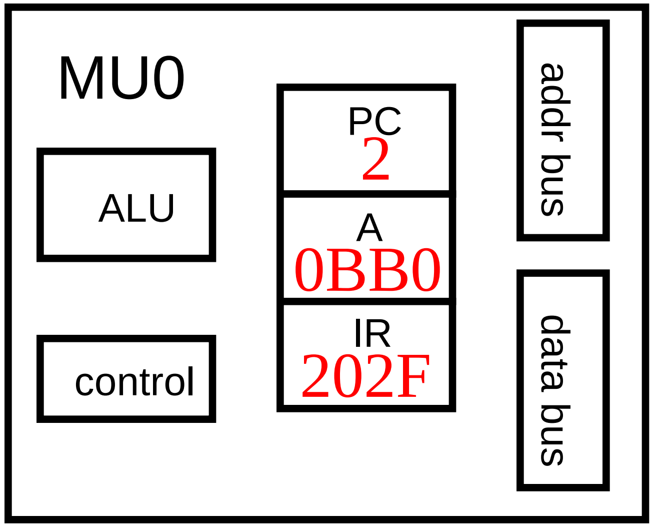
000	**0** 02E
001	**2** 02F
002	**1** 030
003	**7** 000
004	--
005	--
006	--
⋮	▨
02E	0AA0
02F	0110
030	--

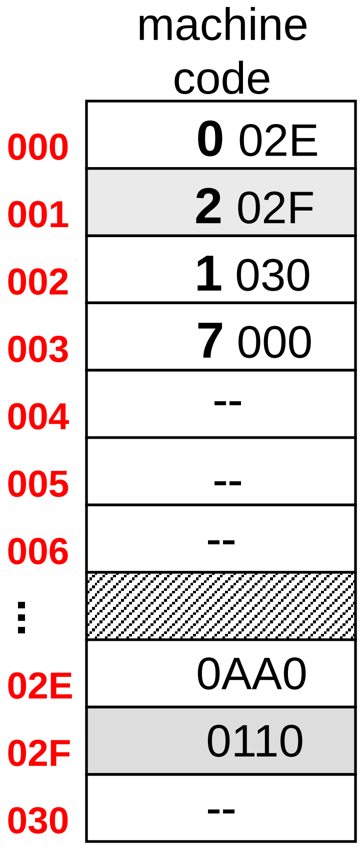
Instruction 2: ADD 02F

Cycle 1



Cycle 2

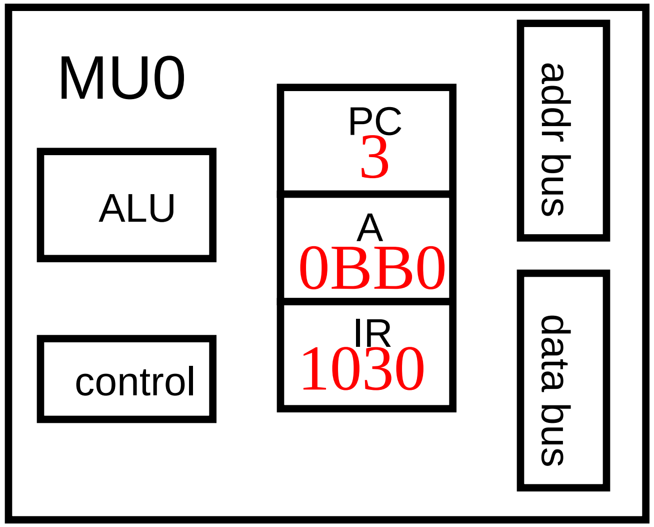




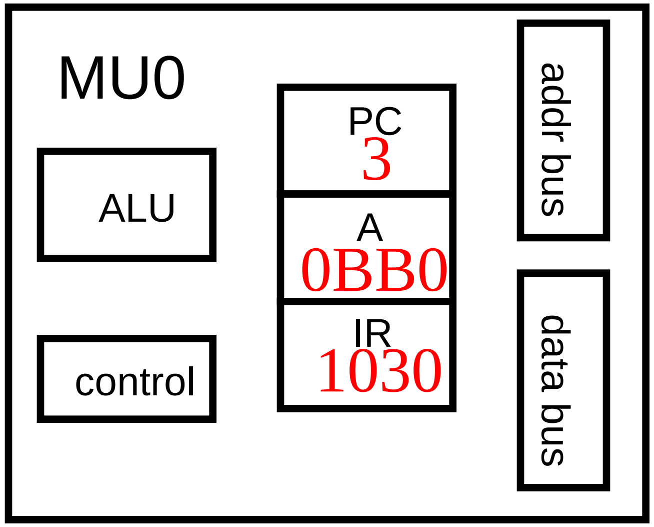
	machine code
000	**0** 02E
001	**2** 02F
002	**1** 030
003	**7** 000
004	--
005	--
006	--
⋮	▨
02E	0AA0
02F	0110
030	--

Instruction 3: STA 030

Cycle 1



Cycle 2

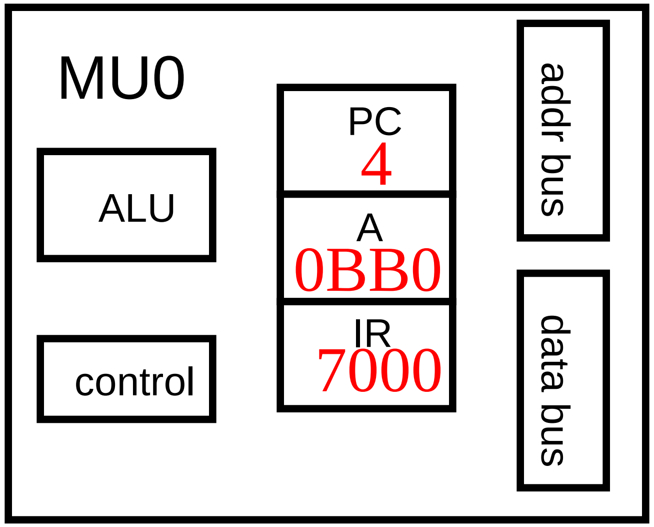


machine code

000	**0** 02E
001	**2** 02F
002	**1** 030
003	**7** 000
004	--
005	--
006	--
⋮	▨
02E	0AA0
02F	0110
030	**0BB0**

Instruction 4: STP

Cycle 1



machine code

000	**0** 02E
001	**2** 02F
002	**1** 030
003	**7** 000
004	--
005	--
006	--
⋮	▨
02E	0AA0
02F	0110
030	0BB0

Key Points: instructions

- Microprocessors perform operations depending on instruction codes stored in memory
- Instructions usually have two parts:
  - Opcode - determines what is to be done with the data
  - Operand - specifies where/what is the data
- Program Counter (PC) - address of current instruction
- PC incremented automatically each time it is used
  - Therefore instructions are normally executed sequentially
- The number of clock cycles taken by a MU0 instruction is the same as the number of memory accesses it makes.
  - LDA, STA, ADD, SUB therefore take 2 clock cycles each: one to READ from memory (fetch) the instruction, a second to fetch (and operate on) the data
  - JMP, JGE, JNE, STP only need one memory read (the instruction itself) and therefore can be executed in one clock cycle.

Key Points: hardware

- Memory contains both programs and data
- Program area and data area in memory are usually well separated. Reading program instructions as data is possible but not useful
- ALU is responsible for arithmetic and logic functions
- There are usually one or more general purpose registers for storing results or memory addresses. MU0 only has one: A
  - more registers => more powerful. See ARM later
- Fetching data from inside the CPU is much faster than from memory unit
  - Assume number of memory operations determines number of cycles needed to execute instruction
- Assume MU0 will always reset to start execution from address 000_{16}.

