Operating Systems

Steven Hand

Michaelmas / Lent Term 2008/09

17 lectures for CST IA

Handout 1

Course Aims

- This course aims to:
  - give you a general understanding of how a computer works,
  - explain the structure and functions of an operating system,
  - illustrate key operating system aspects by concrete example, and
  - prepare you for future courses. . .
- At the end of the course you should be able to:
  - describe the fetch-execute cycle of a computer
  - understand the different types of information which may be stored within a computer memory
  - compare and contrast CPU scheduling algorithms
  - explain the following: process, address space, file.
  - distinguish paged and segmented virtual memory.
  - discuss the relative merits of Unix and NT. . .

Course Outline

- Part I: Computer Organisation
  - Computer Foundations
  - Operation of a Simple Computer.
  - Input/Output.
- Part II: Operating System Functions.
  - Introduction to Operating Systems.
  - Processes & Scheduling.
  - Memory Management.
  - I/O & Device Management.
  - Protection.
  - Filing Systems.
- Part III: Case Studies.
  - Unix.
  - Windows NT.

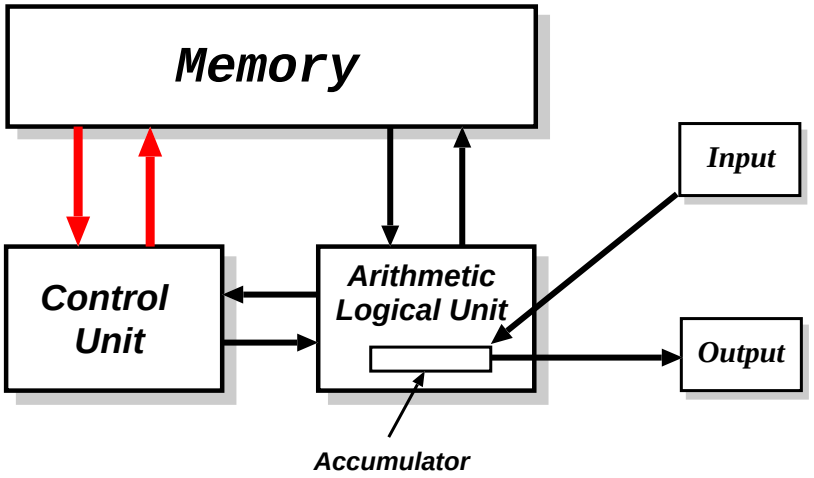
Recommended Reading

- *Structured Computer Organization (3rd Ed)*
  Tannenbaum A S, Prentice-Hall 1990.
- *Computer Organization & Design (2rd Ed)*
  Patterson D and Hennessy J, Morgan Kaufmann 1998.
- *Concurrent Systems* or *Operating Systems*
  Bacon J [and Harris T], Addison Wesley 1997 [2003]
- *Operating Systems Concepts (5th Ed.)*
  Silberschatz A, Peterson J and Galvin P, Addison Wesley 1998.
- *The Design and Implementation of the 4.3BSD UNIX Operating System*
  Leffler S J, Addison Wesley 1989
- *Inside Windows 2000 (3rd Ed)* or *Windows Internals (4th Ed)*
  Solomon D and Russinovich M, Microsoft Press 2000 [2005]

A Chronology of Early Computing

- (several BC): abacus used for counting
- 1614: logarithms disovered (John Napier)
- 1622: invention of the slide rule (Robert Bissaker)
- 1642: First mechanical digital calculator (Pascal)
- Charles Babbage (U. Cambridge) invents:
  - 1812: "Difference Engine"
  - 1833: "Analytical Engine"
- 1890: First electro-mechanical punched card data-processing machine (Hollerith)
- 1905: Vacuum tube/triode invented (De Forest)
- 1935: the relay-based *IBM 601* reaches 1 MPS.
- 1939: *ABC*: first electronic digital computer (Atanasoff & Berry)
- 1941: *Z3*: first programmable computer (Zuse)
- Jan 1943: the *Harvard Mark I* (Aiken)
- Dec 1943: *Colossus* built at 'Station X', Bletchley Park

The Von Neumann Architecture



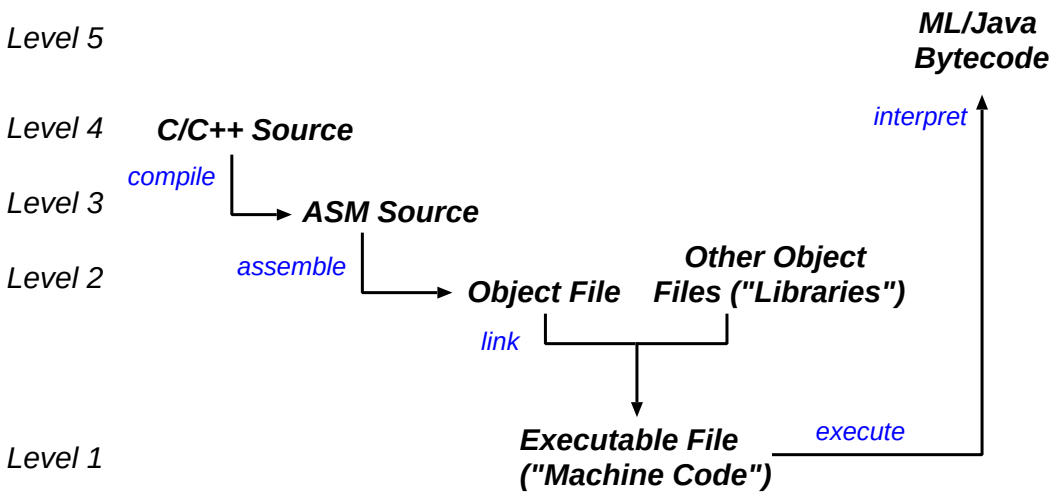
- 1945: *ENIAC* (Eckert & Mauchley, U. Penn):
  - 30 tons, 1000 square feet, 140 kW,
  - 18K vacuum tubes, 20×10-digit accumulators,
  - 100KHz, circa 300 MPS.
  - Used to calculate artillery firing tables.
  - (1946) blinking lights for the media. . .
- But: "programming" is via plugboard ⇒ v. slow.
- 1945: von Neumann drafts "EDVAC" report
  - design for a **stored-program** machine
  - Eckert & Mauchley mistakenly unattributed

Further Progress. . .

- 1947: "point contact" transistor invented (Shockley, Bardeen & Brattain)
- 1949: *EDSAC*, the world's first stored-program computer (Wilkes & Wheeler)
  - 3K vacuum tubes, 300 square feet, 12 kW,
  - 500KHz, circa 650 IPS, 225 MPS.
  - 1024 17-bit words of memory in mercury ultrasonic delay lines.
  - 31 word "operating system" (!)
- 1954: *TRADIC*, first electronic computer without vacuum tubes (Bell Labs)
- 1954: first silicon (junction) transistor (TI)
- 1959: first integrated circuit (Kilby & Noyce, TI)
- 1964: IBM System/360, based on ICs.
- 1971: Intel 4004, first micro-processor (Ted Hoff):
  - 2300 transistors, 60 KIPS.
- 1978: Intel 8086/8088 (used in IBM PC).
- $\sim$ 1980: first VLSI chip ($>$ 100,000 transistors)

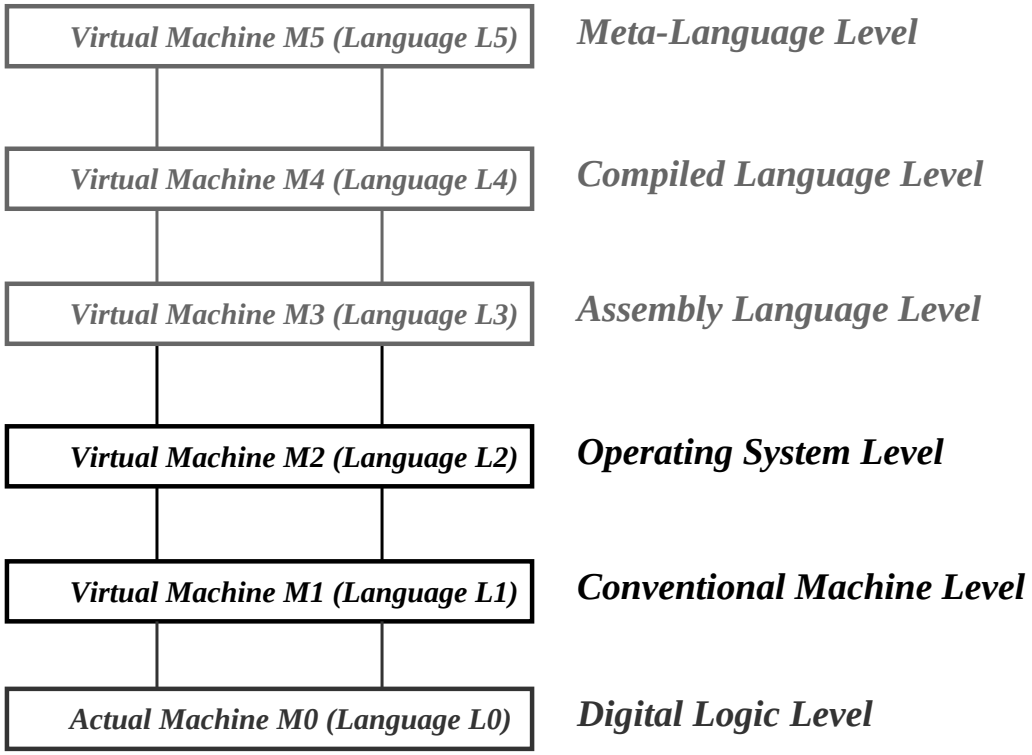
Today: $\sim$ 800M transistors, $\sim 45nm$, ~ 3 GHz.

