Operating Systems (Handout 1)

CST 1A - Michaelmas 2009

MWF @ 12

Hopkinson Lecture Theatre

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. . .

Course Objectives

- At the end of the course you'll 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 Organization**
  - Computer Foundations
  - Operation of a Simple Computer
  - Input / Output
  - MIPS Assembly Language
- **Part II: Operating System Functions**
  - Introduction to Operating Systems.
  - Processes & Scheduling.
  - Memory Management.
  - I/O & Device Management.
  - Filing Systems.
- **Part III: Case Studies**
  - Unix.
  - Windows NT.

Recommended Reading

- ***Structured Computer Organization*** (3rd Ed), Tannenbaum, Prentice-Hall 1990.
- ***Computer Organization & Design*** (2rd Ed), Patterson and Hennessy, Morgan Kaufmann 1998.
- ***Operating Systems***, Bacon and Harris, Addison Wesley 2003
- ***Operating Systems Concepts*** (5th Ed.), Silberschatz, Peterson and Galvin, Addison Wesley 1998.
- ***The Design and Implementation of the 4.3BSD UNIX Operating System***, Leffler, Addison Wesley 1989
- ***Windows Internals*** (4th Edition), Solomon and Russinovich, Microsoft Press 2005

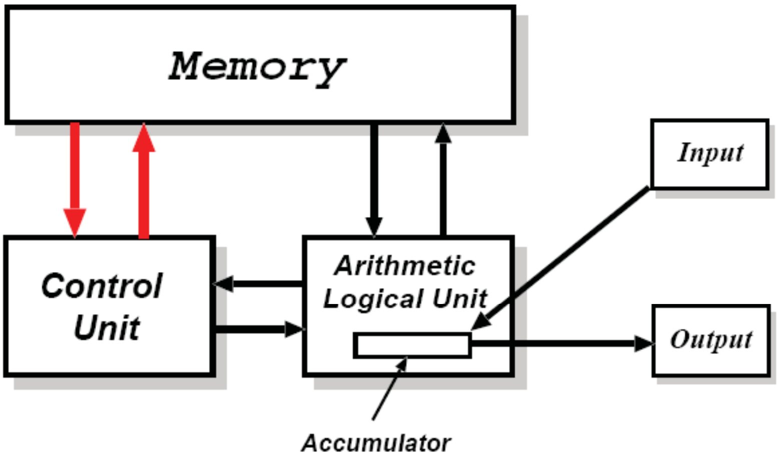
A Chronology of Early Computing

- (several BC): abacus used for counting
- **1614**: logarithms discovered (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)

The War Years…

- **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
- **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 plug-board: tedious and slow

The Von Neumann Architecture



- **1945**: von Neumann drafts "EDVAC" report
  - design for a ***stored-program*** machine
  - Eckert & Mauchley mistakenly unattributed

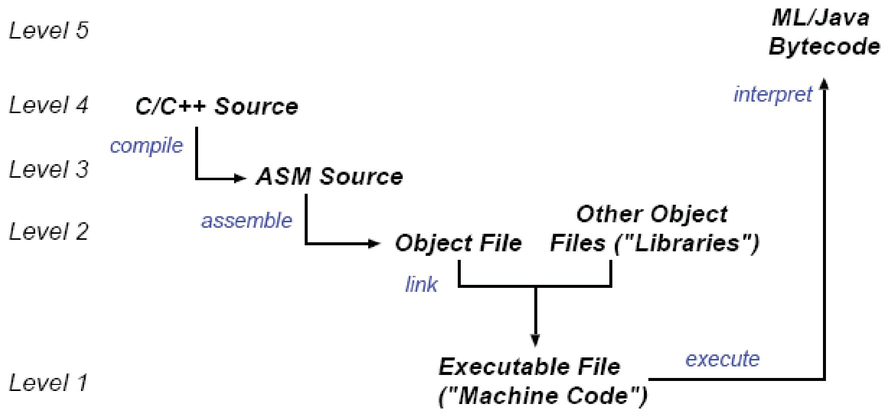
Further Progress…

- **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 – early DRAM ;-)
  - 31 word “operating system” (!)
- **1954**: TRADIC, first electronic computer without vacuum tubes (Bell Labs)

The Silicon Age

- **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).
- **1980**: first VLSI chip (> 100,000 transistors)
- Today: ~800M transistors, 45nm, ~3 GHz.

Languages and Levels


Level 5

Level 4

Level 3

Level 2

Level 1

C/C++ Source

compile

ASM Source

assemble

Object File

Other Object Files ("Libraries")

link

Executable File ("Machine Code")

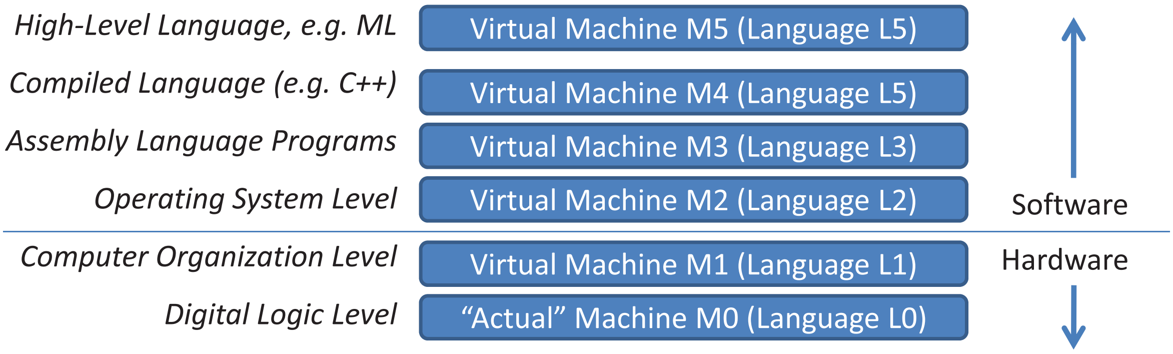
execute

interpret

ML/Java Bytecode

- Computers programmable with variety of different languages.
  - e.g. ML, java, C/C++, python, perl, FORTRAN, Pascal, . . .
- Can describe the operation of a computer at a number of different levels; however all levels are ***functionally equivalent***
- Levels relate via either (a) translation, or (b) interpretation.

Layered Virtual Machines


High-Level Language, e.g. ML — Virtual Machine M5 (Language L5)

Compiled Language (e.g. C++) — Virtual Machine M4 (Language L5)

Assembly Language Programs — Virtual Machine M3 (Language L3)

Operating System Level — Virtual Machine M2 (Language L2)

Software

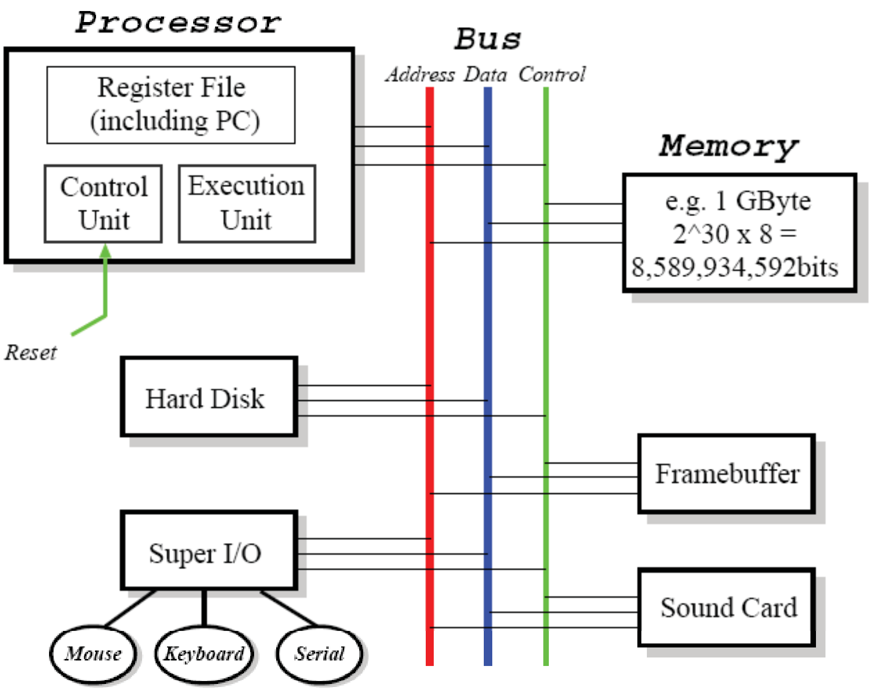
Hardware

Computer Organization Level — Virtual Machine M1 (Language L1)

Digital Logic Level — “Actual” Machine M0 (Language L0)

- Consider a set of machines M0, M1, . . . Mn, each built on top of one another
  - Machine M*i* understands only machine language L*i*
  - Levels 0, -1 covered in Digital Electronics, Physics
- **This course focuses on levels 1 and 2** (and a little 3)
- **NB**: all levels useful; none “the truth”.

