

Computer Hardware & Software

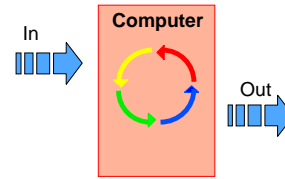
Gorry Fairhurst (c) 1999

Gorry Fairhurst
University of Aberdeen
UK

Objectives:

- To understand how a modern computer operates
- To learn how C and Assembler programs execute
- To provide relevant background for programming

Gorry Fairhurst (c) 1999



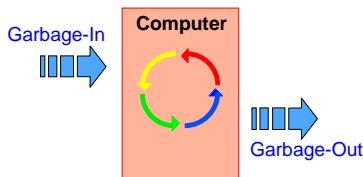
What is a computer?

Definition of "Computer"

Gorry Fairhurst (c) 1999

No fixed definition...

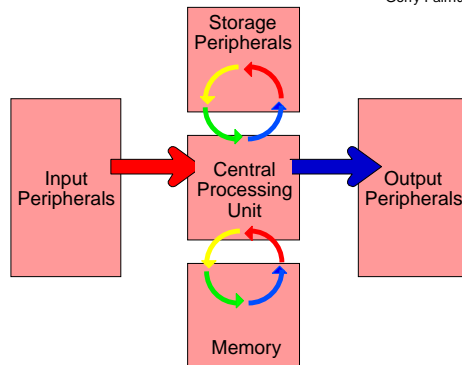
"A computer is a machine which can accept data, process the data and supply the results. The term is used for any computing device that operates according to a stored program."



The computer is only useful with a **valid program** and **correct data**

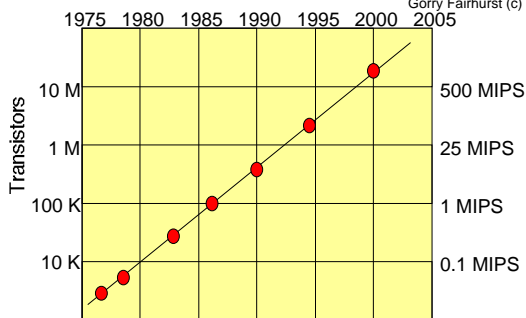
A Computer

Gorry Fairhurst (c) 1999



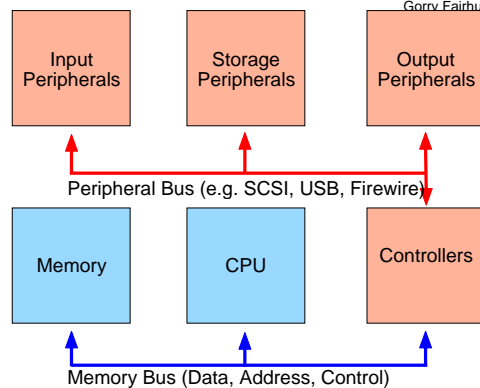
Moore's Law

Gorry Fairhurst (c) 1999



Computer Busses

Gorry Fairhurst (c) 1999



Peripheral Devices

Gorry Fairhurst (c) 1999

Input Devices

Scanner
Optical Character Recognition
Mouse
Keyboard
Microphone
Bar Code Reader
CD-R
DVD-ROM
EPROM
Modem

Storage Devices

Magnetic Tape
Magnetic Disks
Magneto-Optical
Discs
CD-RW
DVD-RAM
Flash Card

Output Devices

Printer
Plotter
Punched Paper Tape
CD-R
DVD-ROM
EPROM
Modem

Address and Values

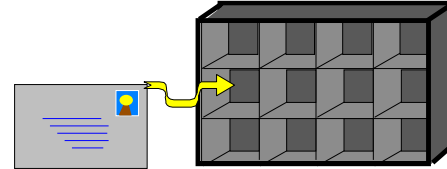
Gorry Fairhurst (c) 1999

Every memory location has a **unique** address

0	0
1	11
2	5
3	23
4	12
5	62

Address of byte

Value of byte (0...255)



Machine Code

Gorry Fairhurst (c) 1999

Computers only understand binary values

0	0	MOV a,#0
1	1	MOV b,#0
2	11	MOV a,b
3	23	a:DATA 23
4	12	b:DATA 12
5	62	c:DATA 62

Human's find this difficult

So we use **Mnemonics** to remember the numbers

Address of byte Value of byte Mnemonic

0 as an instruction could be written: MOV a,#0
Move the value 0 to address a

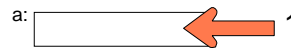
1 as an instruction could be written: MOV b,#0
Move the value 0 to address b

11 as an instruction could be written: MOV a,b
Add the value at address a to address b

Assignment Statement

Gorry Fairhurst (c) 1999

Variable = Constant
e.g. a =1;
a=1; MOV a, #1



Variable = Value of another variable
e.g. a=b;
a=b MOV a, b



The computer performs this by using a "MOV" instruction which copies the value from one place to another

Registers

Gorry Fairhurst (c) 1999

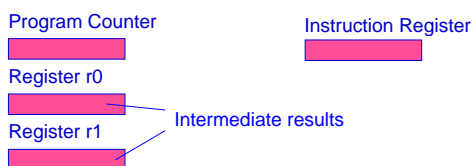
Registers store the intermediate results of programs

Rn Most computers have a set integer registers (R0,R1...)