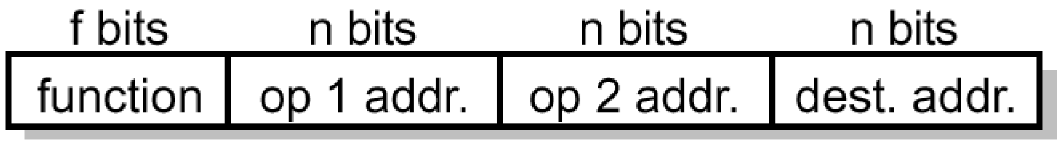
Lecture 3 – Memory Fundamentals

"A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible. There are no prima donnas in engineering." Freeman Dyson

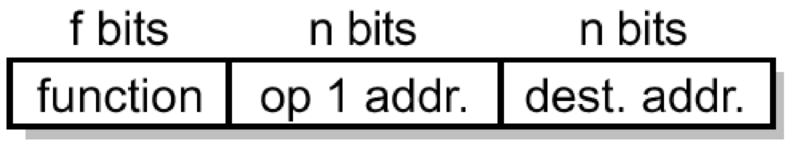
- This will introduce some of the general features needed to understand other ISAs such as the much more complex ARM ISA which we study in detail in Part 2
  - CPU instruction format
  - Memory architecture

Instruction format classification

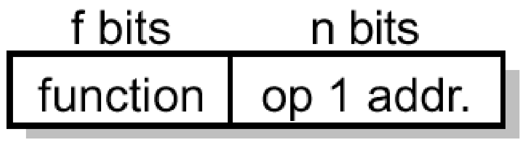
- 3-operand instruction format (used by ARM processor)
  - dest := op1 op op2



- 2-operand instruction format (used by the AVR 8 bit microcontrollers)
  - dest := dest op op1



- 1-operand instruction format (used in MU0 and some 8-bit microcontrollers such as MC6811)
  - acc := acc op op1



a := b+c Memory or Registers?

2 & 3 operand ISAa have multiple registers
Assume 8 registers called R0-R7

a,b,c stored in memory

```
LDA mem[100]
ADD mem[101]
STA mem[102]
```

1 operand (MU0)

a:	mem[102]
b:	mem[101]
c:	mem[100]

a,b,c stored in registers

```
ADD R0,R1
MOV, R2, R0
```

2 operand (AVR)

a:	R2
b:	R1
c:	R0

```
ADD R0,R1     ;R0:=R0+R1
MOV R2,R0     ;R2 := R0
```

```
ADD R2, R1, R0
```

3 operand (ARM)

a:	R2
b:	R1
c:	R0

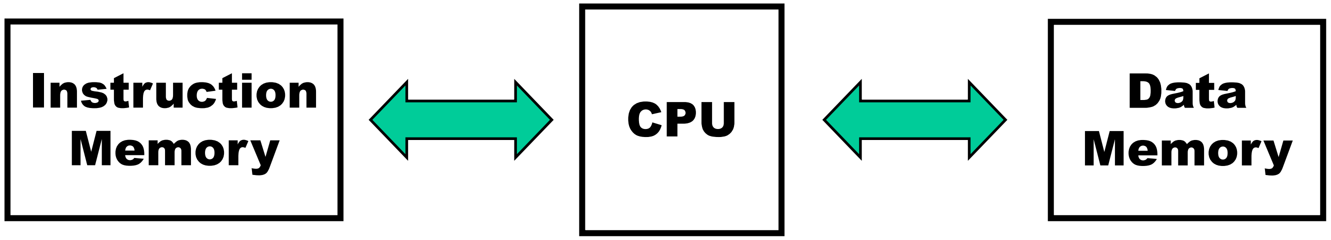
```
ADD R2,R1,R0     ;R2:=R1+R0
```

Modern CPU Design

- **1. Load / Store architecture**
  - registers are faster than memory so use them whenever possible
  - Lots of registers – only go to main memory when really necessary.
  - Separate register/ALU instructions (fast) from memory READ & WRITE.
    - Memory READ is called LOAD, Memory WRITE is called STORE
- **2. Concurrent execution of instructions for greater speed**
  - multiple function units (ALUs, etc) – superscalar or VLIW (EPIC) – examples: Pentium & Athlon
  - "production line" arrangement overlaps instructions in a pipeline: all modern CPU.
    - See later
- **3. On-chip caches for greater speed**
  - Incorpoate some memory (cache) on same chip as CPU
    - See later

Main memory organisation

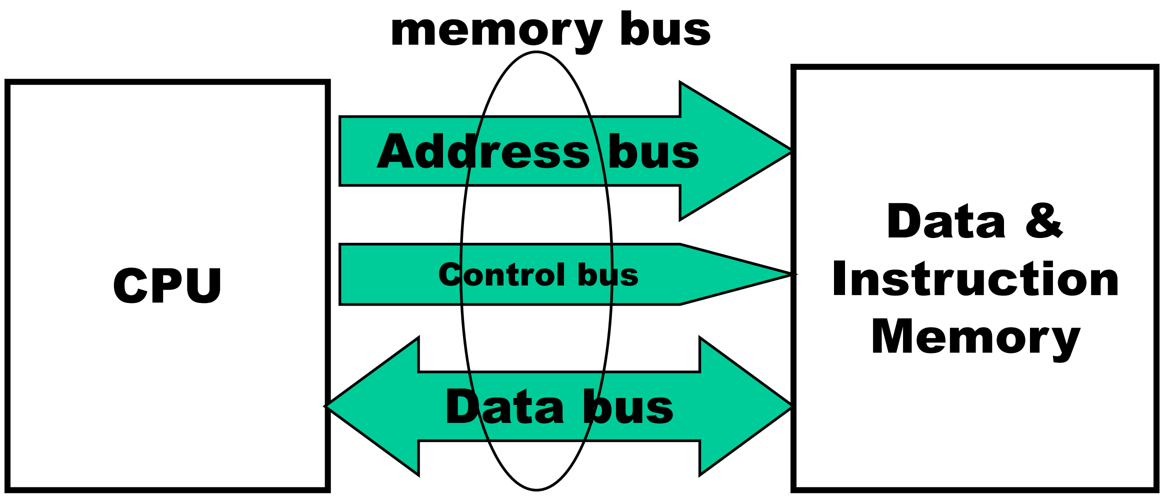
- Main memory is used to store programs, data, intermediate results
- Two main organisations: **Harvard** & **von Neumann**
- **Harvard architecture.**
  - In A Harvard architecture CPU programs are stored in a separate memory (possibly with a different width) from the data memory. This has the added benefit that instructions can be fetched at the same time as data, simplifying & speeding up the hardware.
  - In practice, the convenience of being able to read and write programs just like normal data makes this less usual
    - still popular for fixed program microcontrollers.