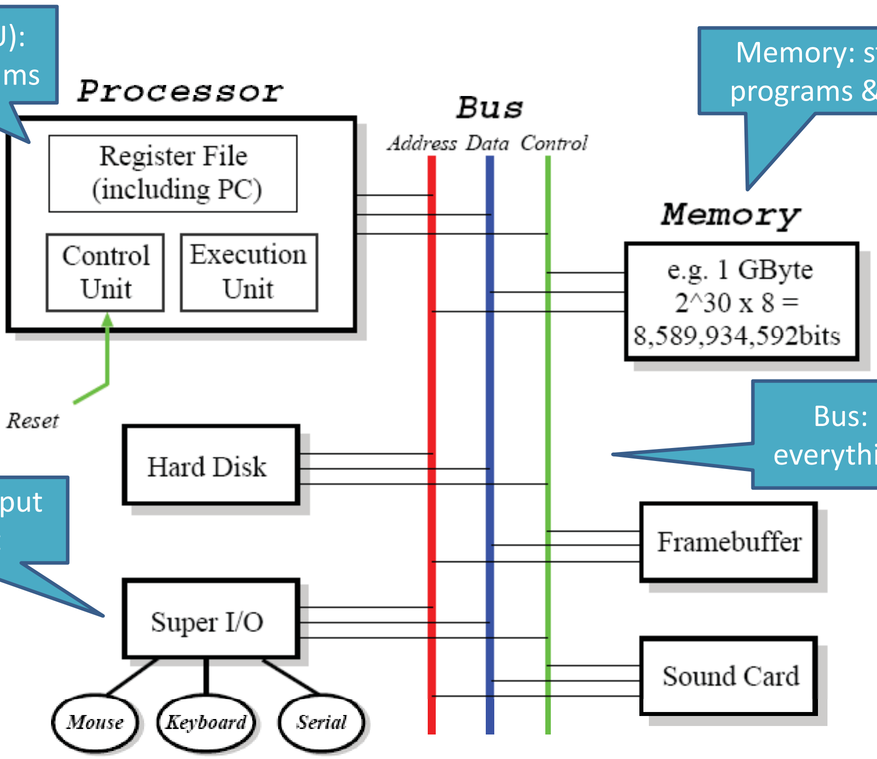
A (Simple) Modern Computer



A (Simple) Modern Computer

Processor (CPU): executes programs

Memory: stores programs & data



Bus: connects everything together

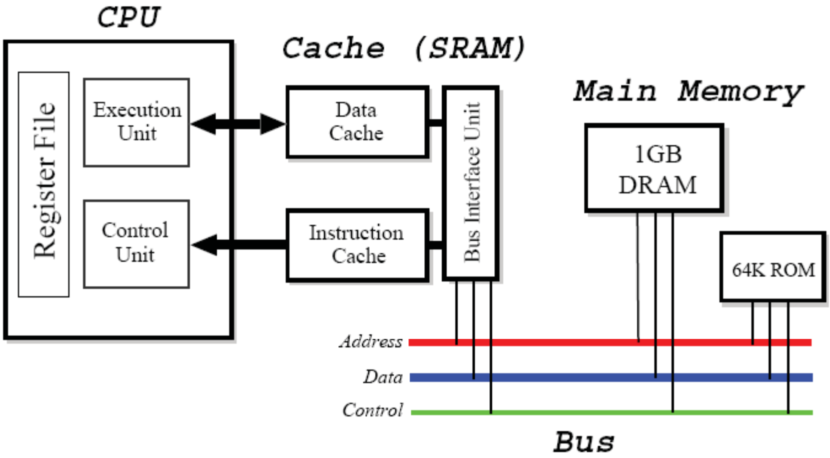
Devices: for input and output

Registers and the Register File

R00	0x5A	R08	0xEA02D1F
R01	0x102034	R09	0x1001D
R02	0x2030ADCB	R10	0xFFFFFFFF
R03	0x0	R11	0x1020FC8
R04	0x0	R12	0xFF0000
R05	0x2405	R13	0x37B1CD
R06	0x102038	R14	0x1
R07	0x20	R15	0x20000000

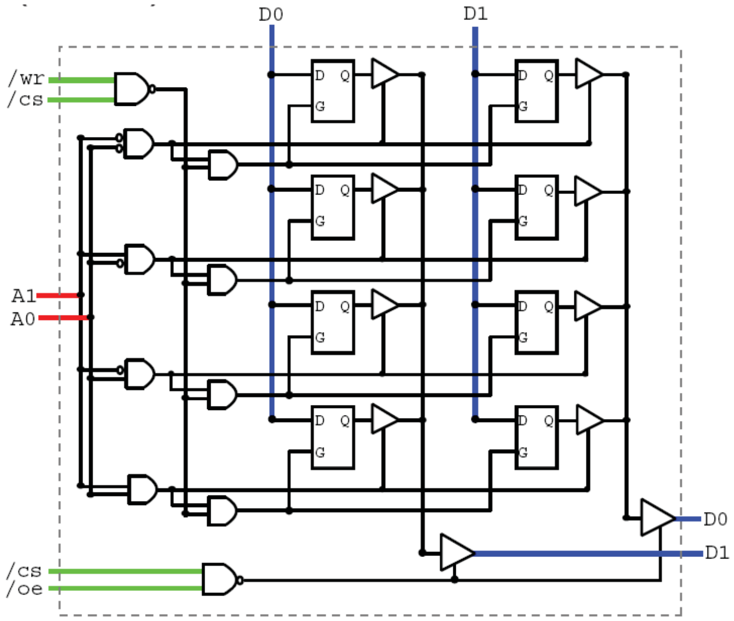
- Computers all about operating on information:
  - information arrives into memory from input devices
  - memory is a large “byte array” which can hold anything we want
- Computer conceptually takes values from memory, performs whatever operations, and then stores results back
- In practice, CPU operates 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 & registers to hide “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:
  - “Harvard” architecture.
- Cache <-> CPU interface uses a custom bus.
- Today have ~8MB cache, ~4GB RAM.

Static RAM (SRAM)

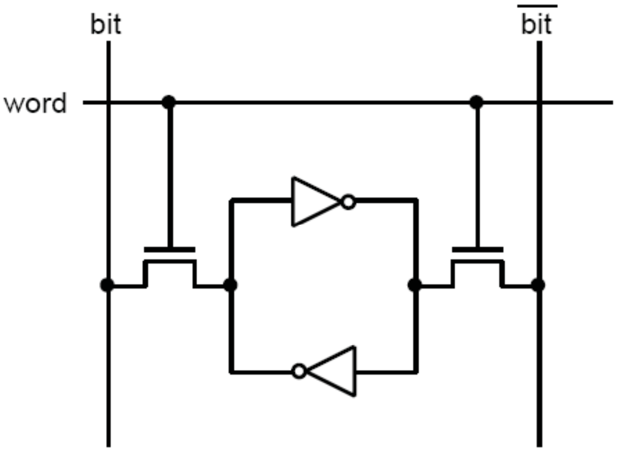


- Relatively fast (currently 5 – 20*ns*).
- Logically an array of (transparent) D-latches
  - In reality, only cost ~6 transistors per bit.

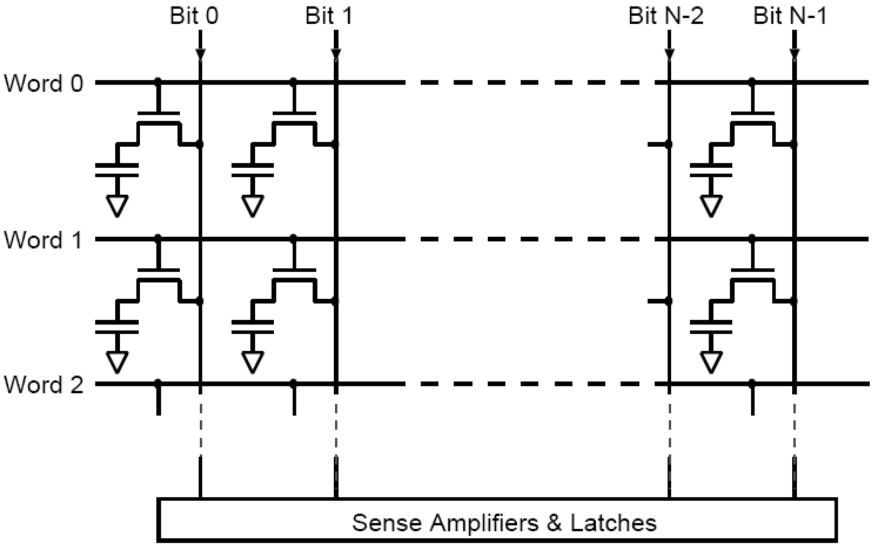
SRAM Reality

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

SRAM Cell (6T)

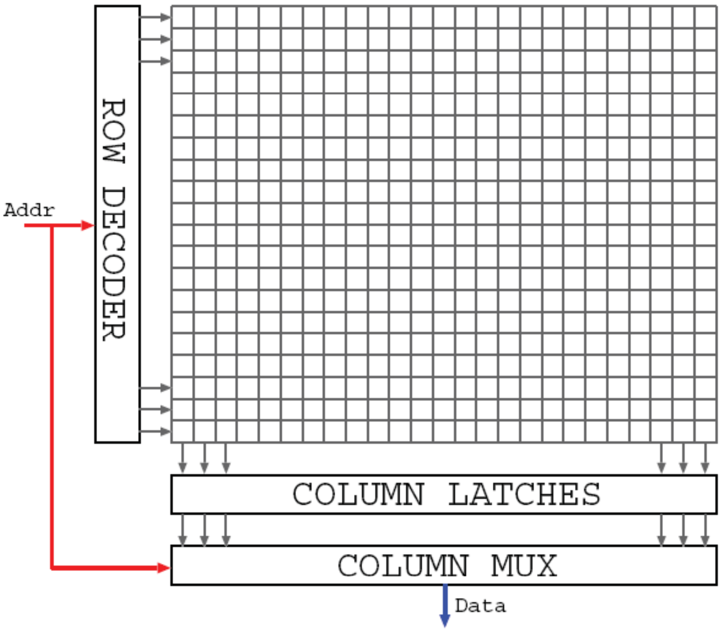


Dynamic RAM (DRAM)



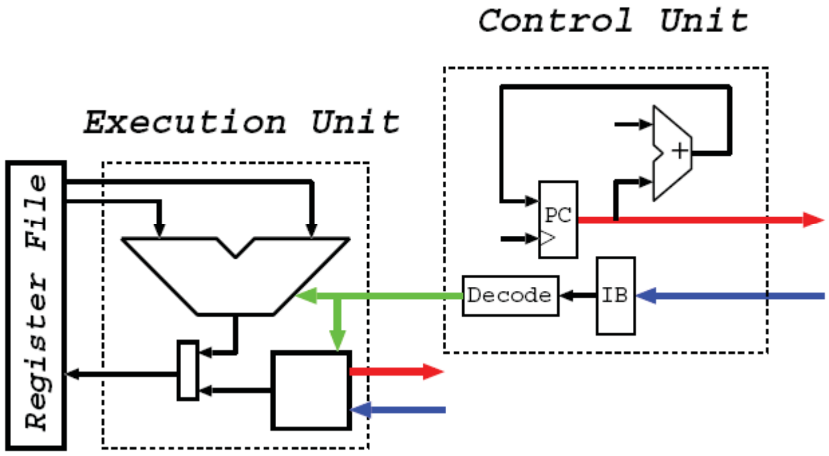
- Use a **single transistor** to store a bit.
- **Write**: put value on bit lines, strobe word line.
- **Read**: pre-charge, strobe word line, amplify, latch.
- “Dynamic”: refresh periodically to restore charge.
- Slower than SRAM: typically 50ns – 100ns.