Languages and Levels



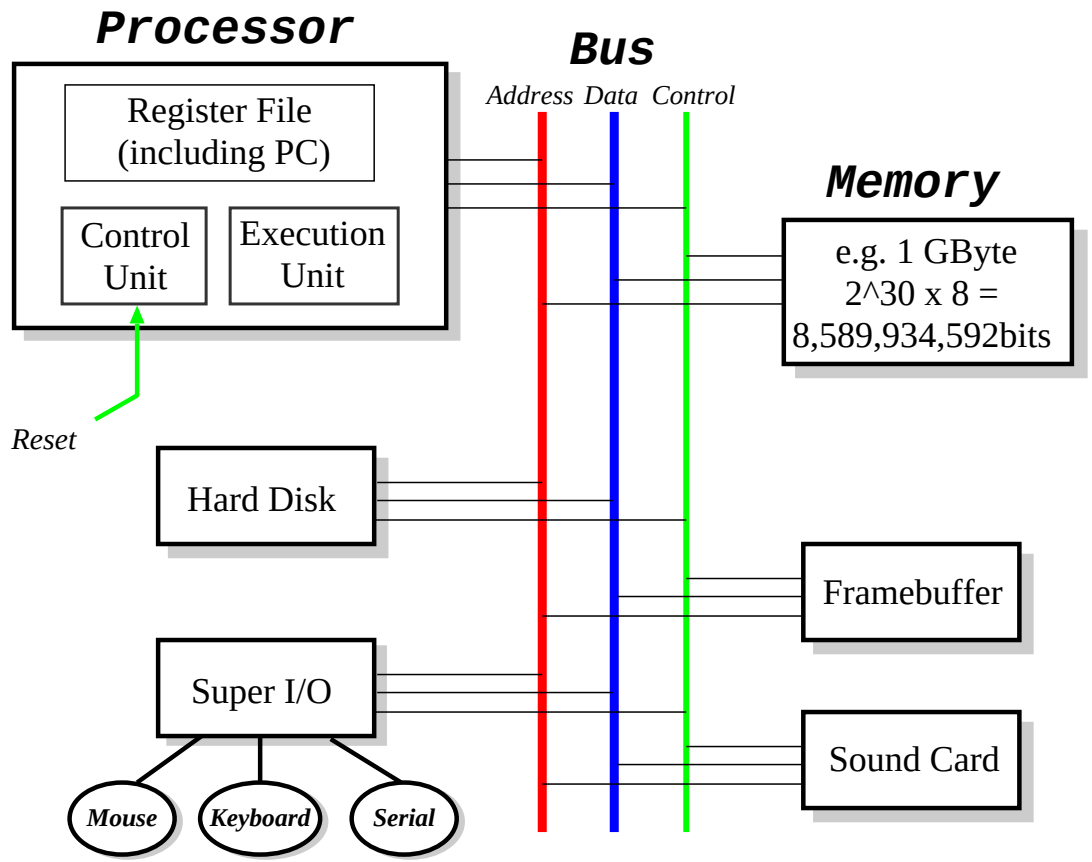
- Modern machines all programmable with a huge variety of different languages.
- e.g. ML, java, C++, C, python, perl, FORTRAN, Pascal, scheme, . . .
- We can describe the operation of a computer at a number of different *levels*; however all of these levels are *functionally equivalent*
  — i.e. can perform the same set of tasks
- Each level relates to the one below via either
  a. translation, or
  b. interpretation.

Layered Virtual Machines



- Conisder a set of machines M_0, M_1, . . . M_n, each built on top of one another
- Each machine M_i understands only machine language L_i.
- Levels 0, -1 covered in Dig. Elec. and (potentially) Physics
- This course focuses on levels 1 and 2.
- NB: all levels useful; none "the truth".

A (Simple) Modern Computer



Processor (CPU): executes programs
Memory: stores both programs & data
Devices: for input and output
Bus: transfers information

Registers and the Register File

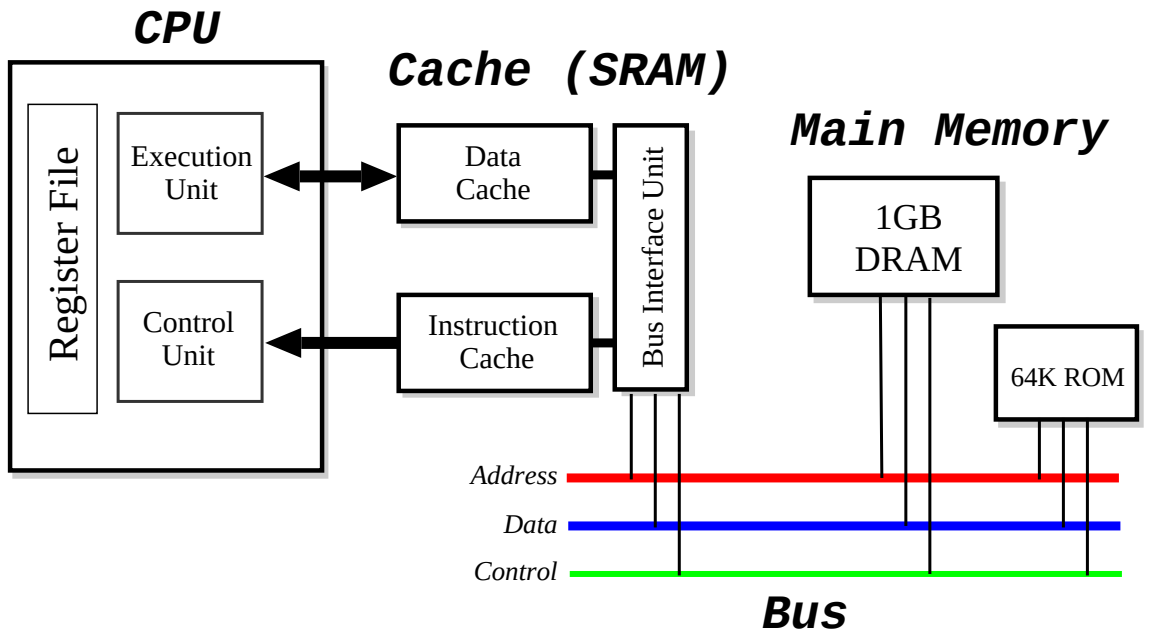
R0	0x5A
R1	0x102034
R2	0x2030ADCB
R3	0x0
R4	0x0
R5	0x2405
R6	0x102038
R7	0x20

R8	0xEA02D1F
R9	0x1001D
R10	0xFFFFFFFF
R11	0x102FC8
R12	0xFF0000
R13	0x37B1CD
R14	0x1
R15	0x20000000

Computers all about operating on information:

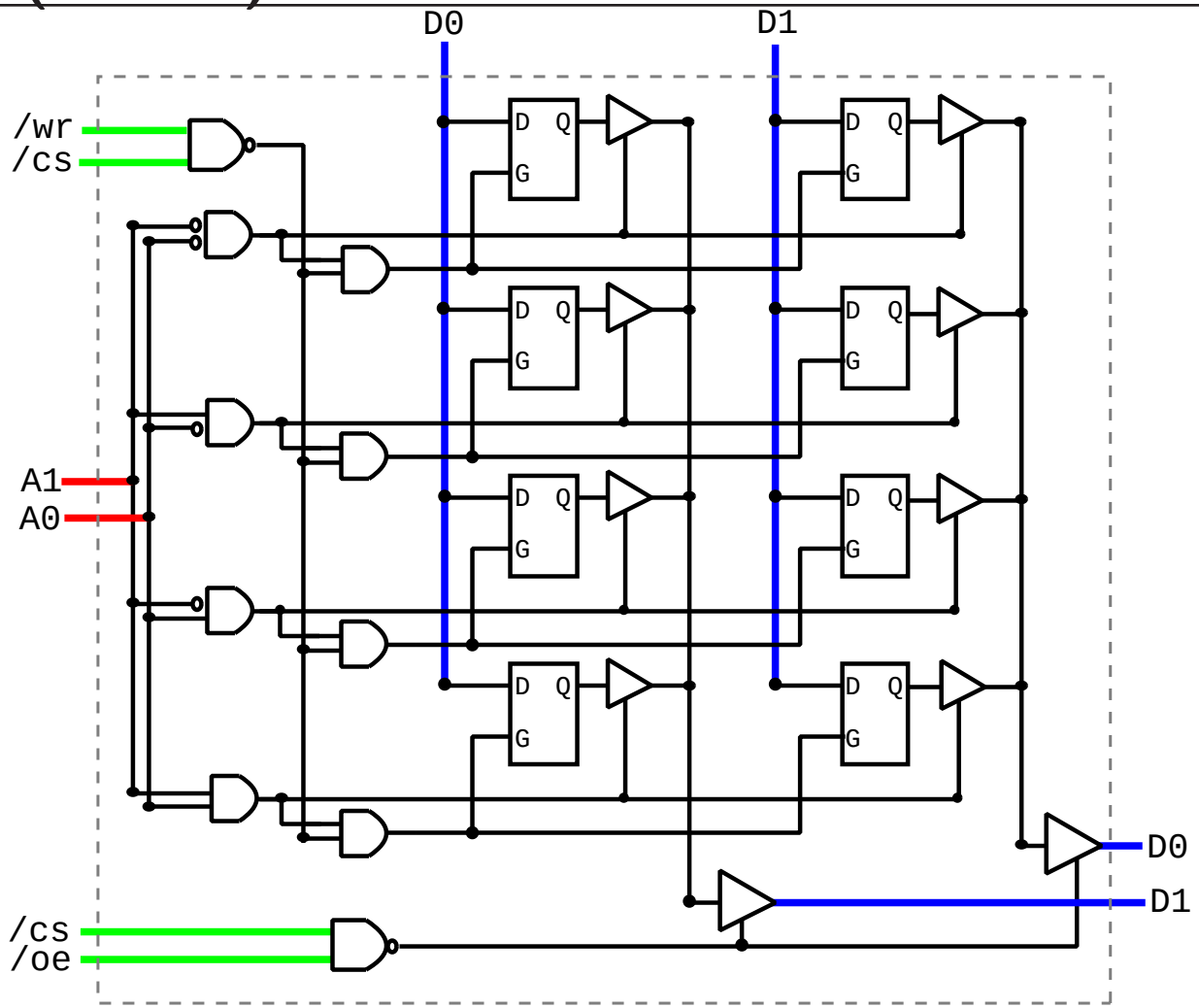
- information arrives into memory from input devices
- memory is a large "byte array" which can hold any information we want
- computer *conceptually* takes values from memory, performs whatever operations, and then stores results back
- in practice, CPU operates on *registers*:
  - a register is an extremely fast piece of on-chip memory
  - modern CPUs have between 8 and 128 registers, each 32/64 bits
  - data values are *loaded* from memory into registers before operation
  - result goes into register; eventually *stored* back to memory again.