Harvard architecture

Von Neumann memory architecture

- **Von Neumann architecture (like MU0).**
  - Programs and data occupy a single memory.
- Think of main memory as being an array of **words**, the array index being the memory **address**. Each word (array location) has **data** which can be separately written or read.
- Usually instructions are one word in length – but can be either more or less



Memory review

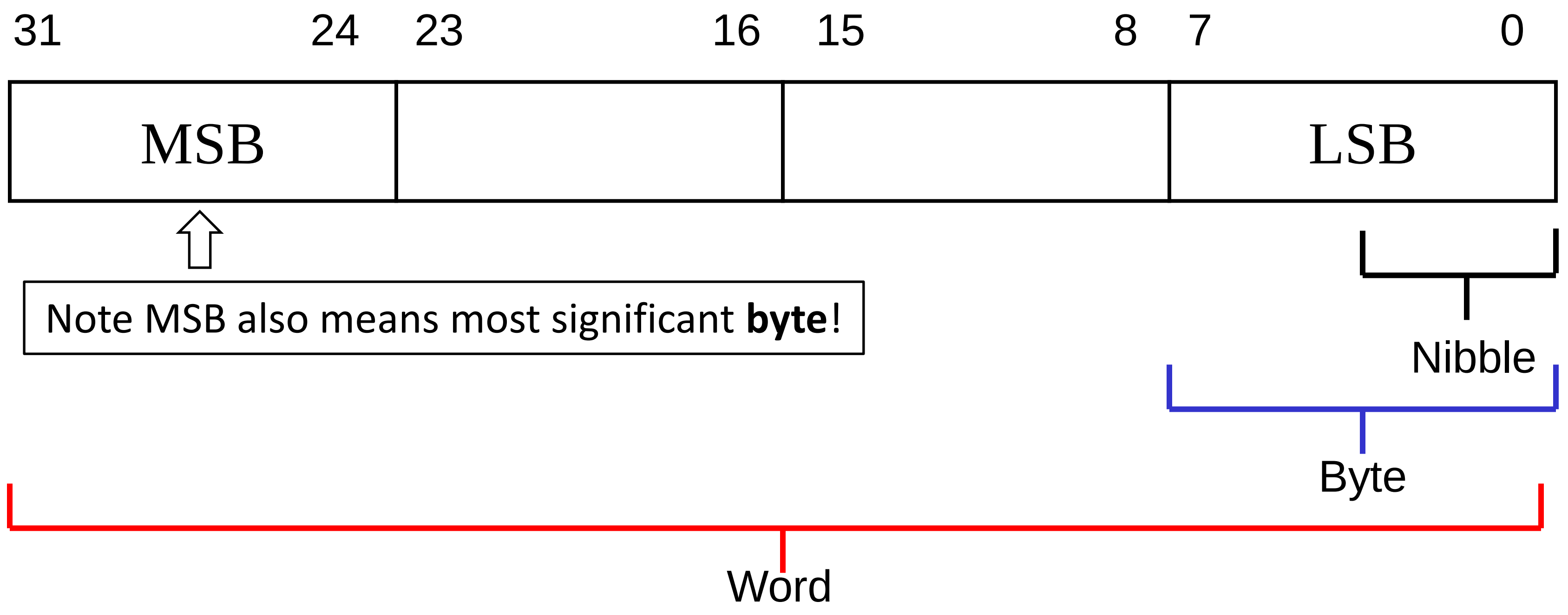
- **Memory locations** store instructions or data and each have unique numeric addresses
  - Usually addresses range from 0 up to some maximum value.
- **Memory space** is the unique range of possible memory addresses in a computer system
- We talk about "the address of a memory location".
- Each memory location stores a fixed number of bits of data, normally 8, 16, 32 or 64
- We write mem_8[100], mem_{16}[100] to indicate the value of the 8 or 16 bits with memory address 100 etc. & indicates hex number.
  - **mem_{16}[&02E] = &0AA0**

	machine code
000	**0** 02E
001	**2** 02F
002	**1** 030
003	**7** 000
004	--
005	--
006	--
⋮	▨
02E	0AA0
02F	0110
030	0BB0

Nibbles, Bytes, Words

- Internal datapaths inside computers could be different width - for example 4-bit, 8-bit, 16-bit or 32-bit.
- For example: ARM processor uses 32-bit internal datapath
- WORD = 32-bit for ARM, 16-bit for MU0, 64 bit for latest x86 processors
- BYTE (8 bits) and NIBBLE (4 bits) are architecture independent

Note bit numbers: 31 - **most significant bit** - MSB, 0 **- least significant bit** - LSB

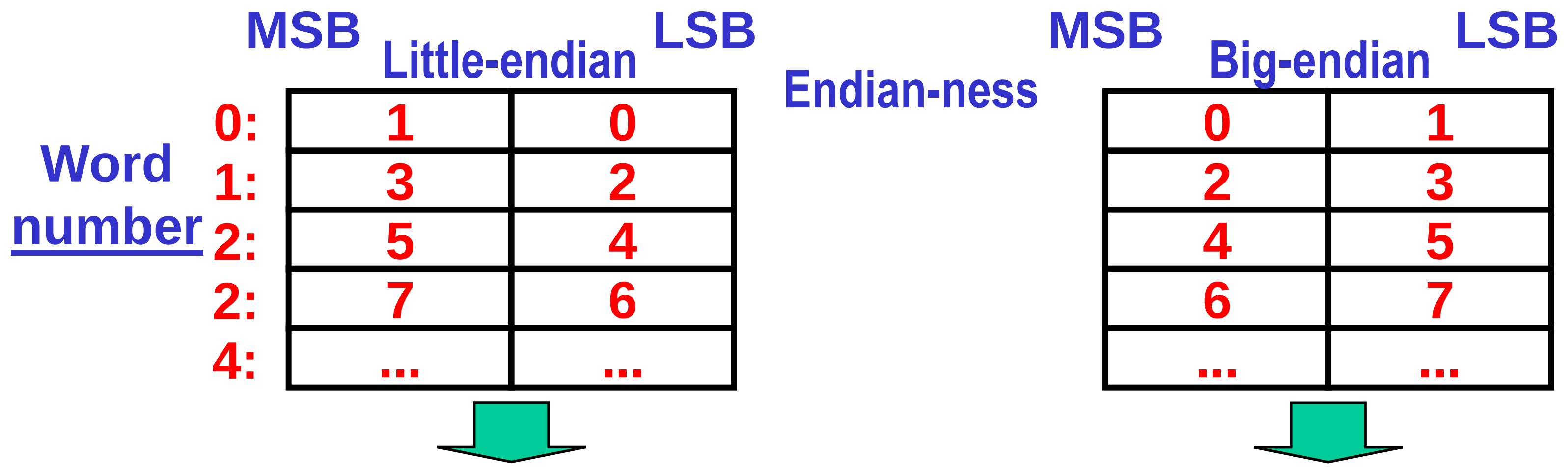


Note MSB also means most significant **byte**!

Byte and Word addressing

- Many programs, for example those manipulating characters – each of which takes up one byte – need to access memory as an array of bytes, not words.
- For example in a 16 bit system there are 2 bytes for every word. This means that byte addresses must increase twice as fast as word addresses. There are two different ways to number bytes within a word as below.

NB. Numbers in locations are byte address, not data



Little-endian

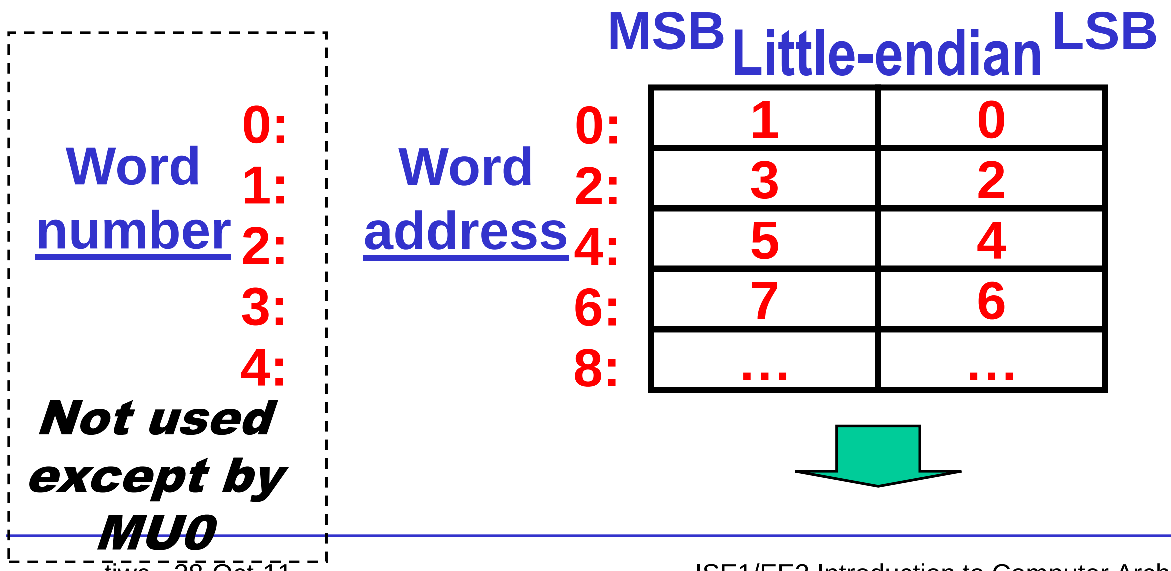
Word number	MSB	LSB
0:	1	0
1:	3	2
2:	5	4
2:	7	6
4:	...	...