DRAM Decoding



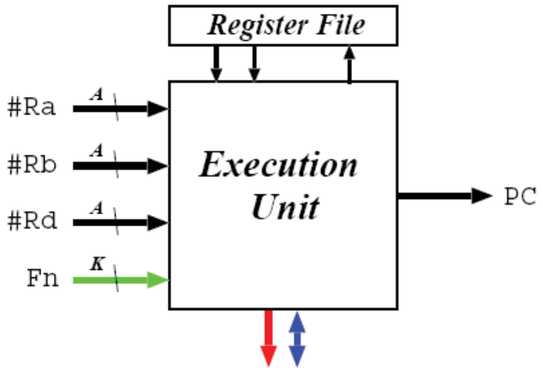
- Two stage: row, then column.
- Usually share address pins: RAS & CAS select decoder or mux.
- FPM, EDO, SDRAM faster for same row reads.

The Fetch-Execute Cycle



- A special register called PC holds a memory address
  - on reset, initialized 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...

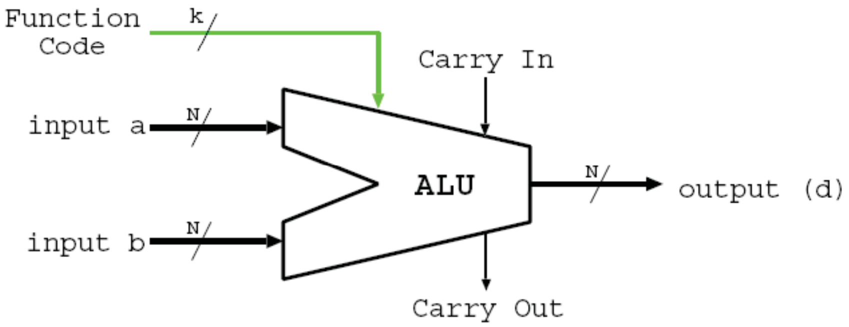
The Execution Unit



- 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 (ALU)

An N-bit ALU



- 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}

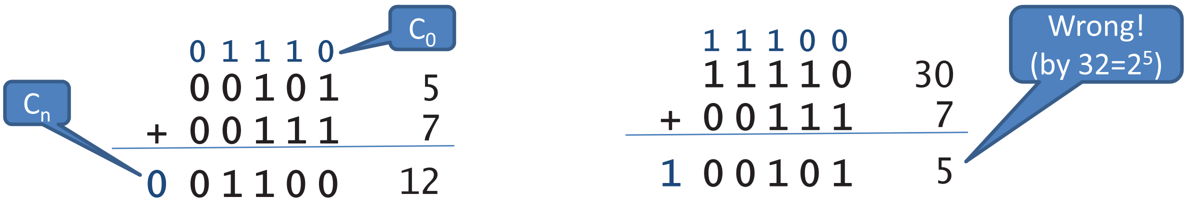
- n-bit register $b_{n-1}b_{n-2} \dots b_1b_0$ can represent 2^n different values.
- Call b_{n-1} the ***most significant bit*** (msb), b_0 the ***least significant bit*** (lsb).
- Unsigned numbers: val = $b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_12^1 + b_02^0$
  - e.g. $1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$.
- 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

Signed Numbers

- What about signed numbers? Two main options:
- **Sign & magnitude**:
  - top (leftmost) bit flags if negative; remaining bits make value.
  - e.g. byte 10011011_2 → $-0011011_2 = -27$.
  - 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...

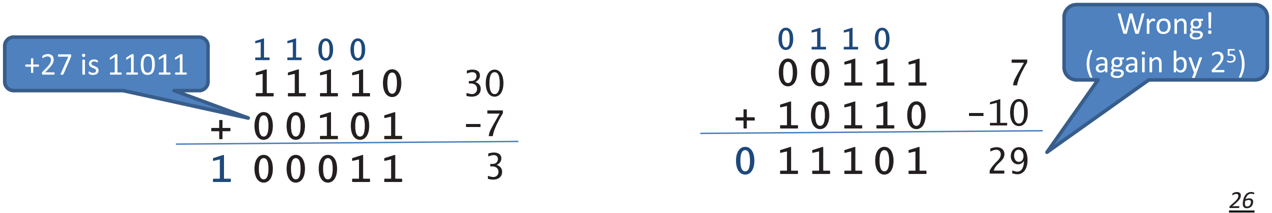
Unsigned Arithmetic

- Unsigned addition (using 5-bit registers)



```
      0 1 1 1 0               1 1 1 0 0
      0 0 1 0 1    5          1 1 1 1 0    30
    + 0 0 1 1 1    7        + 0 0 1 1 1     7
    0 0 1 1 0 0   12        1 0 0 1 0 1     5
```

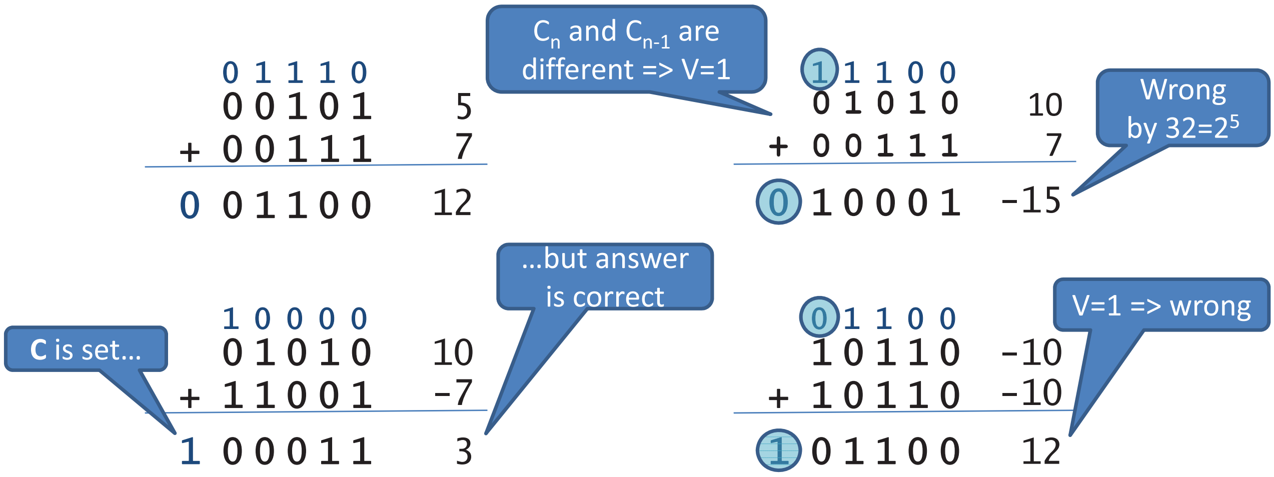
- Carry bits C_0 (=C_{in}), C_1, C_2, ... C_n (=C_{out})
  - usually refer to C_n as **C**, the ***carry flag***
  - In addition, if **C** is 1, we got the wrong answer
- Unsigned subtraction: if **C** is 0, we "borrowed"



```
      1 1 0 0                 0 1 1 0
      1 1 1 1 0   30          0 0 1 1 1     7
    + 0 0 1 0 1   -7        + 1 0 1 1 0   -10
    1 0 0 0 1 1    3        0 1 1 1 0 1    29
```

Signed Arithmetic

- In signed arithmetic, **C** on its own is useless…
  - Instead use ***overflow flag***, **V** = $C_n \oplus C_{n-1}$



```
     0 1 1 1 0
     0 0 1 0 1      5
   + 0 0 1 1 1      7
  0  0 1 1 0 0     12
```

C_n and C_{n-1} are different => V=1

```
  1 1 1 0 0
    0 1 0 1 0      10
  + 0 0 1 1 1       7
  0 1 0 0 0 1     -15
```

Wrong by $32=2^5$

```
     1 0 0 0 0
     0 1 0 1 0     10
   + 1 1 0 0 1     -7
  1  0 0 0 1 1      3
```

C is set…

…but answer is correct

```
  0 1 1 0 0
    1 0 1 1 0     -10
  + 1 0 1 1 0     -10
  1 0 1 1 0 0      12
```

V=1 => wrong

  - ***Negative flag*** **N** = C_{n-1} (i.e. msb) flips on overflow

Arithmetic and Logical Instructions

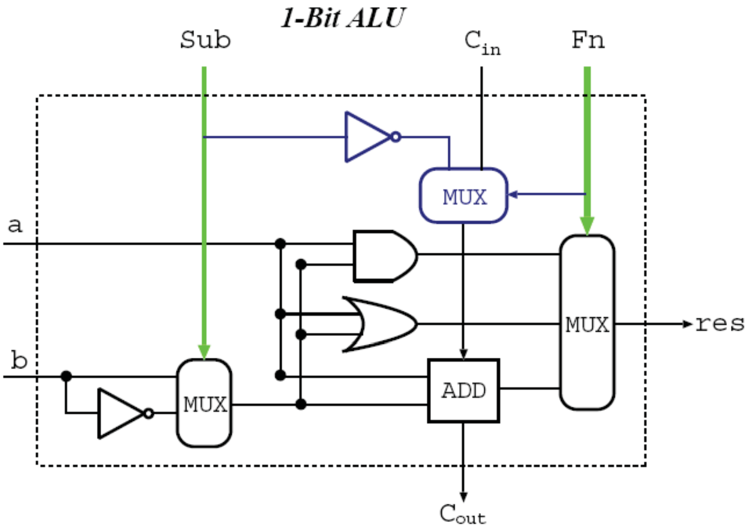
Mnemonic	C/Java Equivalent	Mnemonic	C/Java Equivalent
`and` *d* ← *a*, *b*	`d = a & b;`	`add` *d* ← *a*, *b*	`d = a + b;`
`xor` *d* ← *a*, *b*	`d = a ^ b;`	`sub` *d* ← *a*, *b*	`d = a - b;`
`orr` *d* ← *a*, *b*	`d = a | b;`	`rsb` *d* ← *a*, *b*	`d = b - a;`
`bis` *d* ← *a*, *b*	`d = a | b;`	`shl` *d*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