Memory Hierarchy



- Use $cache$ between main memory and registers to mask delay of "slow" DRAM
- Cache made from faster SRAM: more expensive, and hence smaller.
  - holds copy of subset of main memory.
- Split of instruction and data at cache level $\Rightarrow$ "Harvard" architecture.
- Cache $\leftrightarrow$ CPU interface uses a custom bus.
- Today have ~ 8MB cache, ~ 4GB RAM.

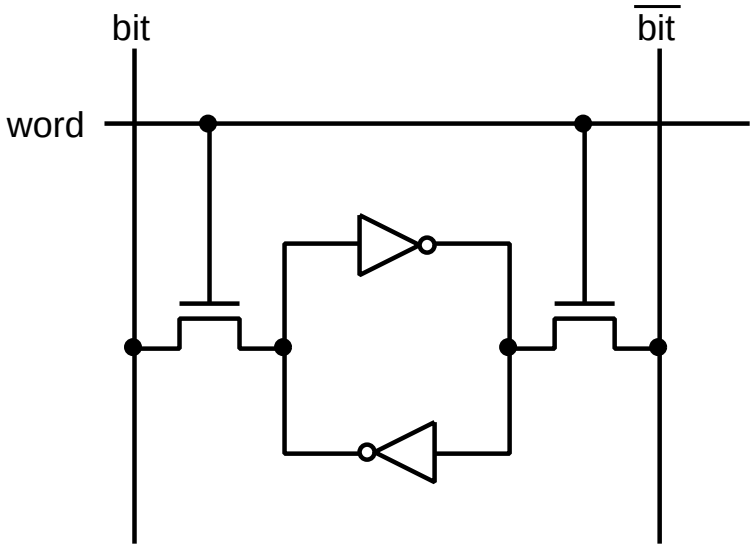
Static RAM (SRAM)



- Relatively fast (currently $5 - 20ns$).
- Logically an array of (transparent) D-latches
  - In reality, only cost ~ 6 transistors per bit.

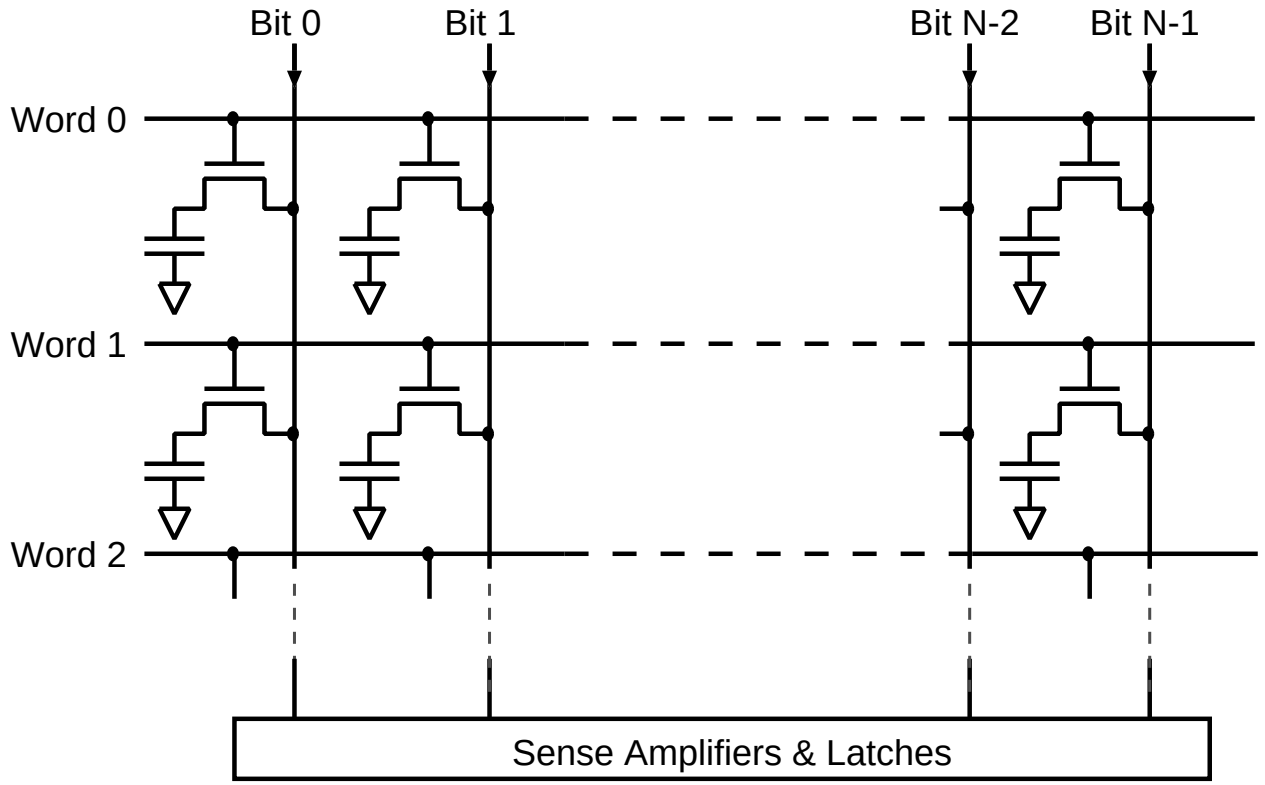
SRAM Reality

SRAM Cell (6T)



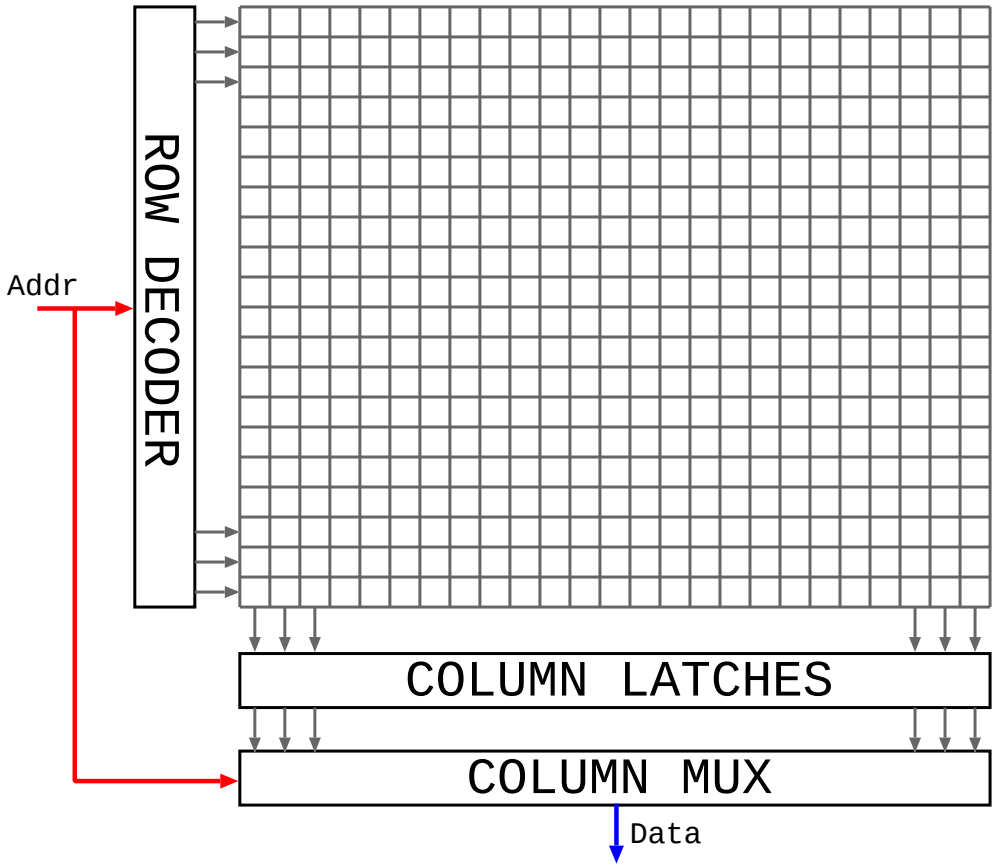
- Data held in cross-coupled inverters.
- One $word$ line and two bit lines.
- To read:
  - precharge both `bit` and $\overline{\texttt{bit}}$, and then strobe `word`
  - $\overline{\texttt{bit}}$ discharged if there was a 1 in the cell; `bit` discharged if there was a 0.
- To write:
  - precharge either `bit` (for "1") or $\overline{\texttt{bit}}$ (for "0"),
  - strobe `word`.