Big-endian

MSB	LSB
0	1
2	3
4	5
6	7
...	...

Byte addresses for words

- Most computer systems now use **little-endian** byte addressing, in which the least-significant byte has the lower address.
- It is inconvenient to have completely separate byte and word addresses, so **word addressing** usually follows **byte addressing**.
  - The word address of a word is the byte address of its lowest numbered byte. This means that consecutive words have addresses separated by 2 (16 bit words) or 4 (32 bit words) etc.



Word address	MSB	LSB
0:	1	0
2:	3	2
4:	5	4
6:	7	6
8:	…	…

16 bit memory with consecutive word addresses separated by 2

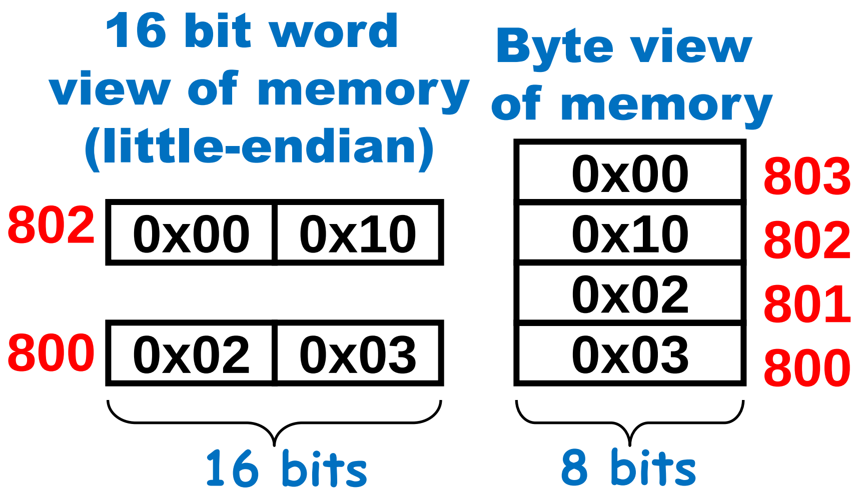
What is an "address"?

- **Conceptually, a memory location address is just a name, like A,B,C,D. The location value (stored data) changes, not the address.**
- **Practically, because addresses are numbers, we can use this fact to advantage**
- **MU0 executes sequential programs by adding one to PC each instruction. PC is the address of the next instruction to be executed**
- **Later we will see that numeric addresses allow sequential data access**
- **Words & bytes both need addresses so they can be READ & WRITTEN**
- **The best solution is to make word addresses follow bytes**
  - **16 bit memory: 0,2,4,6,...**
  - **32 bit memory: 0,4,8,12,...**

Different views of memory access: byte or word

NB 0x alternate notation for hex (&)

- Different instructions to allow either **byte** or **word** access to memory.
  - Word **read** or **write** operates on a whole word **Mem**$_{16}$ in memory
  - The memory address is the word address (even number)
  - Byte read or write is similar except that a single byte (half a word here) is read or written **Mem**$_8$.