Some registers have special jobs:

PC Program Counter (points to next instruction)

IR Instruction Register ("Value" of current instruction)



Program Counter

Gorry Fairhurst (c) 1999

PC points to the location of the next instruction to be executed

Program to be executed

0	MOV r0, #1
1	MOV r1, #6
2	ADD r0, r1
3
4
5

Program Counter

0

Register r0

1

Register r1

Instruction Register

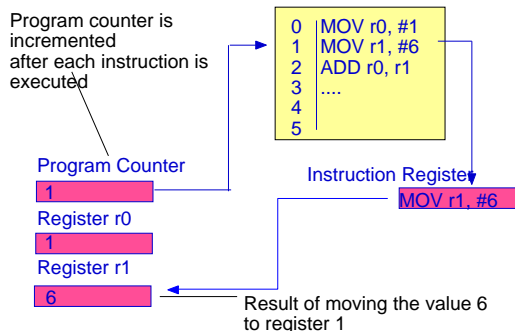
MOV r0, #1

Instruction to be executed

The result of moving 1 to register 0

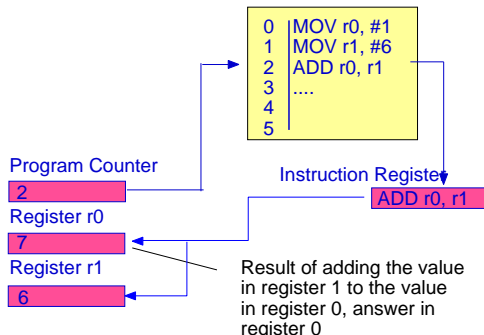
Program Counter

Gorry Fairhurst (c) 1999



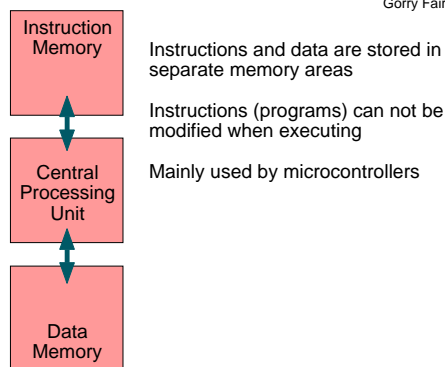
Program Counter

Gorry Fairhurst (c) 1999



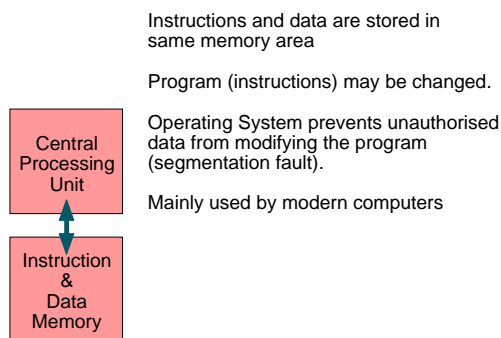
Harvard Architecture

Gorry Fairhurst (c) 1999



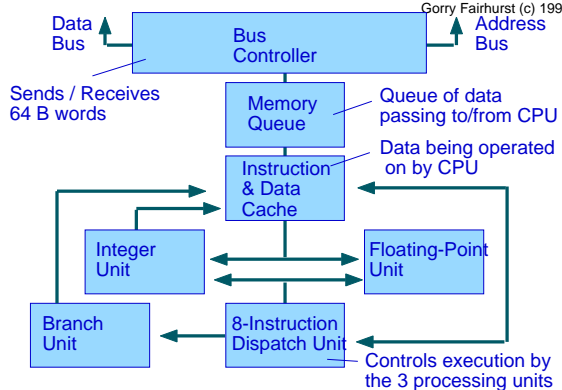
Von Neuman Architecture

Gorry Fairhurst (c) 1999

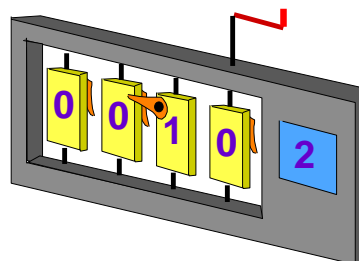


Power PC 601 CPU Architecture

Gorry Fairhurst (c) 1999



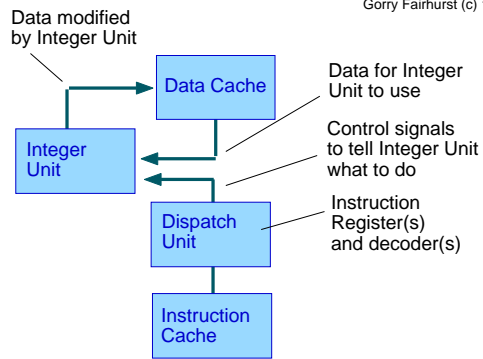
Gorry Fairhurst (c) 1999



Integer Unit

Integer Unit

Gorry Fairhurst (c) 1999



Number Systems

Gorry Fairhurst (c) 1999

Binary

2 values per digit {0,1}
 e.g. $10100 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

Decimal

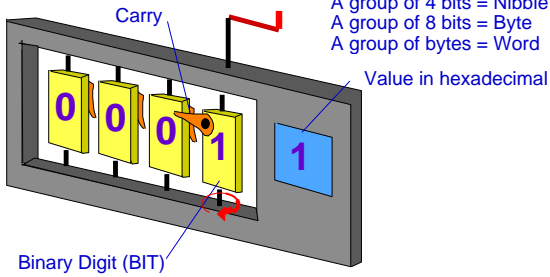
10 values per digit {0,1,2,3,4,5,6,7,8,9}
 e.g. $20 = 2 \times 10^1 + 0 \times 10^0$

Hexadecimal

16 values per digit {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}
 e.g. $14 = 1 \times 16^1 + 4 \times 16^0$

Model of a Register

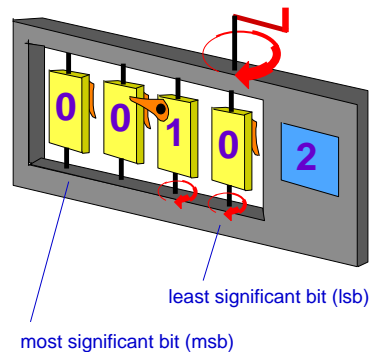
Gorry Fairhurst (c) 1999



N.B. Register values normally described in hexadecimal (base 16)

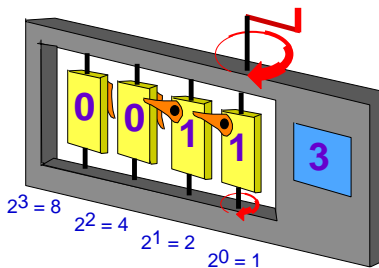
Incrementing the Model Register

Gorry Fairhurst (c) 1999



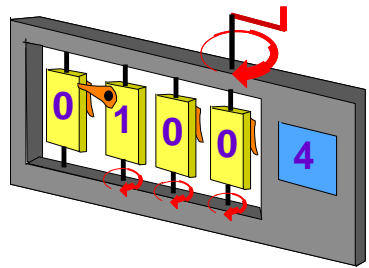
Incrementing the Model Register

Gorry Fairhurst (c) 1999



Incrementing the Model Register

Gorry Fairhurst (c) 1999



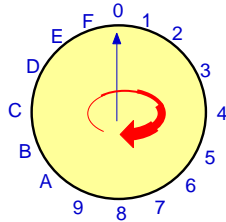
N.B. To add n, turn handle "n" times.