Dynamic RAM (DRAM)



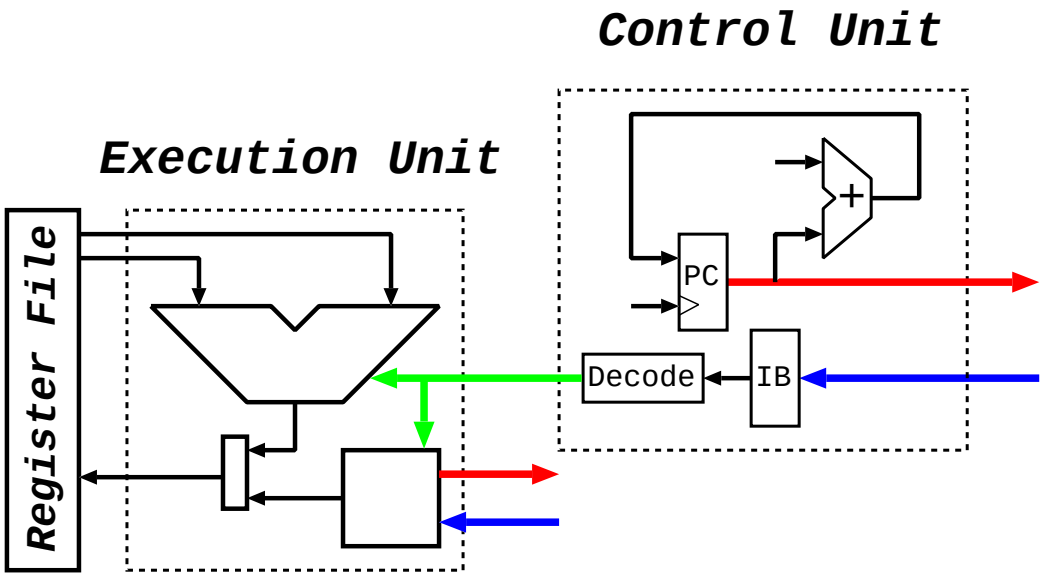
- Use a single transistor to store a bit.
- Write: put value on bit lines, strobe word line.
- Read: pre-charge, strobe word line, amplify, latch.
- "Dynamic": refresh periodically to restore charge.
- Slower than SRAM: typically $50ns - 100ns$.

DRAM Decoding



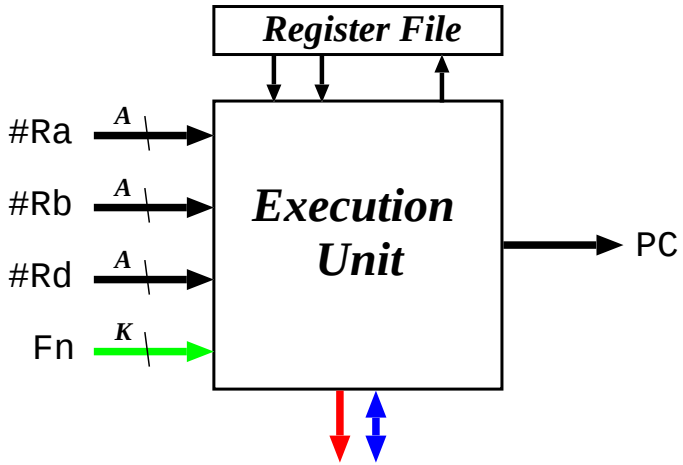
- Two stage: row, then column.
- Usually share address pins: RAS & CAS select decoder or mux.
- FPM, EDO, SDRAM faster for same row reads.

The Fetch-Execute Cycle



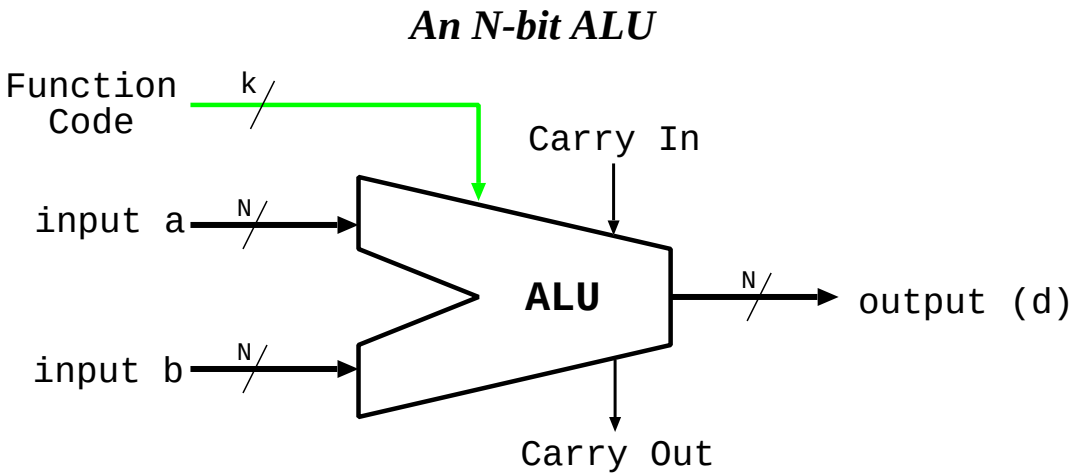
- A special register called *PC* holds a memory address; on reset, initialised to 0.
- Then:
  1. Instruction *fetched* from memory address held in PC into instruction buffer (IB).
  2. Control Unit determines what to do: *decodes* instruction.
  3. Execution Unit *executes* instruction.
  4. PC updated, and back to Step 1.
- Continues pretty much forever. . .

Execution Unit



- The "calculator" part of the processor.
- Broken into parts (*functional units*), e.g.
  - Arithmetic Logic Unit (ALU).
  - Shifter/Rotator.
  - Multiplier.
  - Divider.
  - Memory Access Unit (MAU).
  - Branch Unit.
- Choice of functional unit determined by signals from control unit.

Arithmetic Logic Unit



- Part of the execution unit.
- Inputs from register file; output to register file.
- Performs simple two-operand functions:
  - a + b;
  - a - b
  - a AND b
  - a OR b
  - etc.
- Typically perform *all* possible functions; use function code to select (mux) output.

Number Representation

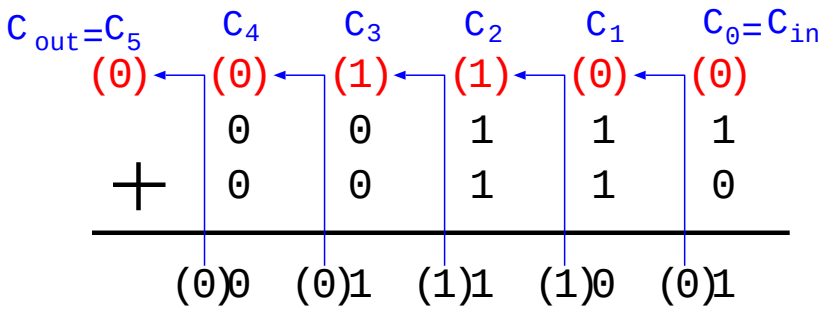
0000_2	0_{16}	0110_2	6_{16}	1100_2	C_{16}
0001_2	1_{16}	0111_2	7_{16}	1101_2	D_{16}
0010_2	2_{16}	1000_2	8_{16}	1110_2	E_{16}
0011_2	3_{16}	1001_2	9_{16}	1111_2	F_{16}
0100_2	4_{16}	1010_2	A_{16}	10000_2	10_{16}
0101_2	5_{16}	1011_2	B_{16}	10001_2	11_{16}

- a n-bit register $b_{n-1}b_{n-2}\ldots b_1b_0$ can represent 2^n different values.
- Call b_{n-1} the *most significant bit* (msb), b_0 the *least significant bit* (lsb).
- Unsigned numbers: val $= b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0$, e.g. $1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$.
- Represents values from 0 to $2^n - 1$ inclusive.
- For large numbers, binary is unwieldy: use hexadecimal (base 16).
- To convert, group bits into groups of 4, e.g. $1111101010_2 = 0011|1110|1010_2 = 3EA_{16}$.
- Often use "$0x$" prefix to denote hex, e.g. $0x107$.
- Can use dot to separate large numbers into 16-bit chunks, e.g. $0x3FF.FFFF$.

Number Representation (2)

- What about $signed$ numbers? Two main options:
- Sign & magnitude:
  - top (leftmost) bit flags if negative; remaining bits make value.
  - e.g. byte $10011011_2 \rightarrow -0011011_2 = -27$.
  - represents range $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$, and the bonus value -0 (!).
- 2's complement:
  - to get $-x$ from x, invert every bit and add 1.
  - e.g. $+27 = 00011011_2 \Rightarrow -27 = (11100100_2 + 1) = 11100101_2$.
  - treat $1000\ldots000_2$ as -2^{n-1}.
  - represents range -2^{n-1} to $+(2^{n-1}-1)$
- Note:
  - in both cases, top-bit means "negative".
  - both representations depend on n;
- In practice, all modern computers use 2's complement. . .

Unsigned Arithmetic



- (we use 5-bit registers for simplicity)
- Unsigned addition: C_n means "carry"; usually refer to this as $\mathbf{C}$, the "carry flag":

```
   00101    5          11110   30
 + 00111    7        +  00111    7
------------          -------------
0  01100   12        1  00101    5
```

- Unsigned subtraction: $\overline{C_n}$ means "borrow":

```
   11110   30          00111    7
 + 00101  -27        +  10110  -10
------------          -------------
1  00011    3        0  11101   29
```

Signed Arithmetic

- In signed arithmetic, carry flag useless. . .
  - Instead we use the *overflow* flag, $\mathbf{V} = (C_n \oplus C_{n-1})$.
  - Also have *negative* flag, $\mathbf{N} = b_{n-1}$ (i.e. the msb).
- Signed addition:

```
   00101    5          01010   10
 + 00111    7        +  00111    7
------------          -------------
0  01100   12        0  10001  -15
------------          -------------
   0                     1
```

- Signed subtraction:

```
   01010   10          10110  -10
 + 11001   -7        +  10110  -10
------------          -------------
1  00011    3        1  01100   12
------------          -------------
   1                     0
```

- In overflow cases the sign of the result is always wrong (i.e. the N bit is inverted).