Mem$_{16}$[800]= 0x0203$_{(16)}$ = 515

Mem$_{16}$[802]= 10$_{(16)}$ =16

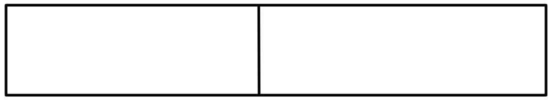
Mem$_8$[800]= 3

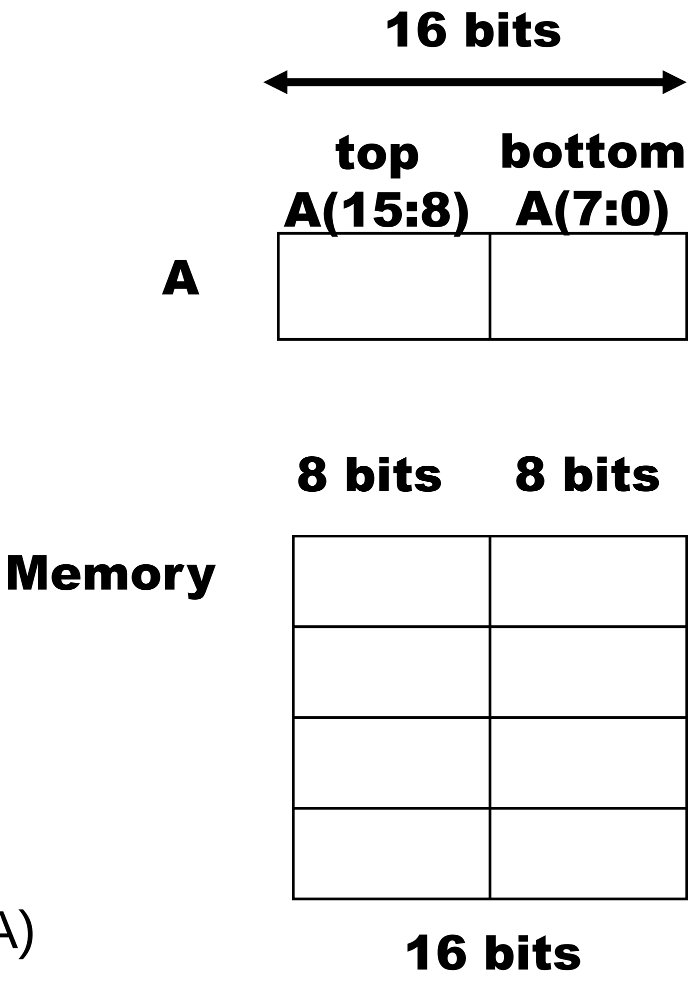
Mem$_8$[801]= 2

Mem$_8$[802]= 10$_{(16)}$= 16

Mem$_8$[803]= 0

CPU Registers & Memory

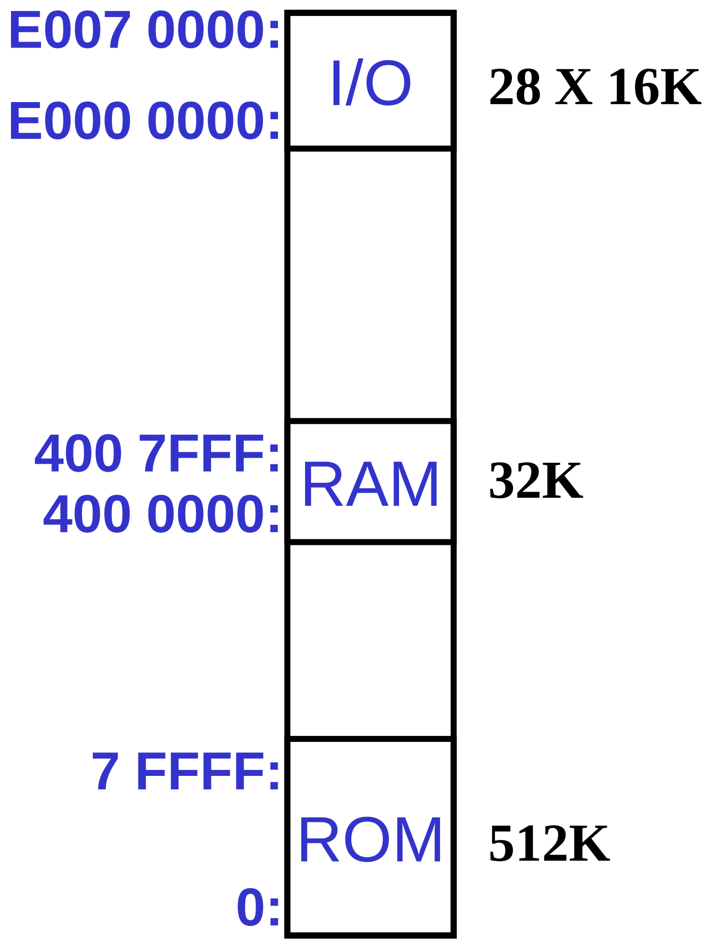
- CPU registers (e.g. A, R0) are normally same length as memory **word**
- Word READ (easy):
  - A := Mem_{16}[addr]
- Word WRITE (easy):
  - Mem_{16}[addr] := A
- Byte READ: 8 →16?
  - A: 
  - A := 00000000 Mem_8[addr]
- Byte WRITE: 16 → 8?
  - Mem_8[addr] := A(7:0) (bottom 8 bits of A)



What are memory locations used for?

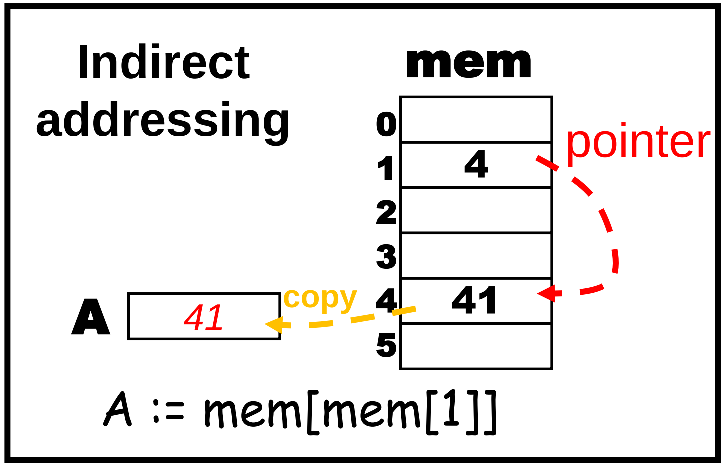
- *Read-write memory* (RAM) is used for data and programs. It loses its contents on power-down.
- *Read-only memory* (ROM) typically used to hold programs that do not change
  - *Flash ROM* allows data to be changed by programming (but not by memory write).
- *Memory-mapped I/O*. Some locations (addresses) in memory allow communication with peripheral devices.
  - For example, a memory write to the data register of a serial communication controller might output a byte on a serial port of a PC.
  - In practice, all I/O in modern systems is memory-mapped

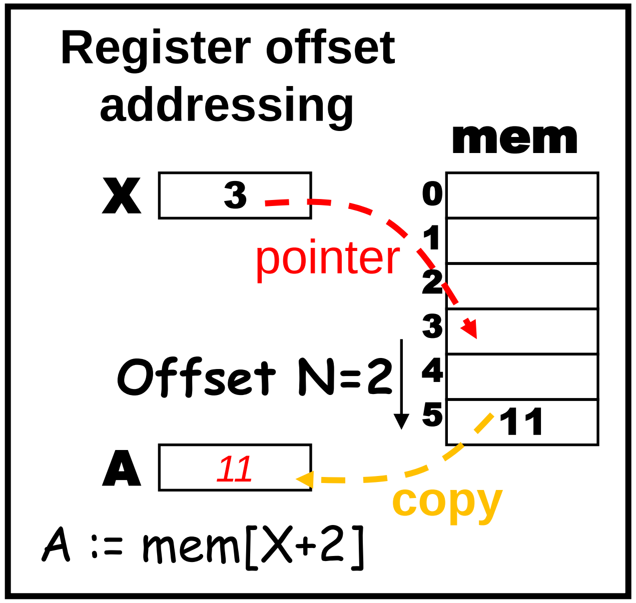
LPC2138 microcontroller On-chip memory map



Memory addressing modes

- **Direct addressing**
  - A := mem[N] (N in instruction word)
  - Limited memory space (by number of bits available)
  - Does not allow array access (where location is selcted by number in register)
- **Indirect addressing** (not often used now)
  - A := mem[mem[N]]
- **Register addressing**:
  - A := mem[X]
    - where X is the name of a **register** in CPU
- **Register offset addressing**
  - A := mem[X + N]
  - Register indirect is special case where offset = 0



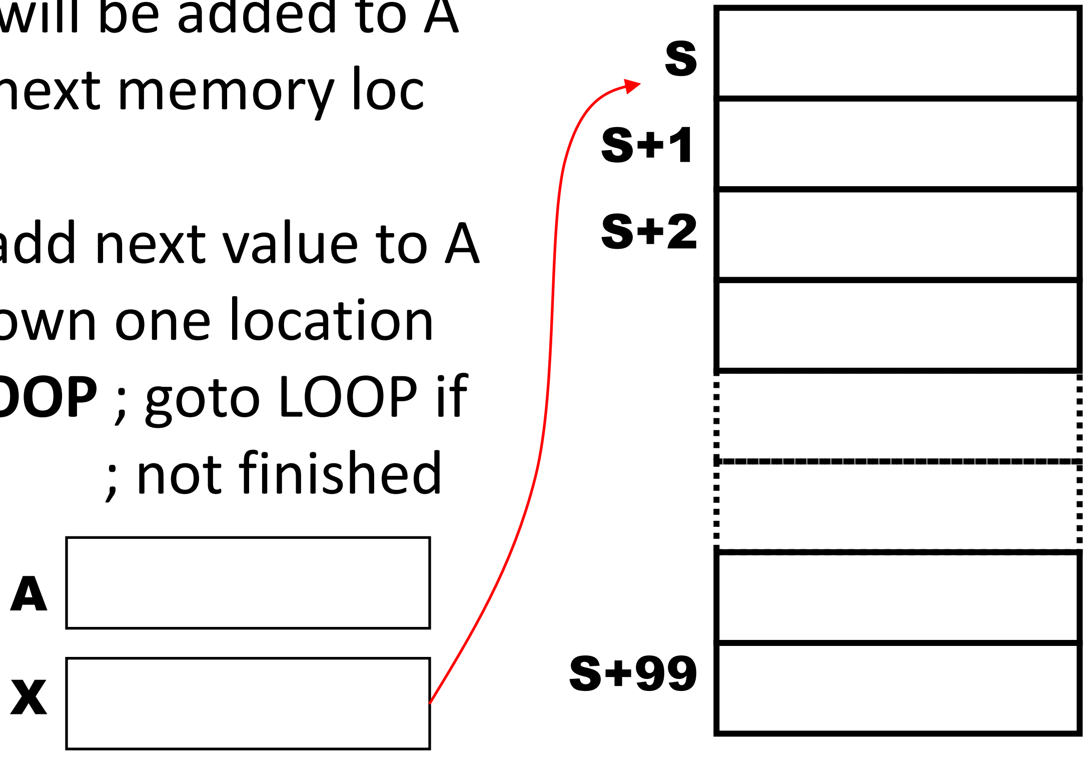


Example of register offset addressing

- Using register indirect addressing set A equal to sum of values in memory locations from S to S+99, then stop.
- Assume X is a register. Assume register addressing added to ISA

```
        A := 0 ; each value will be added to A
        X := S ; X points to next memory loc
LOOP:
        A := A + mem[X] ; add next value to A
        X := X + 1 ; move down one location
        if X < S+100 JMP LOOP ; goto LOOP if
        STP                   ; not finished
```

This uses register addressing. How to change to register offset addressing?