Hexadecimal

Gorry Fairhurst (c) 1999

Dec.	Hex.	Binary
0	0x00	0000 0000
1	0x01	0000 0001
2	0x02	0000 0010
3	0x03	0000 0011
4	0x04	0000 0100
5	0x05	0000 0101
6	0x06	0000 0110
7	0x07	0000 0111
8	0x08	0000 1000
9	0x09	0000 1001
10	0x0A	0000 1010
11	0x0B	0000 1011
12	0x0C	0000 1100
13	0x0D	0000 1101
14	0x0E	0000 1110
15	0x0F	0000 1111
16	0x10	0001 0000
...		

Read hexadecimal numbers in groups of 4 bits (nibbles)



Decimal to Hexadecimal Conversion

Gorry Fairhurst (c) 1999

Decimal to Hexadecimal

Converting 53241 decimal to hexadecimal:

$$\begin{array}{r}
 53241 \div 16 = 3327 \text{ R } 9 \quad (0x9) \text{hsb} \\
 3327 \div 16 = 207 \text{ R } 15 \text{ } 10 \quad (0xF) \\
 207 \div 16 = 12 \text{ R } 15 \text{ } 10 \quad (0xF) \\
 12 \div 16 = 0 \text{ R } 12 \text{ } 10 \quad (0xC) \text{lsb} \\
 53241 = 0x00CFF9
 \end{array}$$

Convention that positive numbers start with 0x0

Hexadecimal to Decimal

Converting 0x00CFF9 to decimal:

$$\begin{aligned}
 &= (9 \times 16^3) + (15 \times 16^2) + (15 \times 16^1) + (12 \times 16^0) \\
 &= 53241
 \end{aligned}$$

$$0x00CFF9 = 53241$$

Binary Addition

Gorry Fairhurst (c) 1999

Adding single digits

$$\begin{array}{r}
 0 \\
 + 0 \\
 \hline
 0
 \end{array}
 \quad
 \begin{array}{r}
 0 \\
 + 1 \\
 \hline
 1
 \end{array}
 \quad
 \begin{array}{r}
 1 \\
 + 0 \\
 \hline
 1
 \end{array}
 \quad
 \begin{array}{r}
 1 \\
 + 1 \\
 \hline
 1 \ 1
 \end{array}$$

Adding binary numbers

$$\begin{array}{r}
 010 \\
 + 101 \\
 \hline
 111
 \end{array}
 \quad
 \begin{array}{r}
 110 \\
 + 101 \\
 \hline
 1011
 \end{array}
 \quad
 \begin{array}{r}
 101 \\
 + 011 \\
 \hline
 1000 \\
 111
 \end{array}$$

Hexadecimal Addition

Gorry Fairhurst (c) 1999

$$\begin{array}{r}
 20 \quad 0x14 \quad 0001 \ 0100 \\
 +5 \quad +0x05 \quad +0000 \ 0101 \\
 \hline
 =25 \quad =0x19 \quad =0001 \ 1001
 \end{array}$$

a	b	c	S	C
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3 bit binary adder

- 0 + 1 = 1
- 0 + 0 = 0
- 1 + 1 = 0, c
- 0 + 0 + c = 1
- 0 + 1 = 1

N.B.
Sum = 1 if there are an odd number of 1's
Carry = 1 if there are two or more 1's

Hexadecimal Signed Numbers

Gorry Fairhurst (c) 1999

1's Complement (bit-wise inversion)

int x
x = ~x
int's are normally 4 r1 (or 8 nibbles)

e.g.

$$20 = 0x00000014$$

msb = 0 for positive number
msb = 1 for negative number.

$$-20 = 0xFFFFFEB$$

0x means the number is in hexadecimal

Note that the size of variable determines how many digits!

Hexadecimal Signed Numbers

Gorry Fairhurst (c) 1999

2's Complement (true negation)

int x
x = (~x)+1

e.g.

$$20 = 0x00000014$$

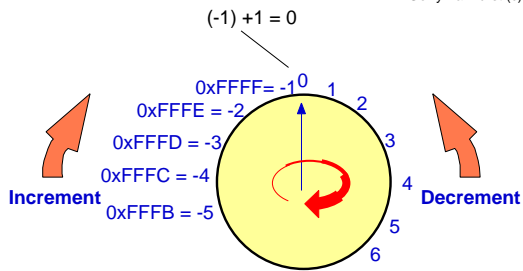
Sufficient to add 1 or 2 zeros before the first non-zero digit.

$$-20 = 0xFFFFFEC$$

More care is needed to get the size correct for negative numbers

Hexadecimal Signed Numbers

Gorry Fairhurst (c) 1999

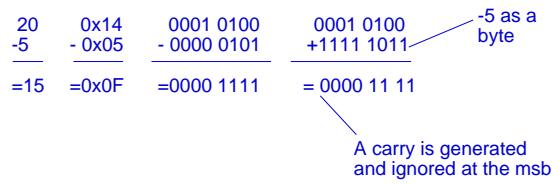


Subtraction

Gorry Fairhurst (c) 1999

Subtraction is **difficult!**

Easier to **negate** a value in 2's complement and then **add**



Signed and Unsigned Numbers

Gorry Fairhurst (c) 1999

1111 1101

The binary value "1111 1101" has the msb set, it may therefore be interpreted as either:

The *unsigned char* 0x00FD (+253)

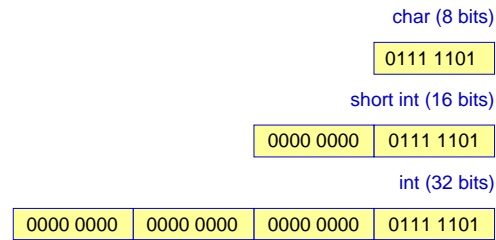
or

The *signed char* 2's complement number 0xFD (-3)

N.B. In C the size of the type "char" is one byte. It is important to know the **type** of the number to determine the value when the msb is set to 1.