Arithmetic & Logical Instructions

- Some common ALU instructions are:

Mnemonic		C/Java Equivalent	Mnemonic		C/Java Equivalent
`and`	$d \leftarrow a, b$	`d = a & b;`	`add`	$d \leftarrow a, b$	`d = a + b;`
`xor`	$d \leftarrow a, b$	`d = a ^ b;`	`sub`	$d \leftarrow a, b$	`d = a - b;`
`orr`	$d \leftarrow a, b$	`d = a | b;`	`rsb`	$d \leftarrow a, b$	`d = b - a;`
`bis`	$d \leftarrow a, b$	`d = a | b;`	`shl`	$d \leftarrow a, b$	`d = a << b;`
`bic`	$d \leftarrow a, b$	`d = a & (~b);`	`shr`	$d \leftarrow a, b$	`d = a >> b;`

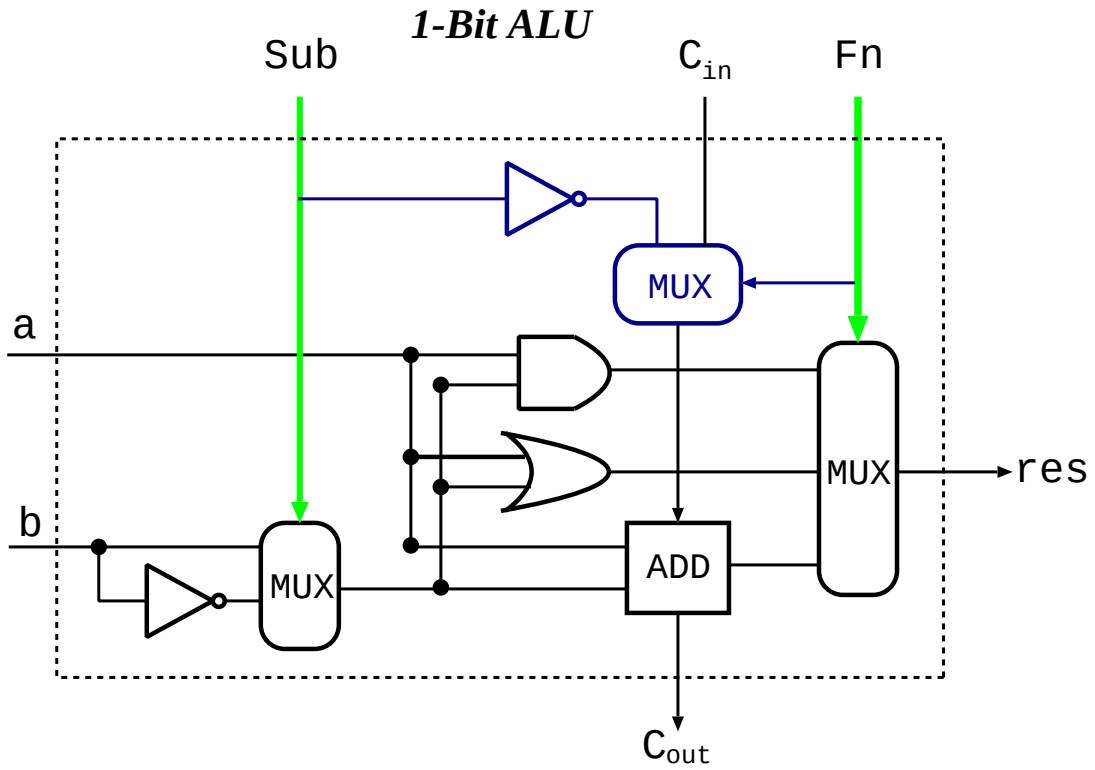
  Both d and a *must* be registers; b can be a register or a (small) constant.

- Typically also have `addc` and `subc`, which handle carry or borrow (for multi-precision arithmetic), e.g.

```
add  d0, a0, b0    // compute "low" part.
addc d1, a1, b1    // compute "high" part.
```

- May also get:
  - Arithmetic shifts: `asr` and `asl`(?)
  - Rotates: `ror` and `rol`.

A 1-bit ALU Implementation



- Eight possible functions (4 types):
  1. a AND b, a AND $\overline{b}$.
  2. a OR b, a OR $\overline{b}$.
  3. $a + b$, $a + b$ with carry.
  4. $a - b$, $a - b$ with borrow.
- To make n-bit ALU bit, connect together (use carry-lookahead on adders).

Conditional Execution

- Seen $\mathbf{C}, \mathbf{N}, \mathbf{V}$ flags; now add $\mathbf{Z}$ (zero), logical NOR of all bits in output.
- Can predicate execution based on (some combination) of flags, e.g.

```
subs d, a, b          // compute d = a - b
beq  proc1            // if equal, goto proc1
br   proc2            // otherwise goto proc2
```

  Java equivalent approximately:

```
if (a==b) proc1() else proc2();
```

- On most computers, mainly limited to branches.
- On ARM (and IA64), everything conditional, e.g.

```
sub   d, a, b         // compute d = a - b
moveq d, #5           // if equal, d = 5;
movne d, #7           // otherwise d = 7;
```

  Java equiv: `d = (a==b) ?  5 :  7;`

- "Silent" versions useful when don't really want result, e.g. `tst`, `teq`, `cmp`.

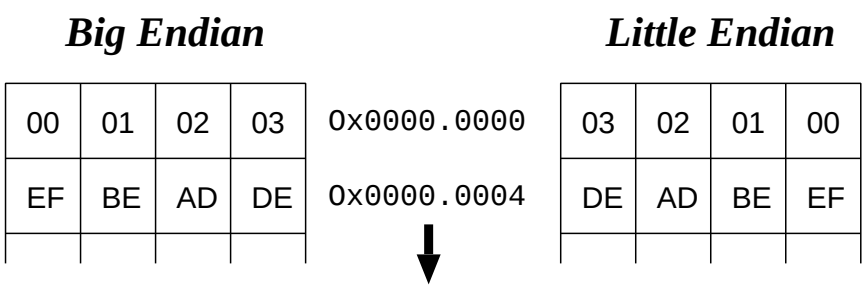
Condition Codes

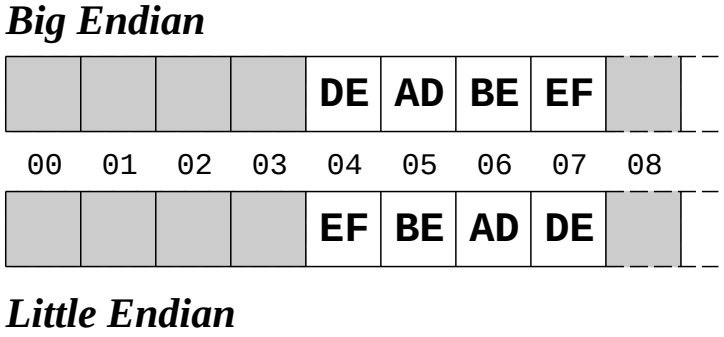
Suffix	Meaning	Flags
`EQ`, `Z`	Equal, zero	$\mathbf{Z}$ == 1
`NE`, `NZ`	Not equal, non-zero	$\mathbf{Z}$ == 0
`MI`	Negative	$\mathbf{N}$ == 1
`PL`	Positive (incl. zero)	$\mathbf{N}$ == 0
`CS`, `HS`	Carry, higher or same	$\mathbf{C}$ == 1
`CC`, `LO`	No carry, lower	$\mathbf{C}$ == 0
`VS`	Overflow	$\mathbf{V}$ == 1
`VC`	No overflow	$\mathbf{V}$ == 0
`HI`	Higher	$\mathbf{C}$ == 1 && $\mathbf{Z}$ == 0
`LS`	Lower or same	$\mathbf{C}$ == 0 || $\mathbf{Z}$ == 1
`GE`	Greater than or equal	$\mathbf{N}$ == $\mathbf{V}$
`GT`	Greater than	$\mathbf{N}$ == $\mathbf{V}$ && $\mathbf{Z}$ == 0
`LT`	Less than	$\mathbf{N}$!= $\mathbf{V}$
`LE`	Less than or equal	$\mathbf{N}$!= $\mathbf{V}$ || $\mathbf{Z}$ == 1

- `HS`, `LO`, etc. used for unsigned comparisons (recall that $\overline{C}$ means "borrow").
- `GE`, `LT`, etc. used for signed comparisons: check both $\mathbf{N}$ and $\mathbf{V}$ so always works.

Loads & Stores

- Have variable sized values, e.g. bytes (8-bits), words (16-bits), longwords (32-bits) and quadwords (64-bits).
- Load or store instructions usually have a suffix to determine the size, e.g. '`b`' for byte, '`w`' for word, '`l`' for longword.
- When storing > 1 byte, have two main options: big endian and little endian; e.g. storing longword `0xDEADBEEF` into memory at address `0x4`.



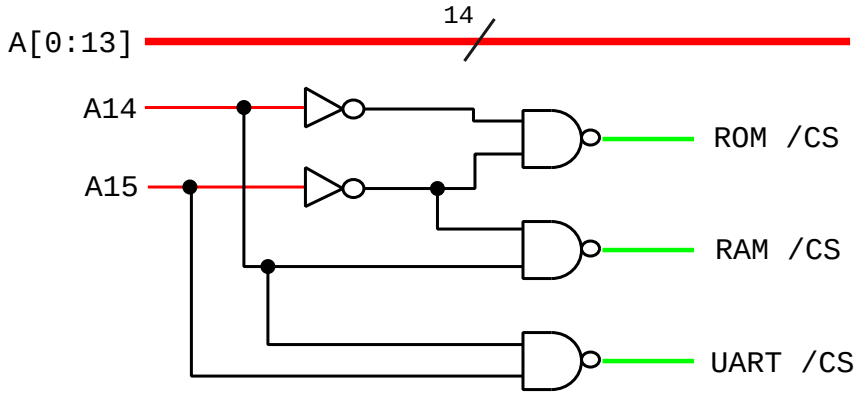


If read back a *byte* from address `0x4`, get `0xDE` if big-endian, or `0xEF` if little-endian. If you always load and store things of the same size, things are fine.

- Today have x86 little-endian; Sparc big-endian; Mips & ARM either.
- Annoying. . . and burns a considerable number of CPU cycles on a daily basis. . .

Accessing Memory

- To load/store values need the *address* in memory.
- Most modern machines are *byte addressed*: consider memory a big array of 2^A bytes, where A is the number of address lines in the bus.
- Lots of things considered "memory" via address decoder, e.g.



- Typically each device decodes only a subset of low address lines, e.g.