- Both *d* and *a* must be registers; *b* can be a register or, in most machines, can also be 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`

1-bit ALU Implementation



- 8 possible functions:
  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 **C**,**N**,**V** flags; now add **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; but 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 equivalent: `d = (a==b) ? 5 : 7;`

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

Condition Codes

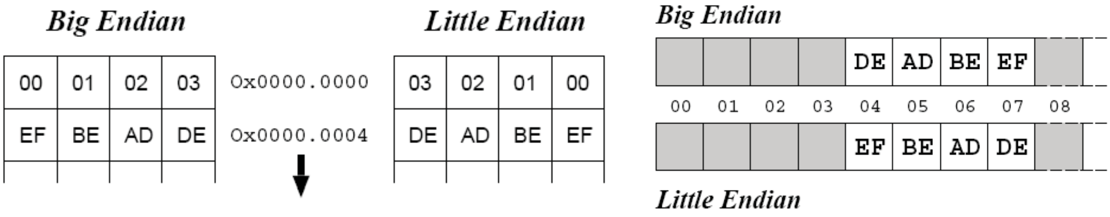
Suffix	Meaning	Flags
EQ, Z	Equal, zero	Z == 1
NE, NZ	Not equal, non-zero	Z == 0
MI	Negative	N == 1
PL	Positive (incl. zero)	N == 0
CS, HS	Carry, higher or same	C == 1
CC, LO	No carry, lower	C == 0
HI	Higher	C == 1 && Z == 0
LS	Lower or same	C == 0 || Z == 1
VS	Overflow	V == 1
VC	No overflow	V == 0
GE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

Used to compare **unsigned** numbers (recall **C**==0 means we borrowed)

Used to compare **signed** numbers (note must check both **N** and **V**)

Loads and 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 0xDEADBEEF into memory at address 0x4



Big Endian

00	01	02	03	
EF	BE	AD	DE	0x0000.0004

Little Endian

03	02	01	00	
DE	AD	BE	EF	0x0000.0004

0x0000.0000

Big Endian

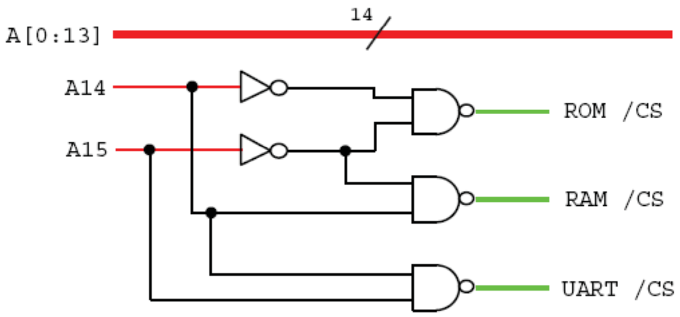
				DE	AD	BE	EF	
00	01	02	03	04	05	06	07	08
				EF	BE	AD	DE	

Little Endian

- 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 devices decode 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]

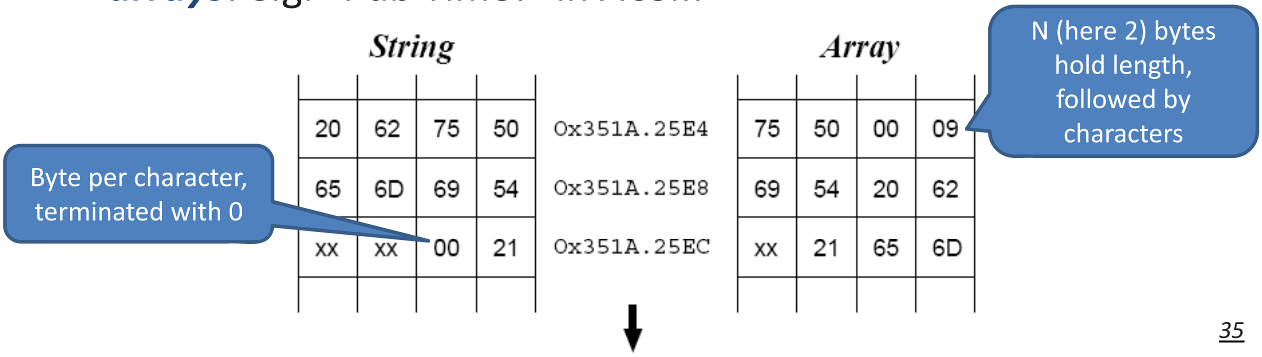
Addressing Modes

- 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 ⇒ 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 – **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 (especially UTF-8!)
- In both cases, represent in memory as either ***strings*** or ***arrays***: e.g. "Pub Time!" in ACSII:



String					*Array*			
20	62	75	50	0x351A.25E4	75	50	00	09
65	6D	69	54	0x351A.25E8	69	54	20	62
xx	xx	00	21	0x351A.25EC	xx	21	65	6D

Byte per character, terminated with 0

N (here 2) bytes hold length, followed by characters

Floating Point

- In many cases need very large or very small numbers
- Use idea of "scientific notation", e.g. $n = \boldsymbol{m} \times 10^{\boldsymbol{e}}$
  - ***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. $n = \boldsymbol{m} \times 2^{\boldsymbol{e}}$, 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, ***s***
  - exponent is stored in a ***biased*** (***excess***) format

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

- e.g. with a 4-bit mantissa and a 3-bit bias-3 exponent, you can represent positive range $[0.001_2 \times 2^{-3}, 1.111_2 \times 2^4]$

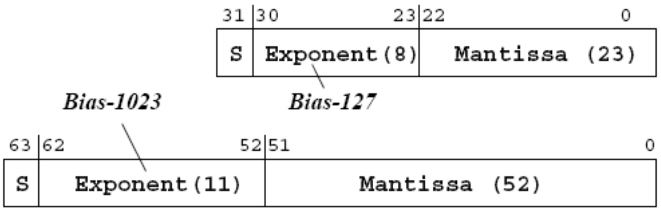
  $= [\ (1/8)(1/8), (15/8)(16)\ ] \ = \ [\ 1/64\ , 30\ ]$

IEEE Floating Point

- To avoid redundancy, in practice modern computers use IEEE floating point with ***normalised*** mantissa ***m*** = $1.xx\ldots x_2$

⇒ $n = (-1)^s((1 + m) \times 2^{e-b})$

- Both single precision (32 bits) and double precision (64 bits)



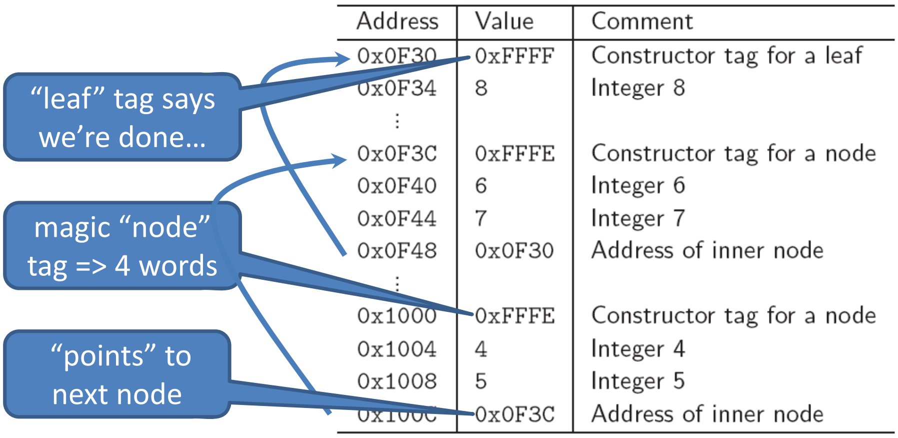
- IEEE fp reserves ***e*** = 0 and ***e*** = max:
  - ±0 (!): both ***e*** and ***m*** zero.
  - ±∞: ***e*** = max, ***m*** zero.
  - NaNs: ***e*** = max, ***m*** non-zero.
  - denorms: ***e*** = 0, ***m*** non-zero
- Normal positive range $[2^{-126}, \sim 2^{128}]$ for single, or $[2^{-1022}, \sim 2^{1024}]$ for double precision.
- **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

"leaf" tag says we're done…

magic "node" tag => 4 words

"points" to next node

Instruction Encoding

- An instruction comprises:
  a. an ***opcode***: specifies what to do.
  b. zero or more ***operands***: where to get values
- Old machines (and x86) use variable length encoding for low code density; most other modern machines use fixed length encoding for simplicity, e.g. **ARM ALU instructions**:

31 28	27 26	25	24 21	20	19 16	15 12	11 0
Cond	00	*I*	*Opcode*	*S*	*Ra*	*Rd*	*Operand 2*

```
and r13, r13, #255
```