Size of Variables

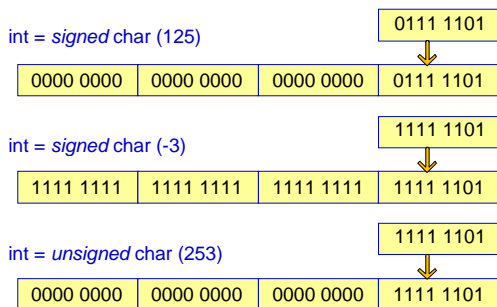
Gorry Fairhurst (c) 1999



N.B. The assembler (or compiler) must determine the size of each variable to use the correct instruction

Type Conversion

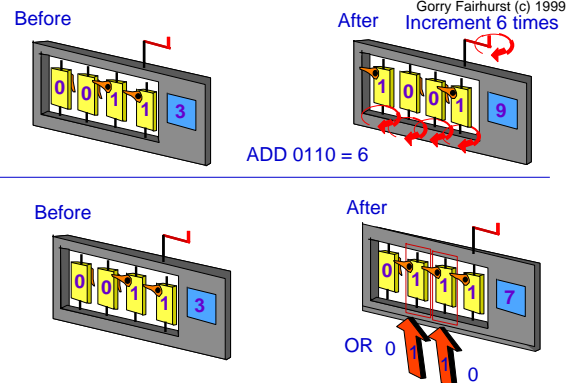
Gorry Fairhurst (c) 1999



N.B. For signed values, the sign must be **extended**

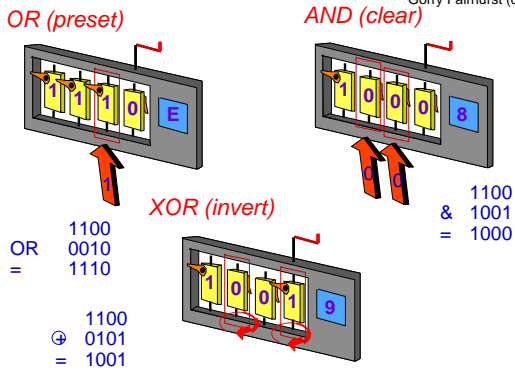
ADD (Increment n) / OR (preset)

Gorry Fairhurst (c) 1999



Bit-Wise Logical Operators in the Model Register

Gorry Fairhurst (c) 1999



Operators

Gorry Fairhurst (c) 1999

Unary operators (operate on one register)

- = assign value (move)
- ~a bit-wise inversion (1's complement)
- a negation (2's complement)
- a++ increment a (add 1 to a)
- a-- decrement a (subtract 1 from a)

Logical operators

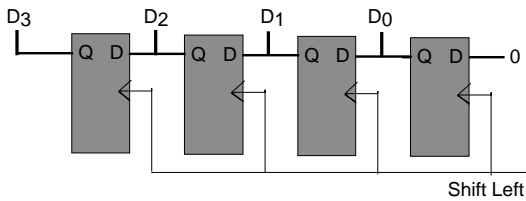
- a & b Bit-wise logical and of a and b
- a ^ b Bit-wise exclusive OR (XOR)
- a | b Bit-wise OR
- a >> n Shift a left n bits (a multiplied by 2ⁿ)
- a << n Shift a right n bits (a divided by 2ⁿ)

Arithmetic Operators

- a + b add a to b
- a - b subtract b from a
- a * b a multiplied by b
- a / b a divided by b