Device	Size	Data	Decodes
ROM	1024 bytes	32-bit	`A[2:9]`
RAM	16384 bytes	32-bit	`A[2:13]`
UART	256 bytes	8-bit	`A[0:7]`

Addressing Modes

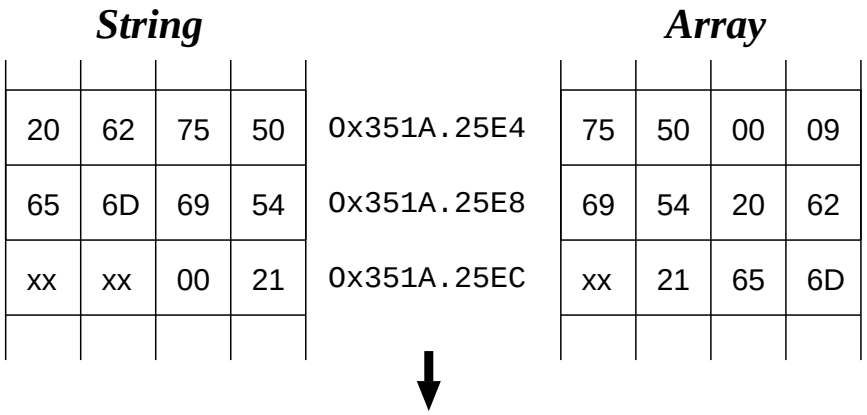
- An *addressing mode* tells the computer where the data for an instruction is to come from.
- Get a wide variety, e.g.

Register:	`add r1, r2, r3`
Immediate:	`add r1, r2, #25`
PC Relative:	`beq 0x20`
Register Indirect:	`ldr r1, [r2]`
" + Displacement:	`str r1, [r2, #8]`
Indexed:	`movl r1, (r2, r3)`
Absolute/Direct:	`movl r1, $0xF1EA0130`
Memory Indirect:	`addl r1, ($0xF1EA0130)`

- Most modern machines are *load/store* $\Rightarrow$ only support first five:
  - allow at most one memory ref per instruction
  - (there are very good reasons for this)
- Note that CPU generally doesn't care *what* is being held within the memory.
- i.e. up to *programmer* to interpret whether data is an integer, a pixel or a few characters in a novel.

Representing Text

- Two main standards:
  1. ASCII: 7-bit code holding (English) letters, numbers, punctuation and a few other characters.
  2. Unicode: 16-bit code supporting practically all international alphabets and symbols.
- ASCII default on many operating systems, and on the early Internet (e.g. e-mail).
- Unicode becoming more popular (esp UTF-8!).
- In both cases, represent in memory as either *strings* or *arrays*: e.g. "Pub Time!"



- 0x4920776973682069742076173203a2d28

Floating Point (1)

- In many cases want to deal with very large or very small numbers.
- Use idea of "scientific notation", e.g. $n = m \times 10^e$
  - m is called the $mantissa$
  - e is called the $exponent$.

  e.g. $C = 3.01 \times 10^8$ m/s.
- For computers, use binary i.e. $n = m \times 2^e$, where m includes a "binary point".
- Both m and e can be positive or negative; typically
  - sign of mantissa given by an additional $sign$ bit.
  - exponent is stored in a $biased$ ($excess$) format.

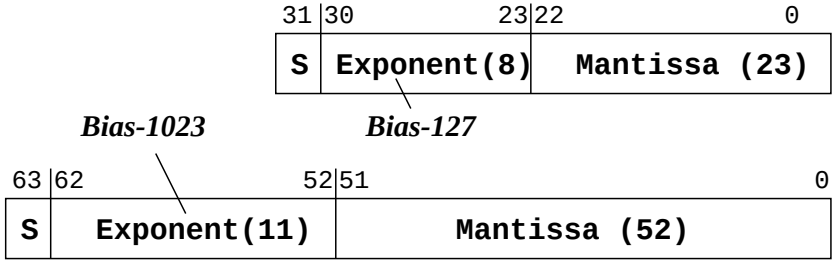
$\Rightarrow$ use $n = (-1)^s m \times 2^{e-b}$, where $0 \leq m < 2$ and b is the bias.

- e.g. 4-bit mantissa & 3-bit bias-3 exponent allows positive range $[0.001_2 \times 2^{-3}, 1.111_2 \times 2^4]$

$=$ [$(\frac{1}{8})(\frac{1}{8})$, $(\frac{15}{8})16$], or [$\frac{1}{64}$, 30]

Floating Point (2)

- In practice use IEEE floating point with $normalised$ mantissa $m = 1.xx\ldots x_2$ $\Rightarrow$ use $n = (-1)^s((1+m) \times 2^{e-b})$,
- Both single (`float`) and double (`double`) precision:



- IEEE fp reserves $e = 0$ and $e =$ max:
  - ± 0 (!): both e and m zero.
  - $\pm\infty$: $e =$ max, m zero.
  - NaNs : $e =$ max, m non-zero.
  - $denorms$: $e = 0$, m non-zero
- Normal positive range $[2^{-126}, \sim 2^{128}]$ for single, or $[2^{-1022}, \sim 2^{1024}]$ for double.
- NB: still only $2^{32}/2^{64}$ values — just spread out.

Data Structures

- Records / structures: each field stored as an offset from a *base address*.
- Variable size structures: explicitly store addresses (*pointers*) inside structure, e.g.

```
datatype rec = node of int * int * rec
             |    leaf of int;

val example = node(4, 5, node(6, 7, leaf(8)));
```

- Imagine `example` is stored at address `0x1000`:

Address	Value	Comment
0x0F30	0xFFFF	Constructor tag for a leaf
0x0F34	8	Integer 8
⋮		
0x0F3C	0xFFFE	Constructor tag for a node
0x0F40	6	Integer 6
0x0F44	7	Integer 7
0x0F48	0x0F30	Address of inner node
⋮		
0x1000	0xFFFE	Constructor tag for a node
0x1004	4	Integer 4
0x1008	5	Integer 5
0x100C	0x0F3C	Address of inner node

Instruction Encoding

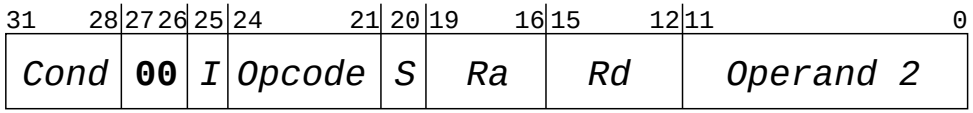
- An instruction comprises:

  a. an *opcode*: specify what to do.
  b. zero or more *operands*: where to get values

  e.g. `add r1, r2, r3` ≡

1010111	001	010	011

- Old machines (and x86) use *variable length* encoding for low code density.
- Other modern machines (e.g. ARM) use fixed length encoding for simplicity



`and r13, r13, #31` = `0xe20dd01f` =

1110	00	1	0000	0	1101	1101	000000011111

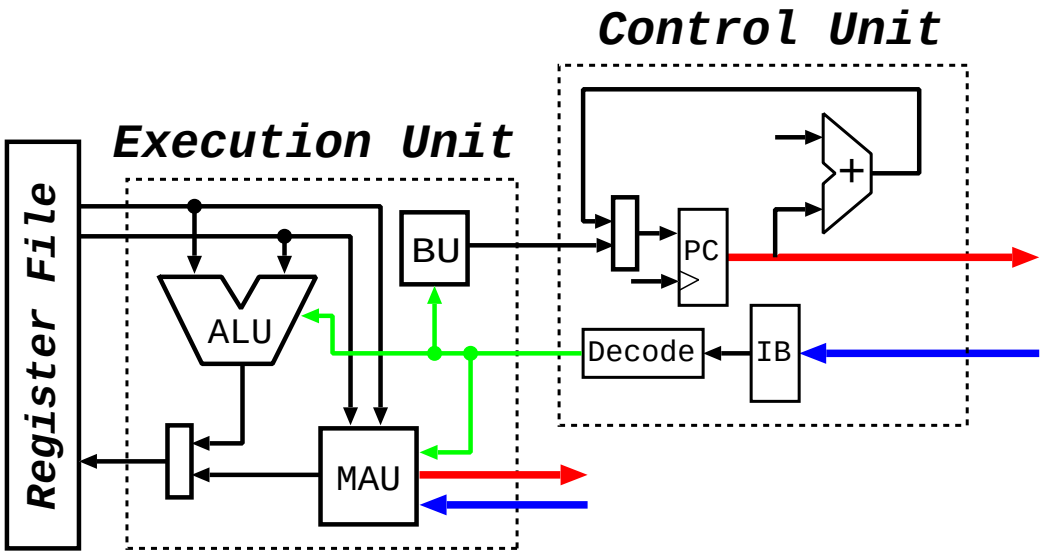
`bic r3, r3, r2` = `0xe1c33002` =

1110	00	0	1110	0	0011	0011	000000000010

`cmp r1, r2` = `0xe1510002` =

1110	00	0	1010	1	0001	0000	000000000010

Fetch-Execute Cycle Revisited

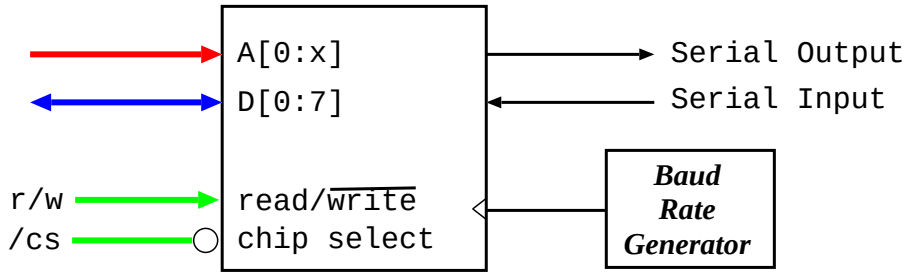


1. CU fetches & decodes instruction and generates (a) control signals and (b) operand information.
2. In EU, control signals select functional unit ("instruction class") and operation.
3. If ALU, then read 1–2 registers, perform op, and (probably) write back result.
4. If BU, test condition and (maybe) add value to PC.
5. If MAU, generate address ("addressing mode") and use bus to read/write value.
6. Repeat *ad infinitum*.