Advantages of **register offset** addressing mode over **direct** addressing

- Memory address size limited by **register size** not by bits in instruction word
  - 32 bits => 2^{32} = 4294967296 memory addresses
- Array memory access is possible (memory location read or written depends on value of a variable etc)
- If register is set to constant value can use just like direct addressing
  - OK if CPU has large number of register so can set one register permanently
- Allows programs to run using data anywhere in memory
  - Set register to base of program data area
- Register used like this is often called *base register*

Lecture 4 - Introduction to ARM programming

"Steve is one of the brightest guys I've ever worked with – brilliant - but when we decided to do a microprocessor on our own, I made two great decisions - I gave them two things which National, Intel and Motorola had never given their design teams: the first was no money; the second was no people. The only way they could do it was to keep it really simple." - Hermann Hauser talking about Steve Furber and the ARM design

- Why learn ARM?
  - Currently dominant architecture for embedded systems
  - 32 bits => powerful & fast
  - Efficient: very low power/MIPS
  - Regular instruction set with many advanced features

Beyond MU0 - A first look at ARM

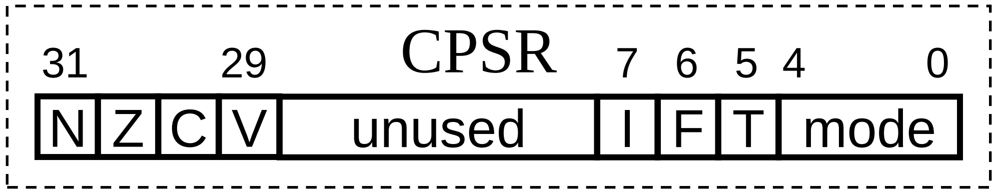
- **Complete instruction set.** Wide variety of arithmetic, logical, shift & conditional branch instructions
- **Larger address space** - 12-bit address gives 4k byte of memory. So use a 32-bit or address bus.
  - Typical physical memory size 1Mbyte (uses 20 bits) but can be anything up to 2^{32} bytes
- **Subroutine call mechanism** - this allows writing modular programs.
- **Additional internal registers** - this reduces the need for accessing external memory & speeds up calculations
- **Interrupts, direct memory access (DMA), and cache memory**.
  - *interrupts*: allow external devices (e.g. mouse, keyboard) to interrupt the current program execution
  - *DMA*: allows external high-throughput devices (e.g. display card) to access memory directly rather than through processor
  - *Cache*: a small amount of fast memory on the processor chip

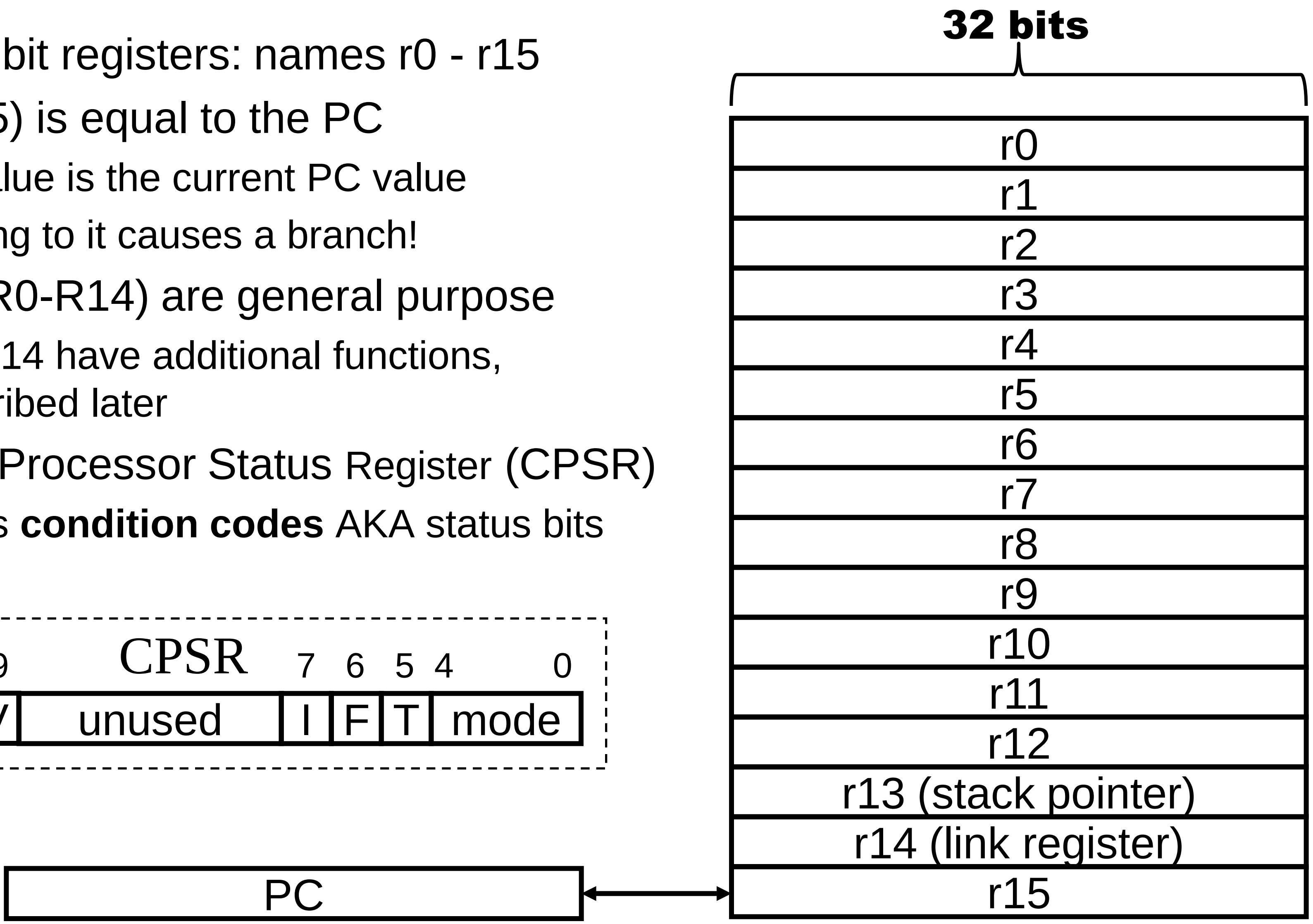
The ARM Instruction Set

- Load-Store architecture
- Fixed-length (32-bit) instructions
- 3-operand instruction format (2 source operand regs, 1 result operand reg): ALU operations very powerful (can include shifts)
- Conditional execution of ALL instructions (v. clever idea!)
- Load-Store multiple registers in one instruction
- A single-cycle *n*-bit shift with ALU operation
- “Combines the best of RISC with the best of CISC”

ARM Programmer's Model

- 16 X 32 bit registers: names r0 - r15
- r15 (R15) is equal to the PC
  - Its value is the current PC value
  - Writing to it causes a branch!
- r0-r14 (R0-R14) are general purpose
  - r13, r14 have additional functions, described later
- Current Processor Status Register (CPSR)
  - Holds **condition codes** AKA status bits