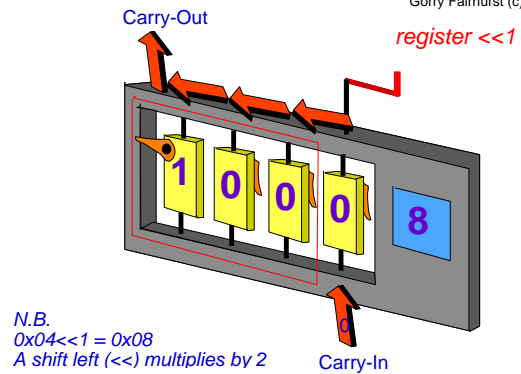
4-Bit Shift Register

Gorry Fairhurst (c) 1999



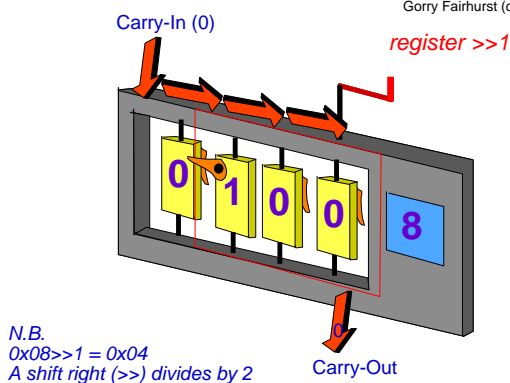
Model Left Shift Register

Gorry Fairhurst (c) 1999

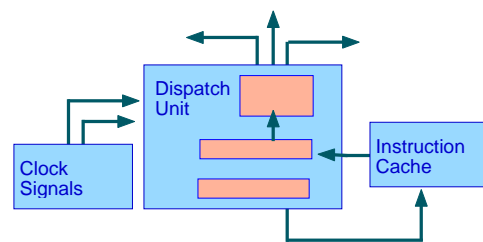


Model Right Shift Register

Gorry Fairhurst (c) 1999



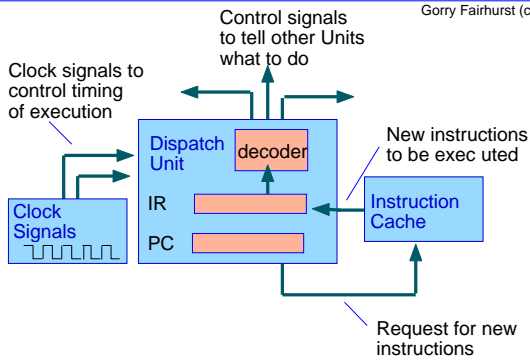
Gorry Fairhurst (c) 1999



Dispatch Unit

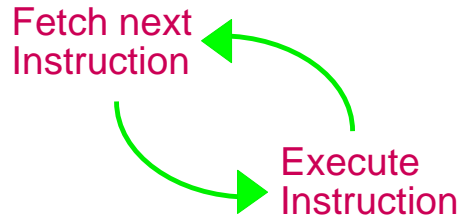
Dispatch Unit

Gorry Fairhurst (c) 1999



Fetch-Execute Cycle

Gorry Fairhurst (c) 1999



Assembler Mnemonics

Gorry Fairhurst (c) 1999

Humans (as a rule) don't understand machine code

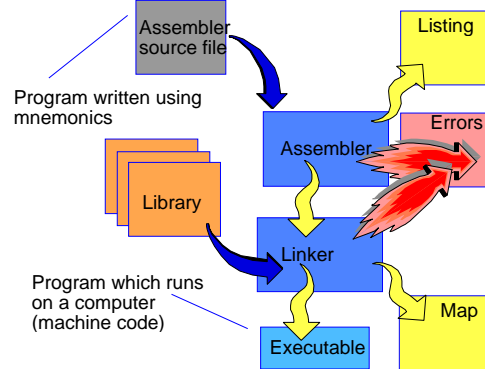
```
// Example assembler program
// to find the average of 4 numbers
// input in three registers: r0-r3
// answer in register r4
```

```
MOV r4, r0
ADD r4, r1
ADD r4, r2
ADD r4, r3
SHIFTR r4,2
```

names of registers
values (starting with #)
or memory locations (no #)
Source register / location / value
Destination register / location
Mnemonic for an instruction

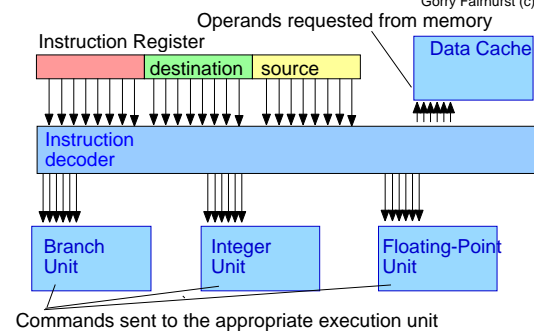
Assemblers

Gorry Fairhurst (c) 1999



Dispatch Unit

Gorry Fairhurst (c) 1999



Gorry Fairhurst (c) 1999

MOV rn, rm	ADD rn, rm	SUB rn, rm
MOV rn, #p	ADD rn, #p	SUB rn, #p
MOV rn,m(p)	ADD rn,m(p)	SUB rn,m(p)
MOV rn,a	ADD rn,a	SUB rn,a
MOV a,rm		

Instruction Sets

How Big is an Instruction?

Gorry Fairhurst (c) 1999

One byte has 2^8 different values

Two bytes have 2^{16} different values

Four bytes have 2^{32} different values

Six bytes have 2^{48} different values

Most CPUs have 20-60 different basic instructions


Simple instructions fit in one byte
(the extra bits in the byte indicate the operand)

Some instructions require many bytes
(usually to contain addresses or data values)

Simple Instructions

Gorry Fairhurst (c) 1999

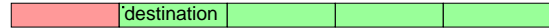
Instructions with no operands

 e.g. HALT, RTS

Instructions with a register operand

 e.g. PUSH r0, POP r1, INC r3

Instructions with one memory operand



e.g. INC 0x07FFFE10, DEC 0x0000001

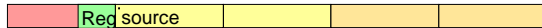
More Instructions

Gorry Fairhurst (c) 1999

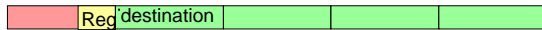
Instructions with two operands



e.g. MOV r0, r1 // copy value in register r1 to register r0
ADD r0, r2 // add value in r2 to r0, answer in r0
SUB r1, r2 // subtract value in r2 from r1, answer in r1



e.g. MOV.W r0, r3 // copy value at location r0 to register r0
ADD.W r1, r5 // add value at location r0 to value in r0



e.g. MOV.W r1, r0 // copy value of register r0 to location r0

Indirect Operands

Gorry Fairhurst (c) 1999

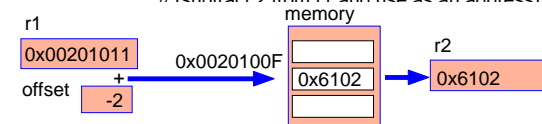
Sometimes program writers do not know an address
and it is calculated by the computer

Indirection

r1 // the value of r1
(r1) // the value of the location pointed to by r1
// (use r1 as an address)

Indirect plus Offset

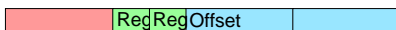
e.g. MOV r2, -2(r1) // the value of the location at minus 2 off r1
// (subtract 2 from r1 and use as an address)



Indexed Offset Operands

Gorry Fairhurst (c) 1999

Instructions with an Indexed Offset Operand



e.g. MOV.W r0, -2(r0) // Subtract 2 from r0 to give an address
// move value from the address to r0

MOV.W 8(sf), r2 // add 8 to sf to give an address
// move value of r2 to the address

These instructions are used extensively by modern compilers

Execution takes place in two phases:

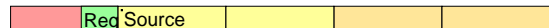
Calculation of address

Execution using the calculated address

Immediate Operands

Gorry Fairhurst (c) 1999

Instructions with immediate data



e.g. MOV.W r0, #4 // Move the constant 4 to register r0
ADD.W r1, #0x0fff // add the constant 0x0fff to value in r1

The # (hash) indicates an actual value rather than a location.