Input/Output Devices

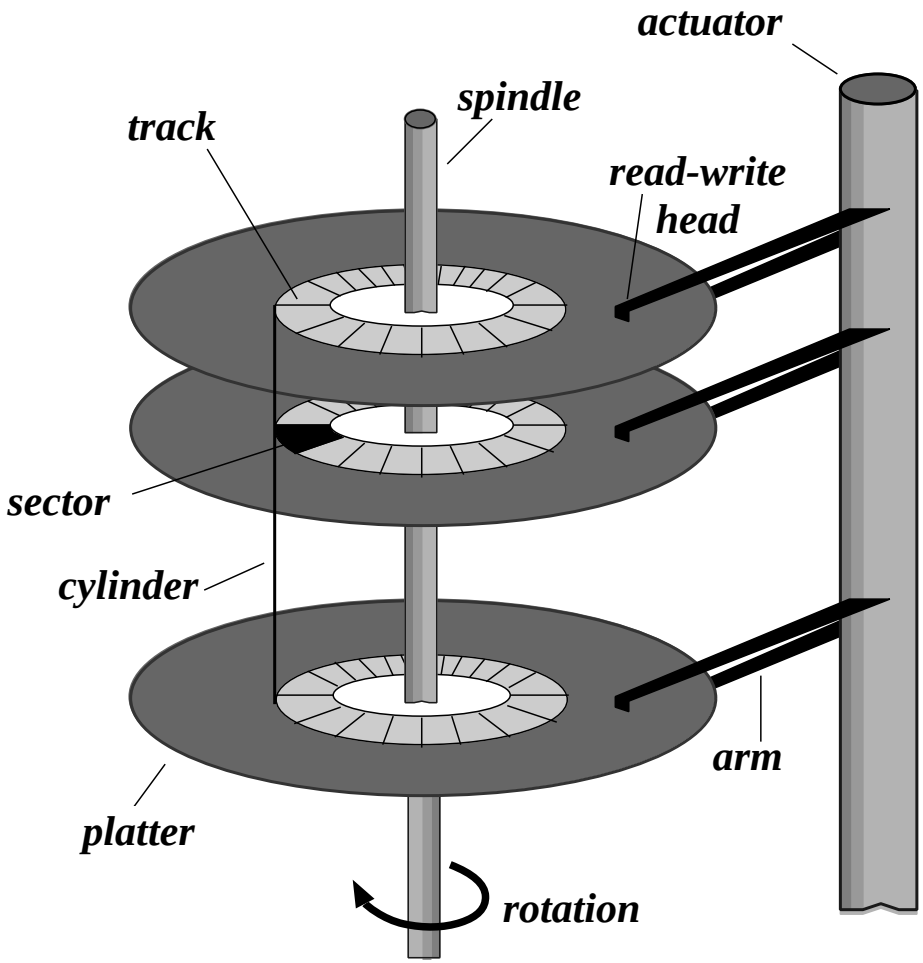
- Devices connected to processor via a bus (e.g. ISA, PCI, AGP).
- Includes a wide range:
  - Mouse,
  - Keyboard,
  - Graphics Card,
  - Sound card,
  - Floppy drive,
  - Hard-Disk,
  - CD-Rom,
  - Network card,
  - Printer,
  - Modem
  - etc.
- Often two or more stages involved (e.g. IDE, SCSI, RS-232, Centronics, etc.)

UARTs



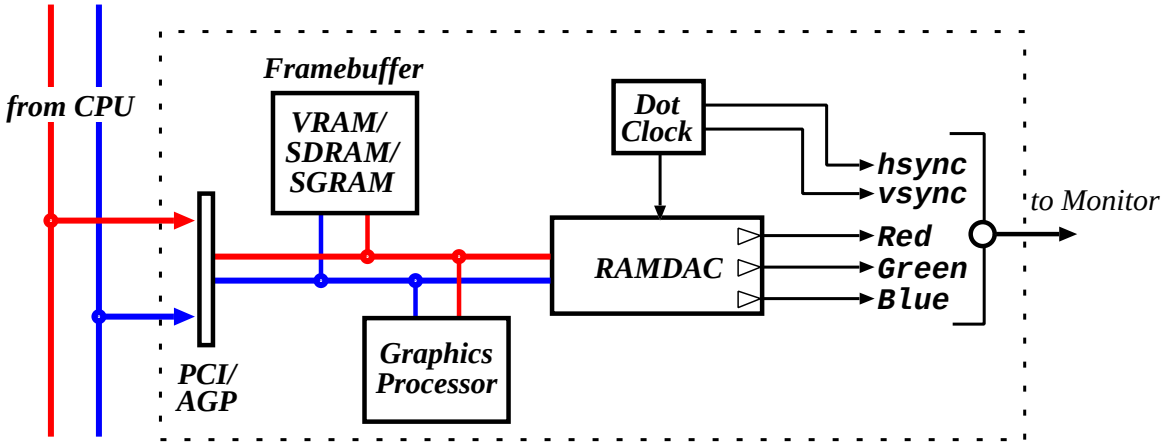
- Universal Asynchronous Receiver/Transmitter:
  - stores 1 or more bytes internally.
  - converts parallel to serial.
  - outputs according to RS-232.
- Various baud rates (e.g. 1,200 – 115,200)
- Slow and simple. . . and very useful.
- Make up "serial ports" on PC.
- Max throughput $\sim$ 14.4KBytes; variants up to 56K (for modems).

Hard Disks



- Whirling bits of (magnetized) metal. . .
- Rotate 3,600 – 12,000 times a minute.
- Capacity $\sim$ 250 GBytes ($\approx 250 \times 2^{30} bytes$).

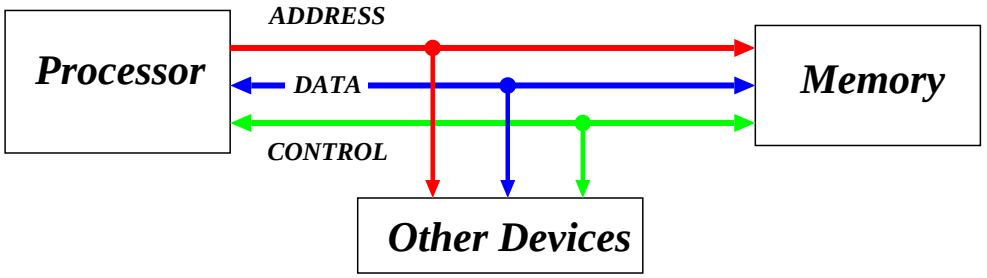
Graphics Cards



- Essentially some RAM (framebuffer) and some digital-to-analogue circuitry (RAMDAC).
- RAM holds array of $pixels$: picture elements.
- Resolutions e.g. 640x480, 800x600, 1024x768, 1280x1024, 1600x1200.
- Depths: 8-bit (LUT), 16-bit (RGB=555, 24-bit (RGB=888), 32-bit (RGBA=888).
- Memory requirement $= x \times y \times$ depth, e.g. 1280x1024 @ 16bpp needs 2560KB.

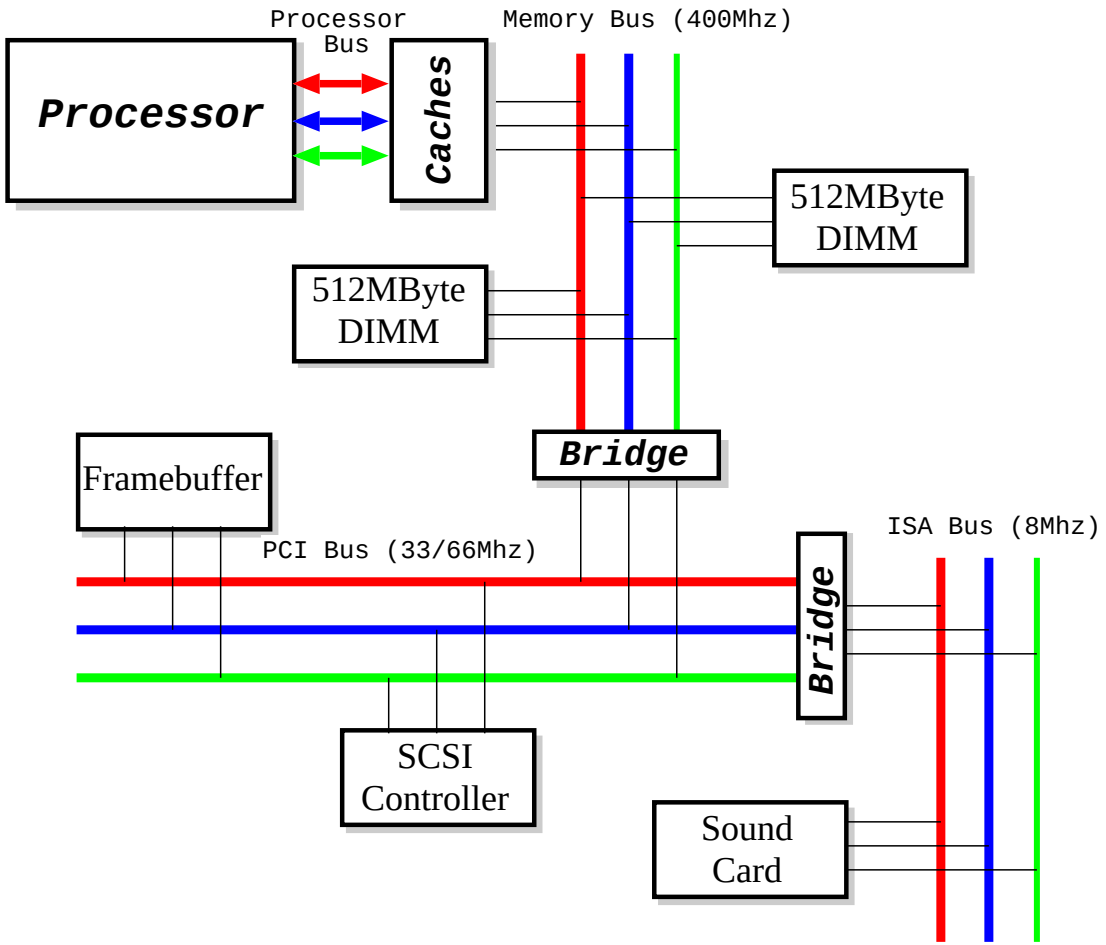
$\Rightarrow$ full-screen 50Hz video requires 125 MBytes/s (or $\sim$ 1Gbit/s).