1110	00	1	0000	0	1101	1101	000011111111

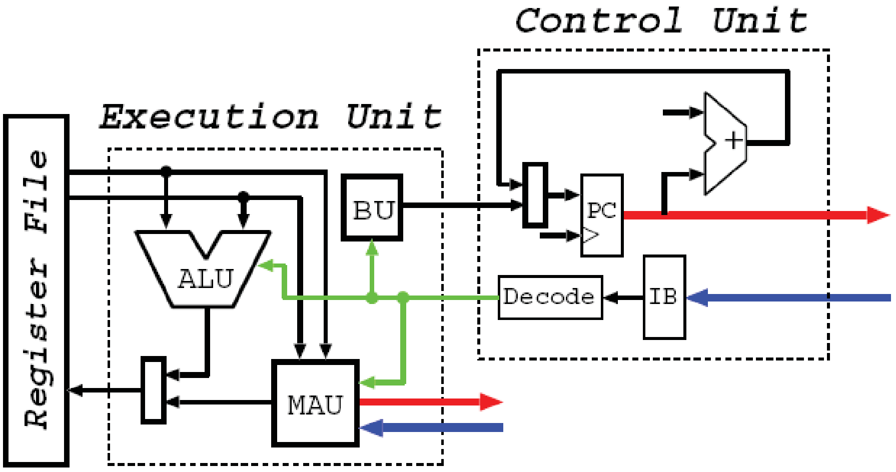
```
bic r03, r03, r02
```

1110	00	0	1110	0	0011	0011	000000000010

```
cmp r01, r02
```

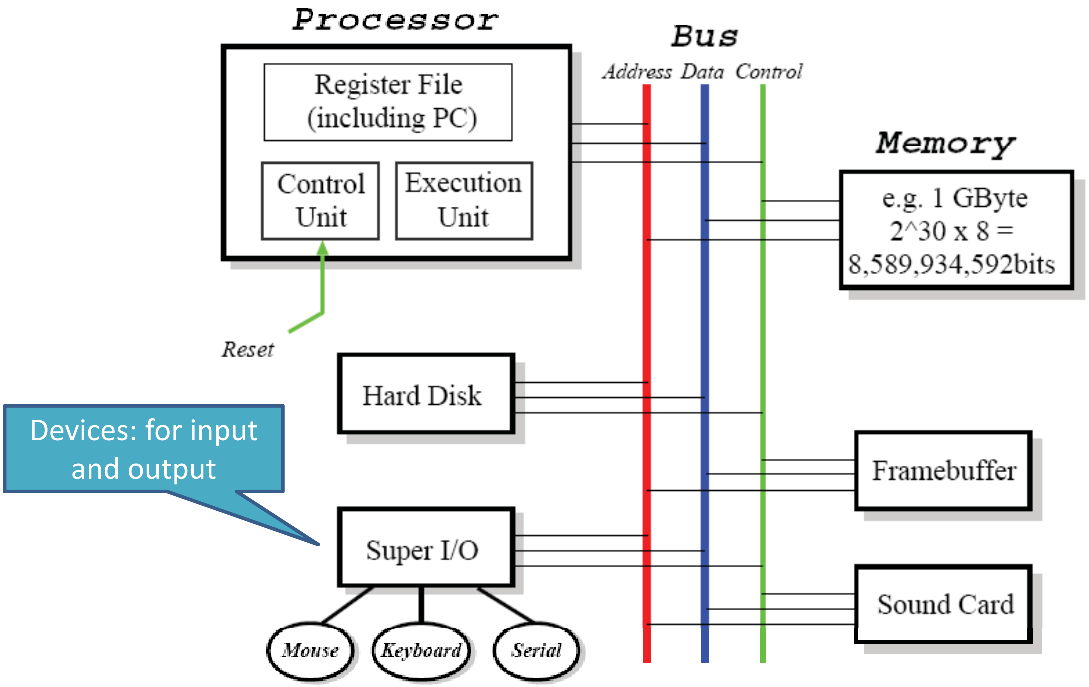
1110	00	0	1010	1	0001	0000	000000000010

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*

A (Simple) Modern Computer

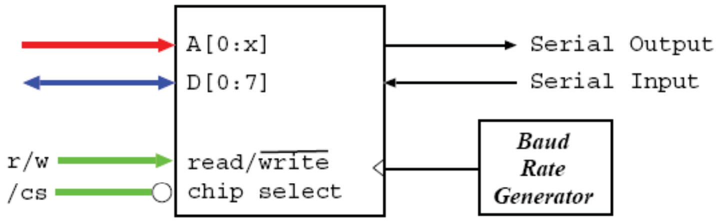


Devices: for input and output

Input/Output Devices

- Devices connected to processor via a bus (e.g. PCI)
- 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. USB, IDE, SCSI, RS-232, Centronics, etc.)

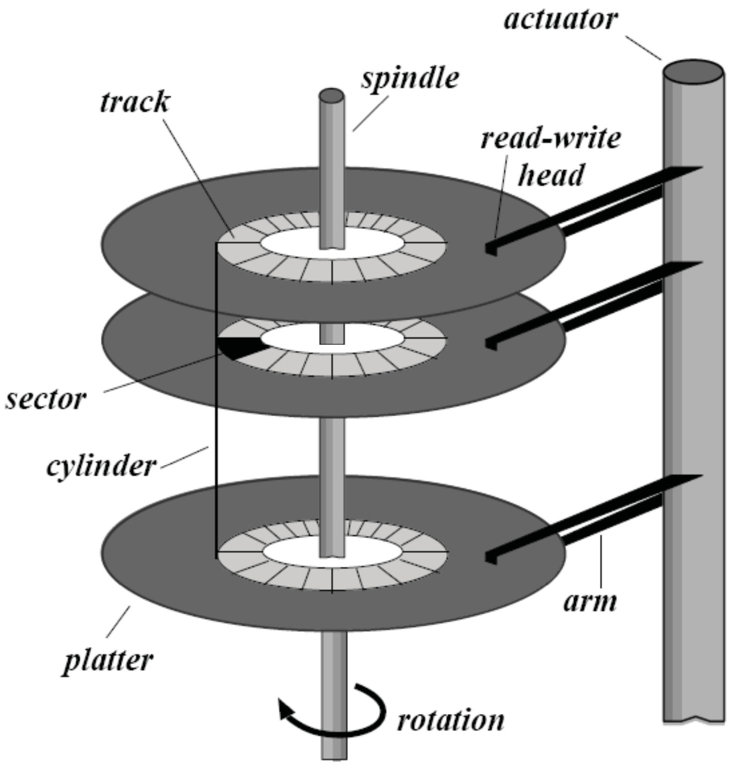
UARTs



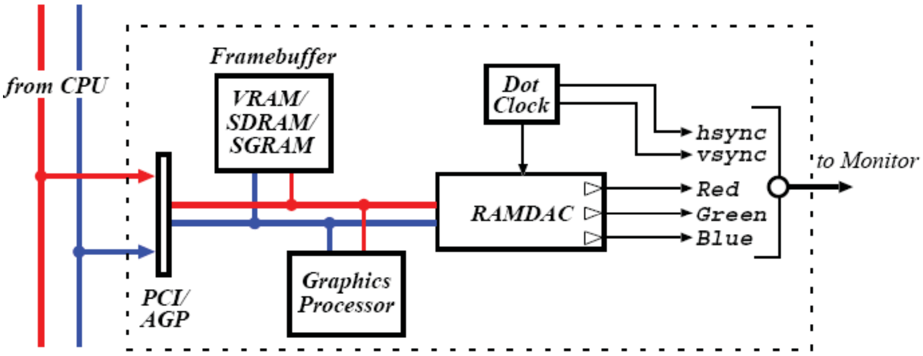
- **UART** = **U**niversal **A**synchronous **R**eceiver/**T**ransmitter:
  - stores 1 or more bytes internally
  - converts parallel to serial
  - outputs according to RS-232
- Various baud rates (e.g. 1,200 – 115,200)
- Slow and simple. . . and very useful.
- Make up “serial ports” on PC
- Max throughput 14.4KBytes; variants up to 56K (for modems).

Hard Disks

- Whirling bits of (magnetized) metal. . .
- Bit like a double-sided record player: but rotates 3,600–12,000 times a minute ;-)
- **To read/write data**:
  - ***move arms to cylinder***
  - ***wait for sector***
  - ***activate head***
- Today capacities are around ~500 GBytes (=500 × 2^{30} bytes)

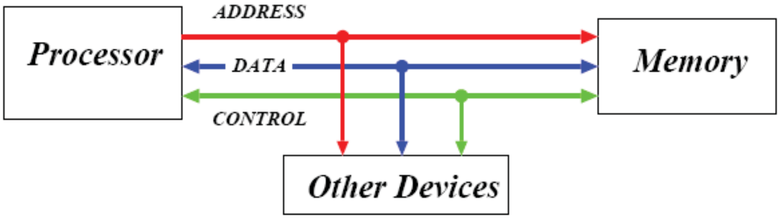


Graphics Cards



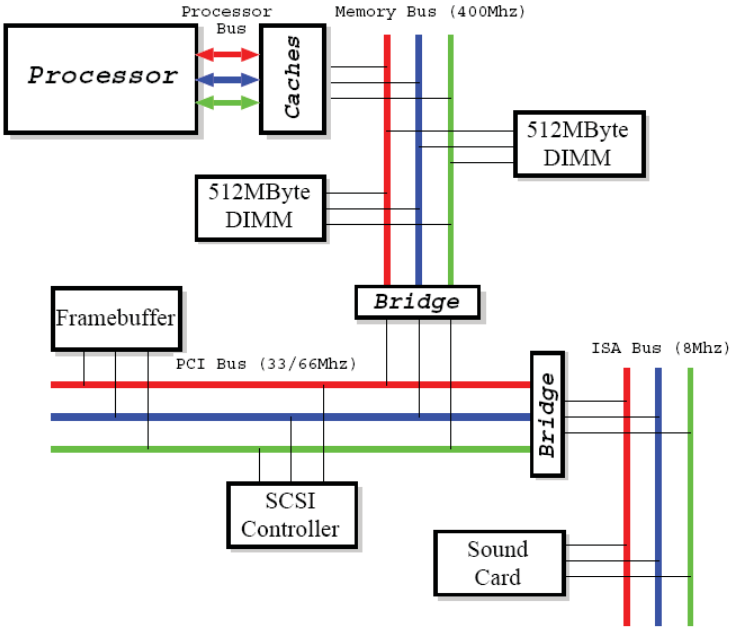
- Essentially some RAM (***framebuffer***) and some digital-to-analogue circuitry (***RAMDAC***) – latter only required for CRTs
- (Today usually also have powerful GPU for 3D)
- Framebuffer holds 2-D array of **pixels**: picture elements.
- Various ***resolutions*** (640x480, 1280x1024, etc) and ***color depths:*** 8-bit (LUT), 16-bit (RGB=555), 24-bit (RGB=888), 32-bit (RGBA=888)
- Memory requirement = x × y × depth
- e.g. 1280x1024 @ 32bpp needs 5,120KB for screen
- => full-screen 50Hz video requires 250 MBytes/s (or 2Gbit/s!)