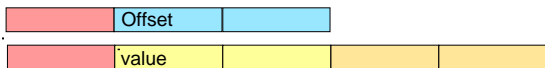
This allows a program to load a constant value into a register

Immediate instructions take the value from the *instruction cache*
rather than from the *data cache* or a *register*.

Instructions that modify PC

Gorry Fairhurst (c) 1999

Instructions with an Indexed Offset Operand



e.g. `JMP label` // move the value of label to PC
 // causes program to "jump" to label

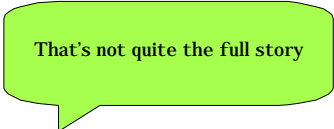
`BRA +6` // Add 6 to PC
 // causes a jump forward in the program

`BRA.NE +6` // Add 6 to PC if "Zero" bit in SR =1
 // jumps if last operation resulted in zero.

These instructions are used to call procedures

Other Instructions

Gorry Fairhurst (c) 1999



Most CPUs actually allow some 64-bit operations.

Most CPUs also implement other more complex instructions (usually combinations of the previous instructions)

You do not need to worry about these to understand how the computer operates.

High-level languages like "C" hide the details of machine code from the users (which is very good news!)

RISC v CISC

Gorry Fairhurst (c) 1999

Some CPUs have a small number of different instructions
 But execute each instruction very fast
 These are called **Reduced Instruction Set Computers**
 e.g. Pentium, 68000

Some CPUs have many different instructions
 They are optimised for common sets of actions
 These are called **Complex Instruction Set Computers**
 e.g. PowerPC, Sparc

At the moment RISC designs are faster

Appropriate use of caching makes a big difference

Increasing the Program Counter

Gorry Fairhurst (c) 1999

The **Program Counter (PC)** points to the next instruction

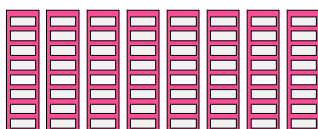
PC is updated after each instruction has been fetched

It needs to be increased by the **size** of the last instruction

Some instructions (e.g. `JMP`, `JSR`, `BRA` ...) modify PC

WARNING
 In most examples in the course we will assume the PC increments (adds 1) after each instruction
 This makes our life easier, but in real life instructions have a range of sizes!!!

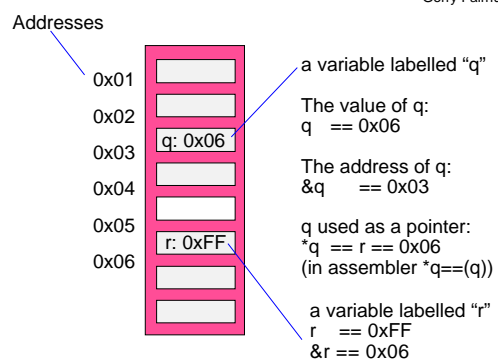
Gorry Fairhurst (c) 1999



Caches & Memory

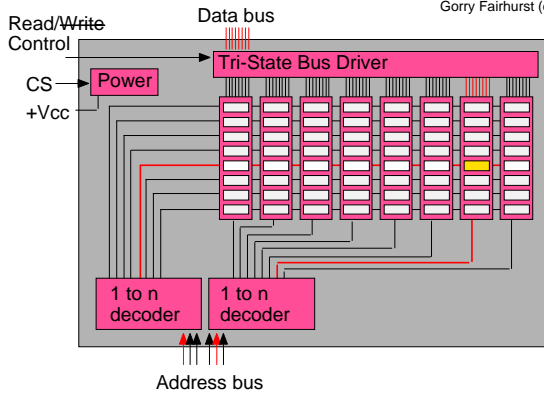
Addresses and Memory

Gorry Fairhurst (c) 1999



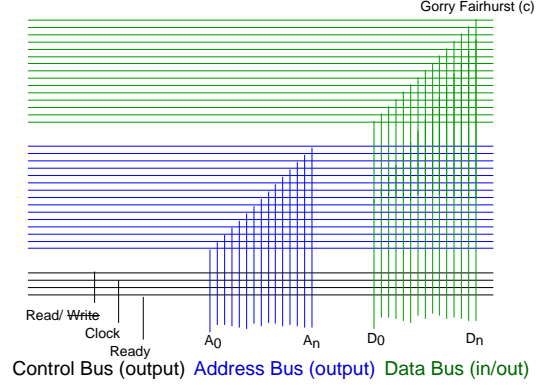
Random Access Memory (RAM)

Gorry Fairhurst (c) 1999



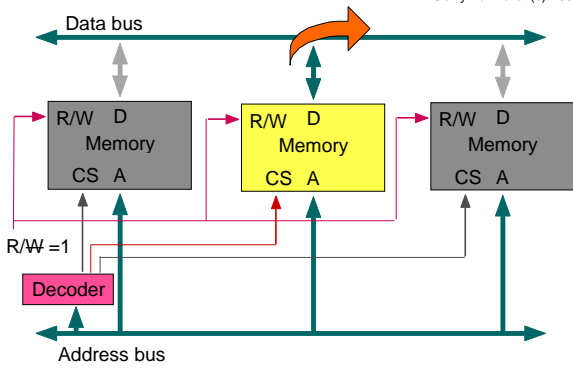
Address, Data, & Control Bus

Gorry Fairhurst (c) 1999



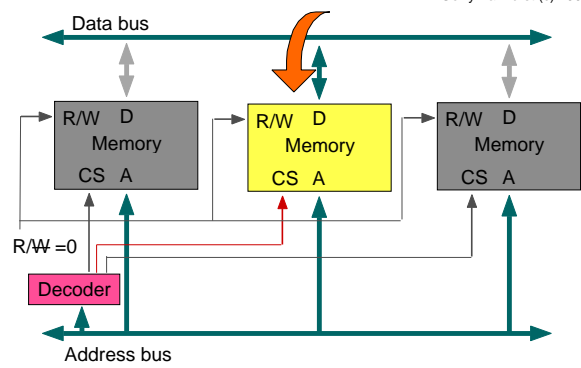
Reading a Location in Memory

Gorry Fairhurst (c) 1999



Writing a Location in Memory

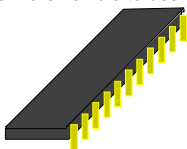
Gorry Fairhurst (c) 1999



Read Only Memory (ROM)