Buses



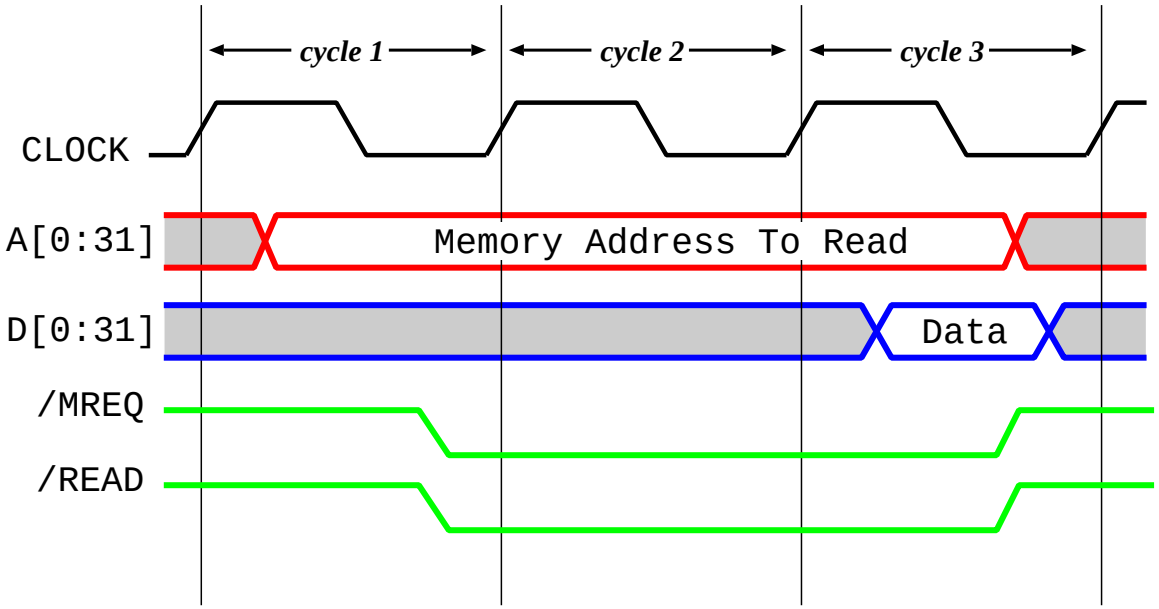
- Bus = collection of *shared* communication wires:
  - ✔ low cost.
  - ✔ versatile / extensible.
  - ✘ potential bottle-neck.
- Typically comprises address lines, data lines and control lines (+ power/ground).
- Operates in a *master-slave* manner, e.g.
  1. master decides to e.g. read some data.
  2. master puts addr onto bus and asserts 'read'
  3. slave reads addr from bus and retrieves data.
  4. slave puts data onto bus.
  5. master reads data from bus.

Bus Hierarchy



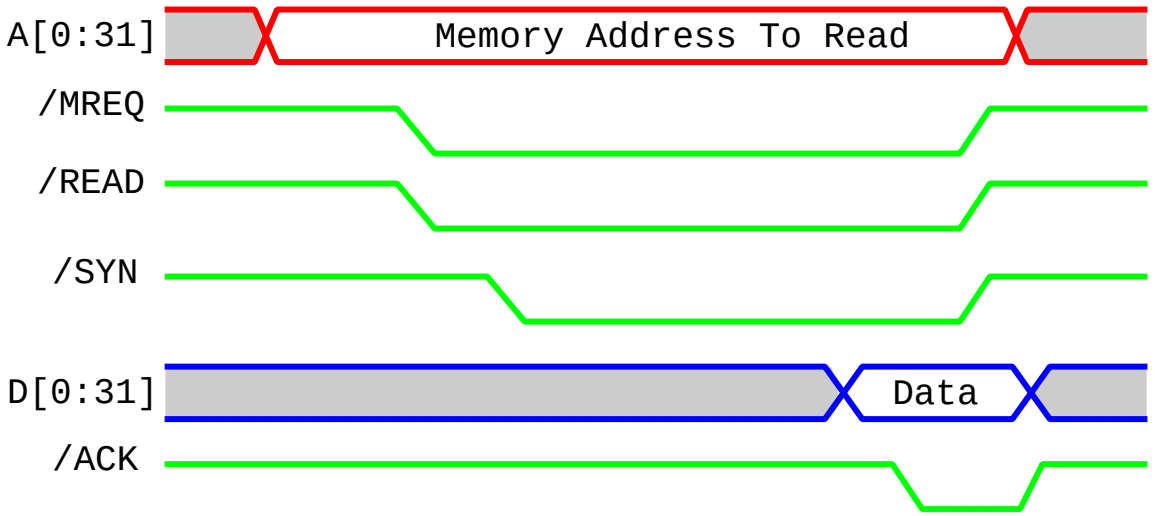
- In practice, have lots of different buses with different characteristics e.g. data width, max #devices, max length.
- Most buses are *synchronous* (share clock signal).

Synchronous Buses



- Figure shows a read transaction which requires three bus cycles.
  1. CPU puts address onto address lines and, after settle, asserts control lines.
  2. Memory fetches data from address.
  3. Memory puts data on data lines, CPU latches value and then deasserts control lines.
- If device not fast enough, can insert *wait states*.
- Faster clock/longer bus can give *bus skew*.

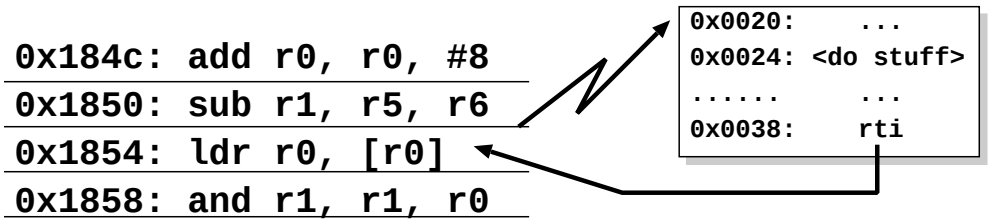
Asynchronous Buses



- Asynchronous buses have no shared clock; instead work by *handshaking*, e.g.
  - CPU puts address onto address lines and, after settle, asserts control lines.
  - next, CPU asserts /SYN to say everything ready.
  - once memory notices /SYN, it fetches data from address and puts it onto bus.
  - memory then asserts /ACK to say data is ready.
  - CPU latches data, then deasserts /SYN.
  - finally, Memory deasserts /ACK.
- More handshaking if multiplex address & data. . .

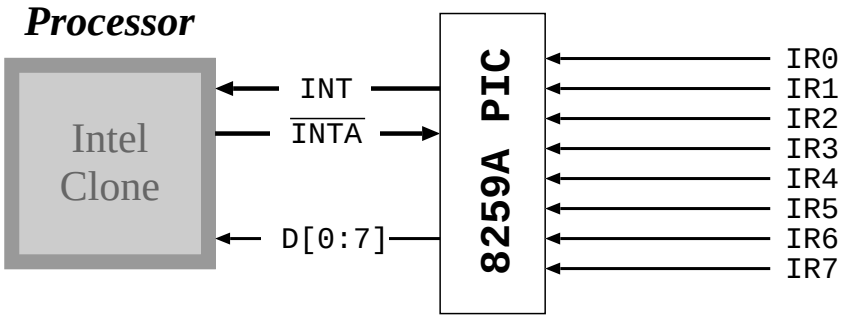
Interrupts

- Bus reads and writes are $transaction$ based: CPU requests something and waits until it happens.
- But e.g. reading a block of data from a hard-disk takes $\sim 2ms$, which is $\sim 5,000,000$ clock cycles!
- $Interrupts$ provide a way to decouple CPU requests from device responses.
  1. CPU uses bus to make a request (e.g. $writes$ some special values to a device).
  2. Device goes off to get info.
  3. Meanwhile CPU continues doing other stuff.
  4. When device finally has information, raises an $interrupt$.
  5. CPU uses bus to read info from device.
- When interrupt occurs, CPU $vectors$ to handler, then $resumes$ using special instruction, e.g.



Interrupts (2)

- Interrupt lines ($\sim 4 - 8$) are part of the bus.
- Often only 1 or 2 pins on chip $\Rightarrow$ need to encode.
- e.g. ISA & x86:



1. Device asserts `IRx`.
2. PIC asserts `INT`.
3. When CPU can interrupt, strobes `INTA`.
4. PIC sends interrupt number on `D[0:7]`.
5. CPU uses number to index into a table in memory which holds the addresses of handlers for each interrupt.
6. CPU saves registers and jumps to handler.

Direct Memory Access (DMA)

- Interrupts good, but even better is a device which can read and write processor memory *directly*.
- A generic DMA "command" might include
  - source address
  - source increment / decrement / do nothing
  - sink address
  - sink increment / decrement / do nothing
  - transfer size
- Get one interrupt at end of data transfer
- DMA channels may be provided by devices themselves:
  - e.g. a disk controller
  - pass disk address, memory address and size
  - give instruction to read or write
- Also get "stand-alone" programmable DMA controllers.

Summary

- Computers made up of four main parts:
  1. Processor (including register file, control unit and execution unit),
  2. Memory (caches, RAM, ROM),
  3. Devices (disks, graphics cards, etc.), and
  4. Buses (interrupts, DMA).
- Information represented in all sorts of formats:
  - signed & unsigned integers,
  - strings,
  - floating point,
  - data structures,
  - instructions.
- Can (hopefully) understand all of these at some level, but gets pretty complex.

$\Rightarrow$ to be able to actually use a computer, need an operating system.