Buses



- Bus = a collection of ***shared*** communication wires:
  - ✓ low cost
  - ✓ versatile / extensible
  - ✘ potential bottle-neck
- Typically comprises *address lines*, *data lines* and *control lines*
  - and of course power/ground
- Operates in a ***master-slave*** manner, e.g.
  1. master decides to e.g. read some data
  2. master puts address onto bus and asserts ‘read’
  3. slave reads address 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

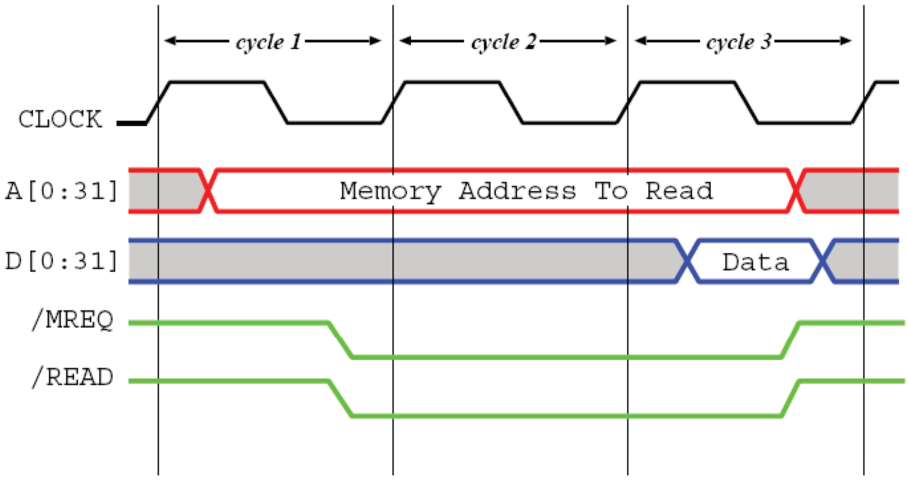
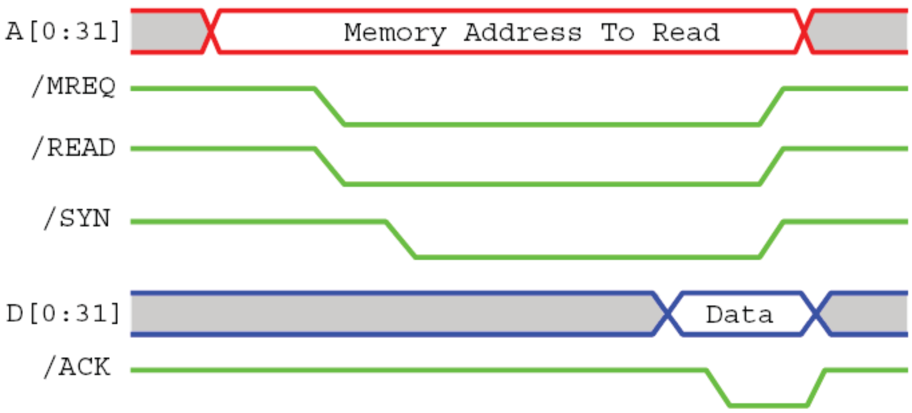


Figure shows a read transaction which requires three bus cycles

1. CPU puts addr onto address lines and, after settle, asserts control lines.
2. Device (e.g. memory) fetches data from address.
3. Device puts data on data lines, CPU latches value and then finally deasserts control lines.

- If device not fast enough, can insert ***wait states***
- Faster clock/longer bus can give ***bus skew***

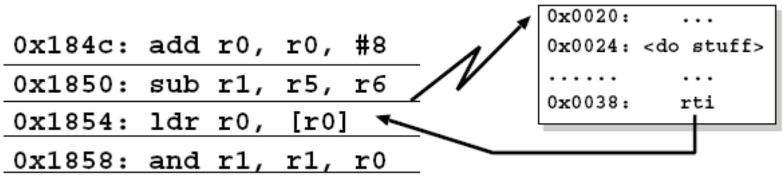
Asynchronous Buses



- Asynchronous buses have no shared clock; instead use ***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 lines

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 ~2ms, which might be over 10,000,000 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 addresses decoded by some 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.

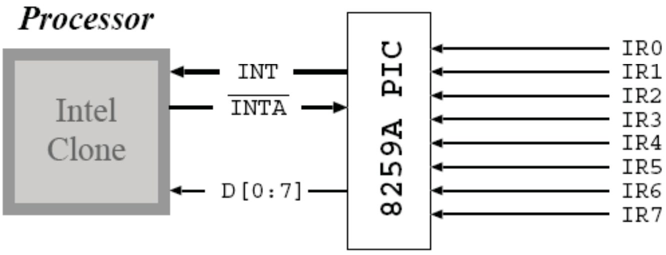


```
0x184c: add r0, r0, #8
0x1850: sub r1, r5, r6
0x1854: ldr r0, [r0]
0x1858: and r1, r1, r0
```

```
0x0020:    ...
0x0024: <do stuff>
......     ...
0x0038:    rti
```

Interrupts (2)

- Interrupt lines (~4–8) are part of the bus.
- Often only 1 or 2 pins on chip ⇒ 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 are 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.

Computer Organization: Summary

- Computers made up of four main parts:
  1. Processor (including register file, control unit and execution unit – with ALU, memory access unit, branch unit, etc),
  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...
- Next up: bare bones programming with MIPS assembly…

What is MIPS?

- A Reduced Instruction Set Computer (RISC) microprocessor:
  - Developed at Stanford in the 1980s [Hennessy]
  - Designed to be fast and simple
  - Originally 32-bit; today also get 64-bit versions
  - Primarily used in embedded systems (e.g. routers, TiVo's, PSPs…)
  - First was R2000 (1985); later R3000, R4000, …
- Also used by big-iron SGI machines (R1x000)

MIPS Instructions

- MIPS has 3 instruction formats:
  - ***R-type*** - register operands
  - ***I-type*** - immediate operands
  - ***J-type*** - jump operands
- All instructions are 1 word long (32 bits)
- Examples of R-type instructions:

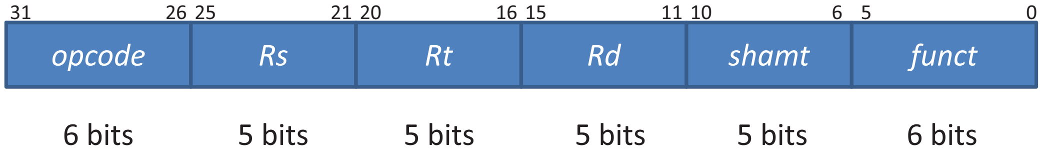
```
add    $8, $1, $2        # $8 <= $1 + $2
sub    $12, $6, $3       # $12 <= $6 - $3
and    $1, $2, $3        # $1 <= $2 & $3
or     $1, $2, $3        # $1 <= $2 | $3
```

- Register 0 ($0) always contains zero

```
add    $8, $0, $0        # $8 <= 0
add    $8, $1, $0        # $8 <= $1
```

R-Type Instructions

- These take three register operands ($0 .. $31)
- R-type instructions have six fixed-width fields:



31 26	25 21	20 16	15 11	10 6	5 0
opcode	*Rs*	*Rt*	*Rd*	*shamt*	*funct*
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

opcode basic operation of the instruction

Rs the first register source operand

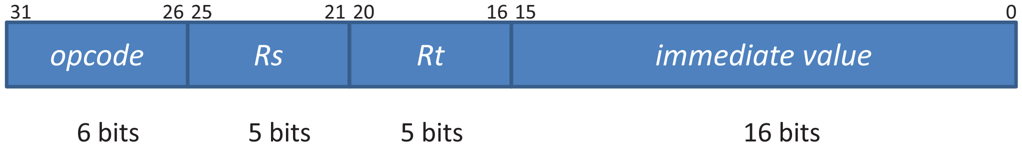
Rt the second register source operand

Rd*:* the register destination operand; gets result of the operation

shamt shift amount (0 if not shift instruction)

funct This field selects the specific variant of the operation and is sometimes called the *function code*; e.g. for opcode 0,
if (funct == 32) => **add** ; if (funct == 34) => **sub**

I-Type Instructions



opcode	Rs	Rt	immediate value
31–26	25–21	20–16	15–0
6 bits	5 bits	5 bits	16 bits

- I = ***Immediate***
  - Value is encoded in instruction & available directly
  - MIPS allows 16-bit values (only 12-bits on ARM)
- Useful for loading constants, e.g:
  - `li  $7, 12  # load constant 12 into reg7`
- This is a big win in practice since >50% of arithmetic instructions involve constants!
- MIPS supports several immediate mode instructions: *opcode* determines which one…

Immediate Addressing on MIPS

- or, and, xor and add instructions have immediate forms which take an “*i*” suffix, e.g:

```
ori    $8, $0, 0x123  # puts 0x00000123 into r8
ori    $9, $0, -6     # puts 0x0000fffa into r9
addi   $10, $0, 0x123 # puts 0x00000123 into r10
addi   $11, $0, -6    # puts 0xfffffffa into r11
                      # (note sign extension...)
```

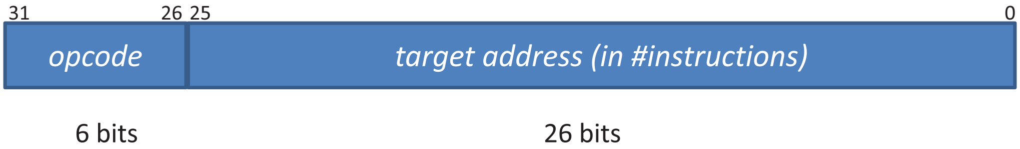
- ***lui*** instruction loads upper 16 bits with a constant and sets the least-significant 16 bits to zero

```
lui    $8, 0xabcd     # puts 0xabcd0000 into r8
ori    $8, $0, 0x123  # sets just low 16 bits
                      # result: r8 = 0xabcd0123
```

- ***li*** *pseudo-instruction* (see later) generates **lui/ori** or **ori** code sequence as needed...