Gorry Fairhurst (c) 1999

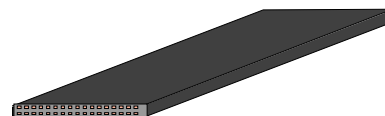
- Program/Data is set at manufacture
- May be mass-produced very cheaply
- Can never be changed (except by replacing ROM)
- Used primarily for storing parts programs that never change
e.g. parts of operating system kernel
- For programs it is more flexible to use EPROM, FLASH



Flash

Gorry Fairhurst (c) 1999

- Program/Data is written by CPU
- May be upgraded very easily
- Used primarily for storing programs and configuration data
- Very expensive compared to ROM, EPROM
- Much slower (particularly to write) than RAM



EPROM

Gorry Fairhurst (c) 1999

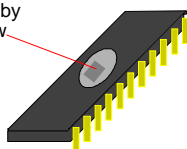
Program/Data is written by an EPROM programmer

Whole chip needs to be erased
(needs to be taken out of computer)

Used primarily for storing programs

More expensive than ROM, but reusable

EPROM erased by exposing window to Ultra-Violet Light



Memory

Gorry Fairhurst (c) 1999

Volatile memory (loses data when no power)		Non-volatile memory (keeps data when no power)		
Dynamic RAM (cheap)	Static RAM (expensive)	ROM (cheap)	EPROM (cheap)	FLASH (cheap)
fast	very fast	fast	fast	slow
main memory	cache & I/O buffer	programs (one use)	programs (reusable)	programs and data

CPUs are faster than Memory

Gorry Fairhurst (c) 1999

CPUs operate **much** faster than memory does!



Accessing memory is a severe **bottleneck**

Accessing Memory

Gorry Fairhurst (c) 1999

Three fortunate observations:

Programs may be optimised
Using registers instead of memory to reduce **data** transfer

Programs often execute loops of instructions
The same **instructions** are often used many times

Programs usually read and write consecutive locations
Data are often stored in words, or larger groups of bytes

Caches

Gorry Fairhurst (c) 1999

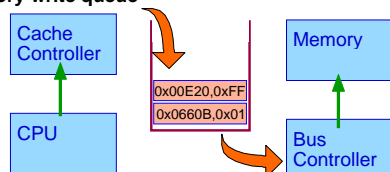
Caches can do three things to improve performance:

Recently **read** data kept in fast memory for quick re-use

They **read** locations from memory before they are required

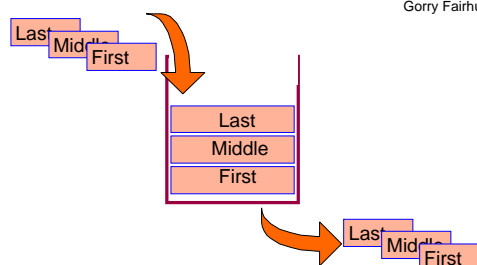
They defer **writing** data to memory
Allowing program to continue while memory catches up

The memory write queue



Queues

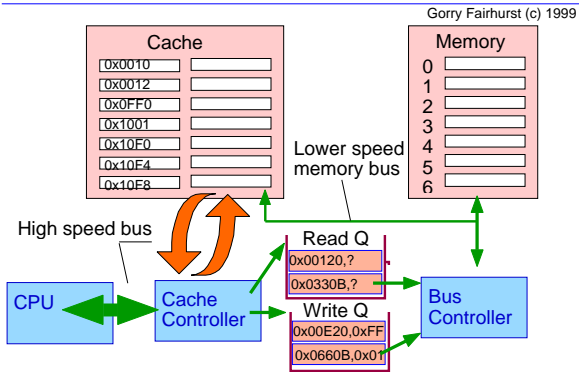
Gorry Fairhurst (c) 1999



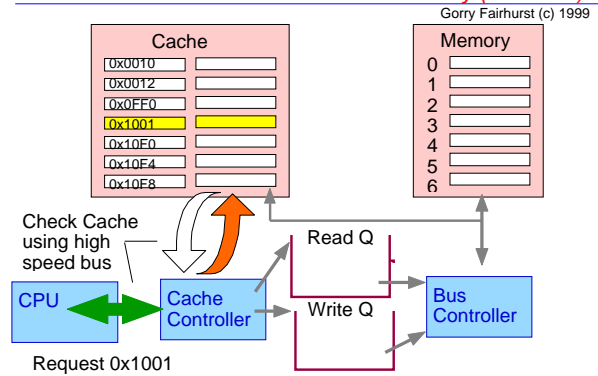
FIFO = First-In First-Out

Queues have only two ends

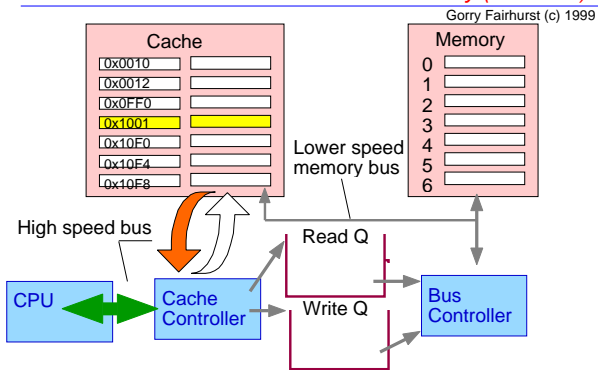
Cache



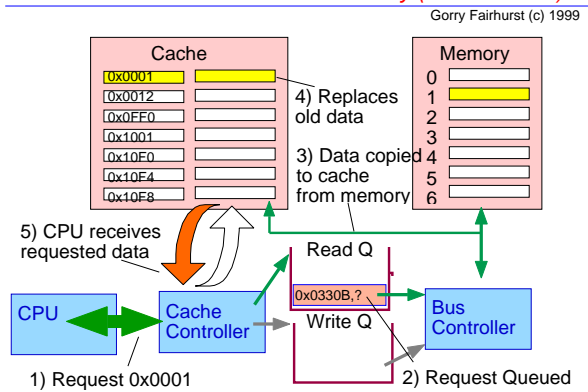
Read from Memory (in Cache)



Read from Memory (in Cache)



Read from Memory (Not in Cache)



Gorry Fairhurst (c) 1999

Why C?