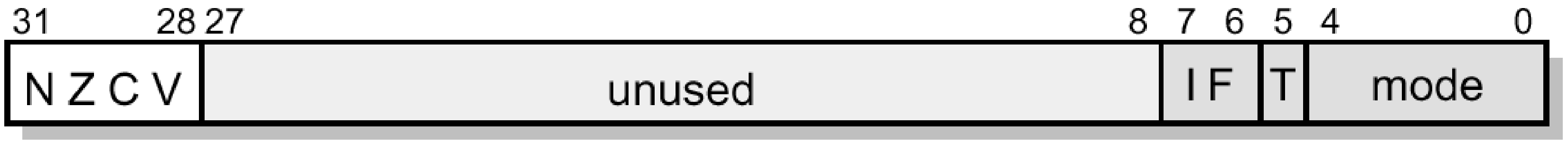




ARM Programmer's Model (con't)

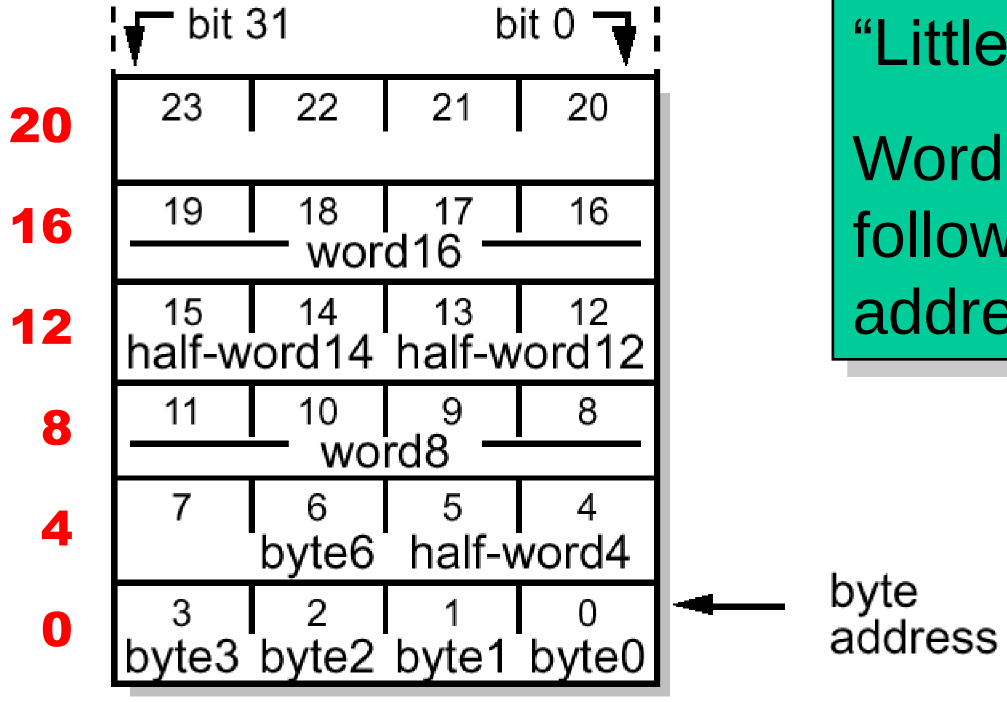
- CPSR is a special register, it cannot be read or written like other registers
  - The result of any data processing instruction can modify **status bits (flags)**
  - These flags are read to determine branch conditions etc
- Main status bits (AKA **condition codes**):
  - N (result was negative)
  - Z (result was zero)
  - C (result involved a carry-out (unsigned overflow)
  - V (result overflowed as signed number)
- Other fields described later

ARM CPSR format



ARM's memory organization

- Byte addressed memory
- Maximum 2^{32} bytes of memory
- A word = 32-bits, half-word = 16 bits
- Words aligned on 4-byte boundaries



NB - Lowest byte address = LSB of word

"Little-endian"

Word addresses follow LSB byte address

ARM Assembly Quick Introduction

MOV ra, rb
MOV ra, #n
	ra := rb
	ra := n
		◆n decimal in range -128 to 127

ADD ra, rb, rc
ADD ra, rb, #n
	ra := rb + rc
	ra := rb + n
		◆**SUB** => – instead of +

CMP ra, rb **CMP ra, #n**	set status bits on ra-rb set status bits on ra-n	◆**CMP** is like **SUB** but has no destination register and sets condition codes **NZCV**
B label	branch to label	**BL label** is branch & link to subroutine
BEQ label **BNE label** **BMI label** **BPL label**	branch to label if zero branch if not zero branch if negative branch if zero or plus	Branch conditions apply to the result of the last instruction to set **status bits** (ADDS/SUBS/MOVS/CMP etc).

LDR ra, label
STR ra, label
ADR ra, label
LDR ra, [rb]
STR ra, [rb]
	ra := mem[label]
	mem[label] := ra
	ra :=address of label
	ra := mem[rb]
	mem[rb] := ra
		◆LDRB/STRB => byte transfer
		◆Other address modes (**see later**):
		[rb,#n] => mem[rb+n]
		[rb,#n]! => mem[rb+n], rb := rb+n
		[rb],#n => mem[rb], rb:=rb+n
		[rb+ri] => mem[rb+ri]

Assembly Language Labels

- Programs need to reference memory addresses:
  - Memory LOAD & STORE (address of memory location)
  - Branch to a different instruction (address of destination instruction)
- In machine code addresses are just numbers
  - In assembler this is inconvenient, we want a symbolic equivalent!
- Labels are used to represent numbers in assembler

address	label	instruction	comment
00F8	START	MOV R10, #0	;R10 := 0
00FC		MOV R0, #4	;R0 := 4
0100	LOOP	ADD R10, R10, R11	;R10 := R10+R11
0104		SUB R0,R0,#1	;R0 := R0 -1
0108		CMP R0, #0	;Compare R0 with 0
010C		BEQ LOOP	;branch if R0 = 0
0110	FINISH		; finished

Symbol table	
START	&00F8
LOOP	&0100
FINISH	&0110

MU0 to ARM translation

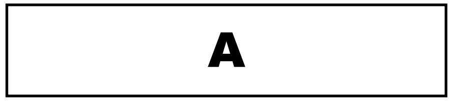
Operation	MU0	ARM

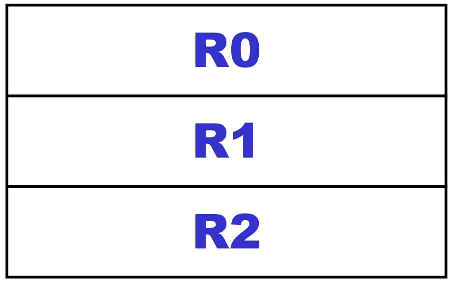
A := mem[S]
R0 := mem[S]
	LDA S	LDR R0, S

mem[S] := A
mem[S] := Rn
	STA S	STR R0, S

A := A + mem[S]
R0 := R0+ mem[S]
	ADD S	LDR R1, S
		ADD R0, R0, R1

R0 := S	n/a	MOV R0, #S
R0 := R1 + R2	n/a	ADD R0, R1, R2
PC := S	JMP S	B S





Introduction to ARM data processing a := b + c - d

ARM has 16 registers R0-R15

If a,b,c,d are in registers:

a:	R0
b:	R1
c:	R2
d:	R3

Machine Instructions:

```
ADD Rx,Ry,Rz     ;Rx := Ry + Rz
SUB Rx,Ry,Rz     ;Rx := Ry - Rz
```

```
ADD R0, R1, R2
SUB R0, R0, R3
```

LOAD data to reg from memory

STORE result to memory from reg

```
LDR R1, LB
LDR R2, LC
LDR R3, LD
ADD R0, R1, R2
SUB R0, R0, R3
STR R0, LA
```

LA,LB,LC,LD are labels for the addresses of 4 word memory locations

	memory
LA	a
LB	b
LC	c
LD	d

Negative numbers: two's complement

- Real programming need to add, signed numbers which can be positive or negative
- Memory locations & registers don't do this!
- SOLUTION - use two's complement signed representation
- Basic idea, use MSB to determine is number positive or negative
- The same bits now have two different meanings