J-Type Instruction

- Last instruction format: ***Jump-type*** (J-Type)


31 26 25 0

opcode | *target address (in #instructions)*

6 bits | 26 bits

- Only used by unconditional jumps, e.g.

```
j dest_addr      # jump to (target<<2)
```

- Cannot directly jump more than 2^{26} instructions away (see later…)
- ***Branches*** use I-type, not J-type, since must specify 2 registers to compare, e.g.

```
beq $1, $2, dest # goto dest iff $1==$2
```

Big Picture

x = a - b + c - d;

High level Language

```
sub    $10, $4, $5
sub    $11, $6, $7
add    $12, $10, $11
```

Assembly Language

0	4	5	10	0	34
0	6	7	11	0	34
0	10	11	12	0	32

```
000000 00100 00101 01010 00000 100010
000000 00110 00111 01011 00000 100010
000000 01010 01011 01100 00000 100000
```

Machine Code

Assumes that a, b, c, d are in $4, $5, $6, $7 somehow

MIPS Register Names

- Registers are used for specific purposes, by ***convention***
- For example, registers 4, 5, 6 and 7 are used as **parameters** or **arguments** for subroutines (see later)
- Can be specified as $4, $5, $6, $7 or as **$a0**, **$a1**, **$a2** and **$a3**
- Other examples:

`$zero`	`$0`	zero
`$at`	`$1`	assembler temporary
`$v0, $v1`	`$2, $3`	expression eval & result
`$t0...$t7`	`$8...$15`	temporary registers
`$s0...$s7`	`$16...$23`	saved temporaries
`$t8, $t9`	`$24, $25`	temporary
`$k0, $k1`	`$26, $27`	kernel temporaries
`$gp`	`$28`	global pointer
`$sp`	`$29`	stack pointer
`$fp`	`$30`	frame pointer
`$ra`	`$31`	return address

Our first program: Hello World!

```
      .text       # begin code section
      .globl main
main: li $v0, 4   # system call for print string
      la $a0, str # load address of string to print
      syscall     # print the string
      li $v0, 10  # system call for exit
      syscall     # exit

      .data       # begin data section
str:  .asciiz “Hello world!\n”
                  # NUL terminated string, as in C
```

- Comments (after “#”) to aid readability
- Assembly language 5-20x line count of high level languages
- (And empirical wisdom is that development time strongly related to number of lines of code...)

Assembler Directives

- On previous slide saw various things that weren't assembly code instructions: ***labels*** and ***directives***
- These are here to assist assembler to do its job ...
- ... but do not necessarily produce results in memory
- Examples:

`main:`	tell assembler where program starts
`str:`	user-friendly[er] way to refer to a memory address
`.text`	tells assembler that following is part of code area
`.data`	following is part of data area
`.ascii str`	insert ASCII string into next few bytes of memory
`.asciiz str`	...as above, but add null byte at end
`.word n1,n2`	reserve space for words and store values n1, n2 etc. in them
`.half n1,n2`	reserve space for halfwords and store values n1, n2 in them
`.byte n1,n2`	reserve space for bytes and store values n1, n2 in them
`.space n`	reserve space for n bytes
`.align m`	align the next datum on 2^m byte boundary, e.g. .align 2 aligns on word boundary

Pseudo Instructions

- Assemblers can also support other things that look like assembly instructions… but aren't!
  - These are called ***pseudo-instructions*** and are there to make life easier for the programmer
  - Can be built from other actual instructions
- Some examples are:

Pseudo Instruction	Translated to
`move $1,$2`	`add $1, $0, $2`
`li $1, 678`	`ori $1, $0, 678`
`la $8, 6($1)`	`addi $8, $1, 6`

`la $8, label`	`lui $1, label[31:16]`
	`ori $8, $1, label[15:0]`

`b label`	`bgez $0, $0, label`

`beq $8, 66, label`	`ori $1, $0, 66`
	`beq $1, $8, label`

Accessing Memory (Loads & Stores)

- Can load bytes, half-words, or words

```
lb $a0,c($s1)  # load byte; $a0 = Mem[$s1+2]
lh $a0,c($s1)  # load half-word [16 bits]
lw $a0,c($s1)  # load word [32 bits]
```

  - gets data from memory and puts into a register
  - c is a [small] constant; can omit if zero
- Same for stores using `sb`, `sh`, and `sw`
- `lw`, sw etc are ***I-type*** instructions:
  - destination register ($a0), source register ($s1), and 16-bit immediate value (constant *c*)
- However assembler also allows `lw`/sw (and `la`) to be pseudo-instructions e.g.

```
lw $a0, addr --->  lui $1, addr[31:16]
                   lw $a0, addr[15:0]($1)
```

Control Flow Instructions

Assembly language has very few control structures…

- ***Branch instructions***: if <cond> then goto <label>

```
beqz $s0, label          # if $s0==0   goto label
bnez $s0, label          # if $s0!=0   goto label
bge  $s0, $s1, label     # if $s0>=$s1 goto label
ble  $s0, $s1, label     # if $s0<=$s1 goto label
blt  $s0, $s1, label     # if $s0<$s1  goto label
beq  $s0, $s1, label     # if $s0==$s1 goto label
bgez $s0, $s1, label     # if $s0>=0   goto label
```

- ***Jump instructions***: (unconditional goto):

```
j label       # goto instruction at “label:”
jr $a0        # goto instruction at Memory[$a0]
```

- We can build while-loops, for-loops, repeat-until loops, and if-then-else constructs from these…

if-then-else

if ($t0==$t1) then /*blockA */ else /* blockB */

```
    beq $t0, $t1, blockA # if equal goto A
    j   blockB           # ... else goto B

blockA:
    ... instructions of blockA ...
    j    exit

blockB:
    ... instructions of blockB ...

exit:
    ... next part of program ...
```

repeat-until

repeat … until $t0 > $t1

```
    ... initialize $t0, e.g. to 0 ...
loop:
    ... instructions of loop ...
    add $t0, $t0, 1      # increment $t0
    ble $t0, $t1, loop   # if <= $t1, loop
```

- Other loop structures (for-loops, while-loops, etc) can be constructed similarly

Jump Instructions

- Recall ***J-Type*** instructions have 6-bit opcode and 26-bit target address
  - in #instructions (words), so effectively 2^{28} bits
- Assembler converts very distant conditional branches into inverse-branch and jump, e.g.

```
    beq $2, $3, very_far_label
    /* next instruction */
```

- … is converted to:

```
    bne $2, $3, L1;    # continue
    j very_far_label;  # branch far
L1:
    /*next instruction */
```

Indirect Jumps

- Sometimes we need to jump (or branch) more than 2^{28} bytes – can use ***indirect jump*** via register

```
  jr $t1              # transfer control to
                      # memory address in $t1
```

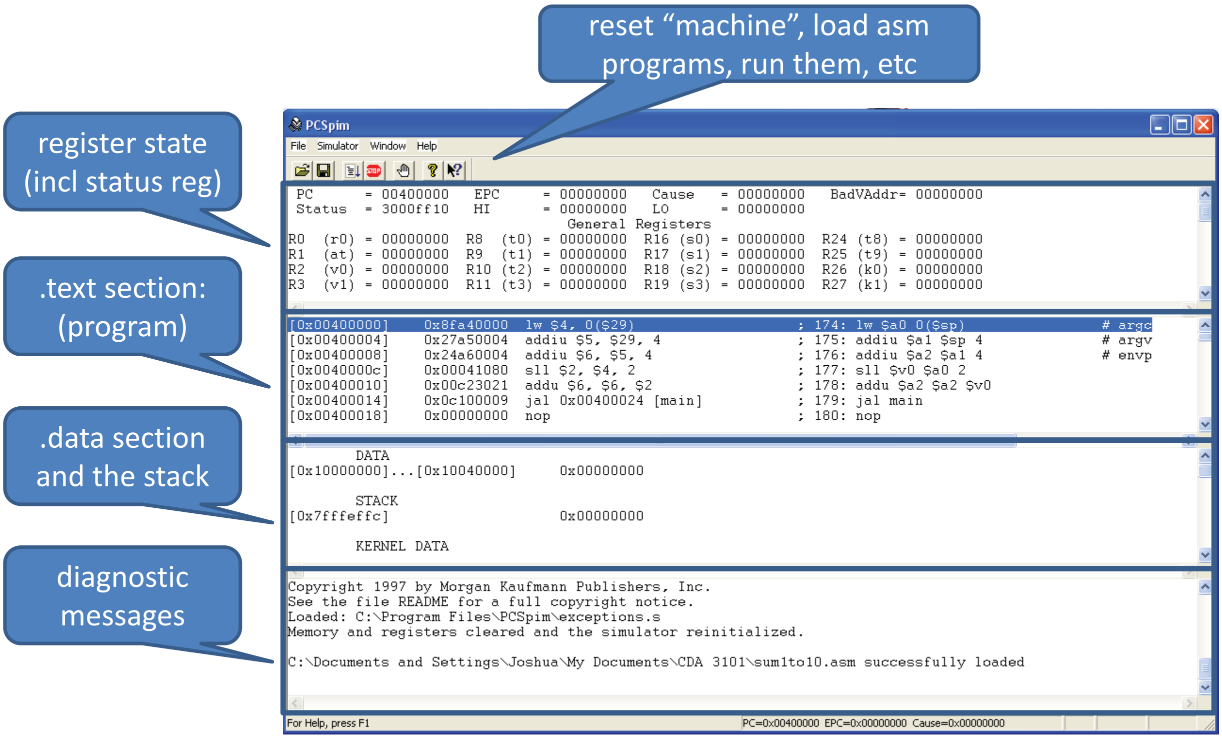
- Can also use to build a ***jump table***
- e.g. suppose we want to branch to different locations depending on the value held in $a0

```
        .data
jtab:   .word l1, l2, l3, l4, l5, l6
        .text
main:   ... instructions setting $a0, etc ...
        lw $t7, jtab($a0)     # load adddress
        jr $t7                # jump
l1:     ... instructions ...
l2:     ... instructions ...
l3:     ... instructions ... (and so on...)
```

The Spim Simulator

- ***"$^{1}/_{25}$th the performance at none of the cost"***
- Simulates a MIPS-based machine with some basic virtual hardware (console)
- Installation
  1. From the Patterson & Hennesey textbook CD
  2. From the internet
     http://www.cs.wisc.edu/~larus/spim.html
- Versions for Windows, Mac and Linux

PC Spim



reset "machine", load asm programs, run them, etc

register state (incl status reg)

.text section: (program)

.data section and the stack

diagnostic messages

```
 PC       = 00400000   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status   = 3000ff10   HI       = 00000000   LO       = 00000000
                                General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
```

```
[0x00400000]    0x8fa40000  lw $4, 0($29)                    ; 174: lw $a0 0($sp)          # argc
[0x00400004]    0x27a50004  addiu $5, $29, 4                 ; 175: addiu $a1 $sp 4        # argv
[0x00400008]    0x24a60004  addiu $6, $5, 4                  ; 176: addiu $a2 $a1 4        # envp
[0x0040000c]    0x00041080  sll $2, $4, 2                    ; 177: sll $v0 $a0 2
[0x00400010]    0x00c23021  addu $6, $6, $2                  ; 178: addu $a2 $a2 $v0
[0x00400014]    0x0c100009  jal 0x00400024 [main]            ; 179: jal main
[0x00400018]    0x00000000  nop                              ; 180: nop
```

```
        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA
```

```
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\exceptions.s
Memory and registers cleared and the simulator reinitialized.

C:\Documents and Settings\Joshua\My Documents\CDA 3101\sum1to10.asm successfully loaded
```

For Help, press F1 PC=0x00400000 EPC=0x00000000 Cause=0x00000000

Using SPIM

- Combines an assembler, a simulator and BIOS
- Assembly language program prepared in your favourite way as a text file
- Label your first instruction as *main*, e.g.

```
main:    add   $5, $3, $4    # comment
```

- Read program into SPIM which will assemble it and may indicate assembly errors (1 at a time!)
- Execute your program (e.g. hit F5)
- Results output to window which simulates console (or by inspection of registers)
- Let's look at an example...

SPIM System Calls

- As you'll have noticed, SPIM allows us to use special code sequences, e.g.

```
li  $a0, 10  # load argument $a0=10
li  $v0, 1   # call code to print integer
syscall      # print $a0
```

  – will print out "10" on the console
- The `syscall` instruction does various things depending on the value of $v0
  – this is very similar to how things work in a modern PC or Mac BIOS, albeit somewhat simpler
- (We'll see why these are called "system calls" later on in the course...)

SPIM System Call Codes

Procedure	code $v0	argument
print int	1	$a0 contains number
print float	2	$f12 contains number
print double	3	$f12 contains number
print string	4	$a0 address of string
read int	5	res returned in $v0
read float	6	res returned in $f0
read double	7	res returned in $f0
read string	8	$a0 buffer, $a1 length
exit program	10	/* none */

Example: Print numbers 1 to 10

```
   .data
newln: .asciiz “\n”
   .text
   .globl main
main:
   li $s0, 1            # $s0 = loop counter
   li $s1, 10           # $s1 = upper bound of loop
loop:
   move $a0, $s0        # print loop counter $s0
   li $v0, 1
   syscall
   li $v0, 4            # syscall for print string
   la $a0, newln        # load address of string
   syscall
   addi $s0, $s0, 1     # increase counter by 1
   ble $s0, $s1, loop   # if ($s0<=$s1) goto loop
   li $v0, 10           # exit
   syscall
```

Example: Increase array elems by 5

```
   .text
   .globl main
main:
   la   $t0, Aaddr         # $t0 = pointer to array A
   lw   $t1, len           # $t1 = length (of array A)
   sll  $t1, $t1, 2        # $t1 = 4*length
   add  $t1, $t1, $t0      # $t1 = address(A)+4*length
loop:
   lw   $t2, 0($t0)        # $t2 = A[i]
   addi $t2, $t2, 5        # $t2 = $t2 + 5
   sw   $t2, 0($t0)        # A[i] = $t2
   addi $t0, $t0, 4        # i = i+1
   bne  $t0, $t1, loop     # if $t0<$t1 goto loop
                           # ... exit here ...
   .data
Aaddr:   .word 0,2,1,4,5 # array with 5 elements
len:     .word 5
```

Procedures

- Long assembly programs get very unwieldy!
- ***Procedures*** or ***subroutines*** (similar to *methods* or *functions*) allow us to structure programs
- Makes use of a new J-type instruction, `jal`:
- `jal addr   # jump-and-link`
  - stores (**current address + 4**) into register `$ra`
  - jumps to address addr
- `jr $ra`
  - we've seen this before – an *indirect* jump
  - after a `jal`, this will return back to the main code

Example Using Procedures

```
   .data
newline:.asciiz “\n”
   .text
print_eol:                    # procedure to print "\n"
       li $v0, 4              # load system call code
       la $a0, newline        # load string to print
       syscall                # perform system call
       jr $ra                 # return
print_int:                    # prints integer in $a0
       li $v0, 1              # load system call code
       syscall                # perform system call
       jr $ra                 # return
main:
       li $s0, 1              # $s0 = loop counter
       li $s1, 10             # $s1 = upper bound
loop:  move $a0, $s0          # print loop counter
       jal print_int          #
       jal print_eol          # print "\n“
       addi $s0, $s0, 1       # increment loop counter
       ble $s0, $s1, loop     # continue unless $s0>$s1
```

Non-leaf Procedures

- Procedures are great, but what if have procedures invoking procedures?

```
procA:  ... instructions to do stuff procA does ...
  li   $a0, 25       # prep to call procB
  jal procB          # $ra = next address
  jr   $ra           # return to caller
procB:  ... instructions to do stuff procB does ...
  jr   $ra           # return to caller

main:
  li   $a0, 10       # prep to call procA
  jal procA          # $ra = next address
  ... rest of program ...
```

$ra

INFINITE LOOP!

The Stack

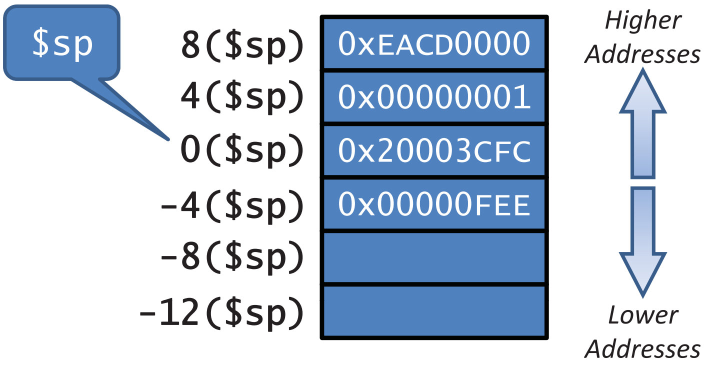
- Problem was that there’s only one $ra!
  - generally need to worry about other regs too
- We can solve this by saving the contents of registers in memory before doing procedure
  - Restore values from memory before return
- ***The stack*** is a way of organizing data in memory which is ideally suited for this purpose
  - Has so-called last-in-first-out (LIFO) semantics
  - ***push*** items onto the stack, ***pop*** items back off
- Think of a pile of paper on a desk
  - “**pushing**” an item is adding a piece of paper
  - “**popping**” is removing it
  - size of pile grows and shrinks over time

The Stack in Practice

- Register $sp holds address of top of stack
  - In SPIM this is initialized to 0x7FFF.EFFC
- A “push” stores data, and decrements $sp
- A “pop” reads back data, and increments $sp

```
# $a0 holds 0xFEE
# ‘push’ $a0
sub $sp, $sp, 4
sw $a0, 0($sp)

# ‘pop’ $a0
lw $a0, 0($sp)
add $sp, $sp, 4
```



8($sp)	0xEACD0000
4($sp)	0x00000001
0($sp)	0x20003CFC
-4($sp)	0x00000FEE
-8($sp)	
-12($sp)	

- We use the stack for parameter passing, storing return addresses, and saving and restoring other registers

Fibonacci… in assembly!

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)

0, 1, 1, 2, 3, 5, 8, 13, 21,…

```
li $a0, 10          # call fib(10)
jal fib             #
move $s0, $v0       # $s0 = fib(10)
```

fib is a recursive procedure with one argument $a0
need to store argument $a0, temporary register $s0 for intermediate results, and return address $ra

Fibonacci: core procedure

```
fib:    sub $sp,$sp,12      # save registers on stack
        sw $a0, 0($sp)      # save $a0 = n
        sw $s0, 4($sp)      # save $s0
        sw $ra, 8($sp)      # save return address $ra
        bgt $a0,1, gen      # if n>1 then goto generic case
        move $v0,$a0        # output = input if n=0 or n=1
        j rreg              # goto restore registers
gen:    sub $a0,$a0,1       # param = n-1
        jal fib             # compute fib(n-1)
        move $s0,$v0        # save fib(n-1)
        sub $a0,$a0,1       # set param to n-2
        jal fib             # and make recursive call
        add $v0, $v0, $s0   # $v0 = fib(n-2)+fib(n-1)
rreg:   lw  $a0, 0($sp)     # restore registers from stack
        lw  $s0, 4($sp)     #
        lw  $ra, 8($sp)     #
        add $sp, $sp, 12    # decrease the stack size
        jr $ra
```

Optional Assembly Ticks

- Tick 0: download SPIM (some version) and assemble + run the hello world program
- Tick 1: write an assembly program which takes an array of 10 values and swaps the values (so e.g. A[0]:= A[9], A[1]:= A[8], … A[9]:= A[0])
- Tick 2 **(*hard*)**: write an optimized version of the Fibonacci code presented here. You may wish do custom stack frame management for the base cases, and investigate tail-recursion.
  - see what Fibonacci number you can compute in 5 minutes with the original and optimized versions

Optional Assembly Ticks

- **Tick 3**: write an assembly program which reads in any 10 values from the keyboard, and prints them out lowest to highest
- There will be a **prize** for the shortest correct answer to Tick 3:
  - make sure you deal with e.g. duplicate values
  - “shortest” means total # of MIPS instructions, so be wary of pseudo-instructions

Email submissions to me by midnight 21st Nov

- use filename ***<CRSID>*-tick3.asm**
- use a text/plain mime attachment