Stored bits	Unsigned	2's comp signed
0011	3	3
0000	0	0
1011	11	-5

Two's Complement in 4 bit binary word

bits in register or memory location

b_3	b_2	b_1	b_0

unsigned binary

$$8b_3 + 4b_2 + 2b_1 + b_0 = u(b_i) \qquad 0 \le u \le 15$$

two's complement signed binary

$$-8b_3 + 4b_2 + 2b_1 + b_0 = z(b_i) \qquad -8 \le z \le 7$$

Computer data b_i has **unsigned** value $u(b_i)$ and **signed** value $z(b_i)$

NB definition of **z()** depends on number of bits since MSB has negative weight

b_i	$u(b_i)$	$z(b_i)$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Two's complement facts (1)

- Two's complement is very surprising!
- Suppose we have unsigned 4 bit addition:
- **u(ci) = u(bi) + u(ai)**
  - The result can be >15 so there is a **carry** or **overflow** bit c4
  - In a CPU the destination register can never hold this, because it is same width as source, so the result is always truncated ($c_3c_2c_1c_0$).
- **z(bi)+z(ai) = u(bi)+u(ai) - 16(a3+b3)**
- Truncated result ($c_3c_2c_1c_0$) of two's complement & unsigned addition is identical
  - Use one hardware adder - don't care whether numbers are signed or unsigned

	a3	**a2**	**a1**	**a0**	
	b3	**b2**	**b1**	**b0**	+
c4	**c3**	**c2**	**c1**	**c0**	

Two's complement signed numbers are universally used in computers because they can be added, subtracted with the same hardware as unsigned numbers

Two's complement facts (2)

- Assume registers are N bits wide
- Two's complement bit pattern for positive number ($b_{N-1} = 0$) is same as unsigned
- How do we find two's complement bit pattern b_i for negative number -S?
- $b_{N-1} = 1$ (because number is negative)

$$-S = z(bi) = u(bi) - 2^N b_N = u(b_i) - 2^N$$

$$\Rightarrow u(b_i) = 2^N - S = 1 + [(2^{N-1} - 1) - S]$$

- This makes an easy way to calculate **two's complement negation** of S.

1	**1**	**1**	**1**		**2^N - 1**
0	**1**	**0**	**1**	**–**	**S = 5**
1	**0**	**1**	**0**	**+1**	**(2^N-1) - S**
1	**0**	**1**	**1**		**2^N - S**

2's complement for -5

invert bits and add 1

Two's complement facts (3)

Overflow $\Rightarrow$ N bit result is not the correct sum of the N bit operands

addition	overflow condition	Note
unsigned	$u(a) + u(b) > 2^N - 1$	Overflow is same as Nth bit of sum (carry bit) = 1
Two's complement signed	$z(a) + z(b) < -2^{N-1}$ or $z(a) + z(b) > 2^{N-1} - 1$	The correct result must lie within two's complement number range

CMP instruction & condition codes

CMP R0, #n

- CMP compares two numbers by calculating the difference x = R0 - n
  - ***x is truncated to 32 bits***
  - Sets condition codes
- Use conditional branch after CMP

condition code	condition for 1	function
N	z(x) < 0	negative
Z	x = 0	zero
C	u(R0) ≥ n	unsigned overflow
V	z(x) ≠ z(R0) - z(n)	signed overflow

Branch	condition CMP R0, #n		after CMP a , b
BEQ	R0 = n	Z=1	a = b
BNE	R0 <> n	Z=0	a ≠ b
BMI	u(R0) < u(n)	C=0	u(a) < u(b)
BPL	u(R0) ≥ u(n)	C=1	u(a) ≥ u(b)

z(x) two complement interpretation

u(x) unsigned interpretation

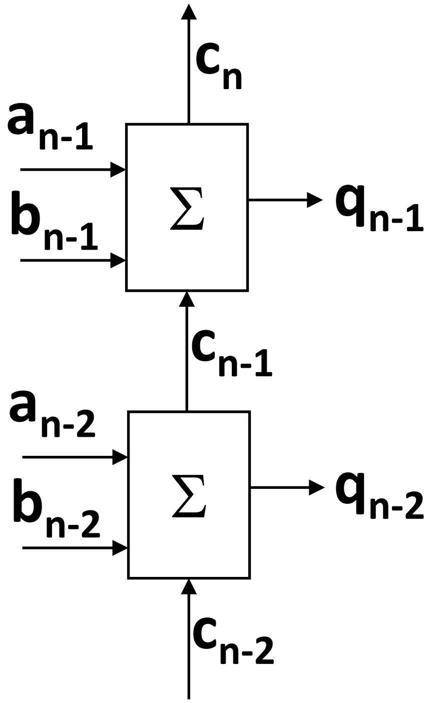
Carry bit (C) - hardware

- Suppose q_i is the N bit sum of a binary adder inputs a_i & b_i

Overflow $\Rightarrow$ N bit sum is wrong:

- Unsigned
  - $u(a_i) + u(b_i) = u(q_i) + 2^n c_n$
- Cn = 1 is condition for *unsigned* overflow
- In ARM architecture C_n is stored in C condition code

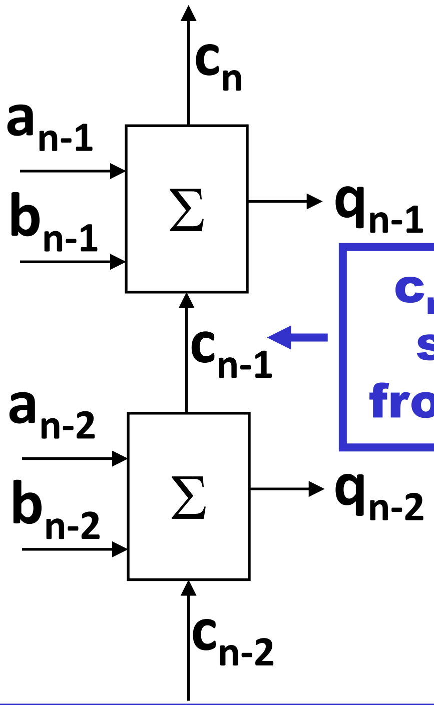
c_n is lost change from 2^n to 0



MS two bits of n bit addition

Signed overflow (V) & bit - hardware

- **Signed**
  - $z(a_i) + z(b_i) = z(qi) - 2^n c_n + 2^{n-1} * 2 * c_{n-1})$
  - Therefore signed overflow if $2^n(c_{n-1} - c_n) \neq 0$
  - $V = c_n \text{ xor } c_{n-1}$



c_n is lost change from -2^n to 0

c_{n-1} weighting sign change from 2^{n-1} to -2^{n-1}

MS two bits of n bit addition

Overflow (how you calculate)

- Compute: z = z(a) + z(b)
  - This is what the answer **should** be
- If x lies within range of two's complement there is no signed overflow
  - **$-2^{n-1} \leq x \leq 2^{n-1}-1 \Rightarrow$ no overflow**
- Note that the same is true for unsigned overflow
- Compute u = u(a) + u(b)
  - **$0 \leq u \leq 2^{n} - 1 \Rightarrow$ no overflow**
- Addition does not care about signedness. Both C & V are computed. Programmer decides which is important.

What is subtraction in binary?

- In a microprocessor
  - Subtract generates correct **two's complement** answer for **two's complement** operands.
  - Subtract = **negate** followed by **add**: a - b = a + (-b)
  - Example: 4 - 1

two's comp negate is
invert bits & **add 1**:
0001 => 1110 => 1111

NB +1 for negation is done by setting adder carry in to 1 , so only one adder is needed

```
 0100
 0001  -
   ⇩
 0100
 1111  +
 -----
10011
```

No overflow because 3 is correct signed result
Note that carry out can be 1 (as here) or 0
when there is no signed overflow