Gorry Fairhurst (c) 1999

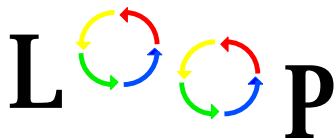
Why do we need assemblers?

- 1) Assemblers use mnemonics to provide "labels" which humans can understand.
- 2) Assemblers let us use all the features of a CPU

Why do we need high level languages?

- 1) Need to "learn" a new language for each CPU
- 2) Programmers like the same program to run using different CPUs.
- 3) Many standard operations are required many times e.g. printing, data arrays, random numbers, trigonometry, databases, reading and writing files,....

A high level language (e.g. C) provides these features



Loops

Average

Gorry Fairhurst (c) 1999

```

{
  int a, sum, num_size, average;
  int num[10];
  num[1]=1; num[2]=3; num[4]=9;
  num[5]=9; num[6]=10; num[7]=10;
  num[8]=11; num[9]=12; num[10]=2;

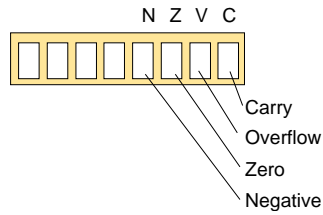
  num_size=10;
  a=1;
  sum = 0;
more: sum = sum + num[a];
  a++;
  if (a <= num_size) {goto more;}
  average=sum/num_size;
}
    
```

Instruction with a label "more"

Branch if a < 10 back to instruction with the label "more"

Status Register

Gorry Fairhurst (c) 1999



Each Execution Unit has a Status Register
Register contains a set of bits (flags)
These are changed after execution of certain instructions

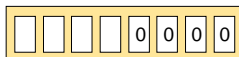
e.g. ADD, ADDC, SHIFTR, SHIFTL, INC, DEC

Status Register

Gorry Fairhurst (c) 1999

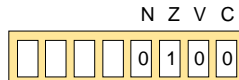
```

ADD 0,1
= 0x0001
    
```



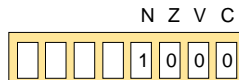
```

ADD 1,-1
= 0x0000
    
```



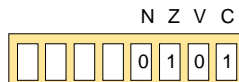
```

ADD -1,-1
= 0xFFFE
    
```



```

SHIFTR, 1,1
= 0x0000
    
```



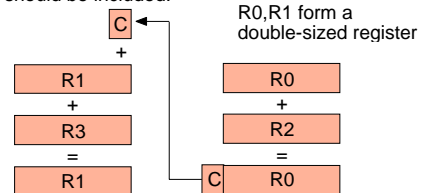
Long Addition

Gorry Fairhurst (c) 1999

```

ADD R0,R2 // add the least significant part
           // the "carry" in the status reg may be set
ADDC R1,R3 // add the most significant part
           // together with the carry
    
```

We use a special arithmetic instruction to say we are using long addition and a carry should be included.



Conditional Branches

Gorry Fairhurst (c) 1999

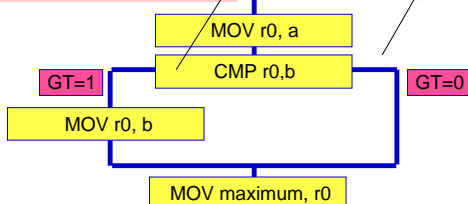
```

{
  int a,b, maximum;
  a=1;
  b=10;

  maximum = a;
  if (maximum < b) {maximum=b;}
}
    
```

Sets Greater Than (GT) bit of status register

If Greater Than (GT) =0 the program counter value is increased by an offset



Loops & Branches

Gorry Fairhurst (c) 1999

Part of a program to do something 20 times:

```

main()
{
  int count;
  count =20;
loop: count--;
  /* SOMETHING */
  if (count != 0) {goto loop;}
}
    
```

Initialise count to 20 (i.e. move 20 to register containing count)

Decrement count (i.e. decrement register containing count and set Z if Zero)

If count is not zero, goto label loop (i.e. if Z is not set to 1, then branch to loop)

Conditional Branches

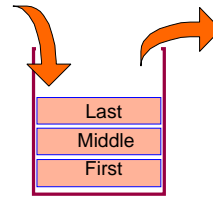
Gorry Fairhurst (c) 1999

Address	Instruction	PC When GT=1	PC When GT=0
0x01 62	MOV r0, a	0x01 64	0x01 64
0x01 64	CMP r0,b	0x01 66	0x01 66
0x01 66	BRA.LE +2	0x01 68	0x01 6A
0x01 68	MOV r0, b	0x01 6A	----
0x01 6A	MOV maximum, r0	0x01 6B	0x016B

Next PC value depends on result in status register

N.B. Less-Than-or-Equal is the opposite of GT (N=1 or Z=1)

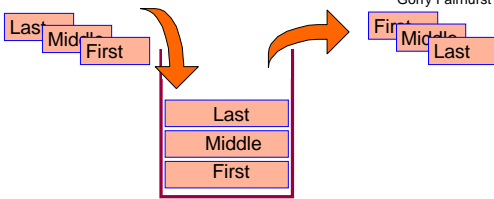
Gorry Fairhurst (c) 1999



Stacks

Stacks

Gorry Fairhurst (c) 1999



LIFO = Last-In First-Out

Stacks have only one end

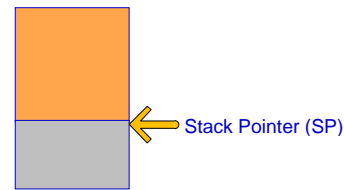
There are two operations:

Push (to put something on the stack)

Pop (to take something off the stack)

Computer Stacks

Gorry Fairhurst (c) 1999



Only one register is needed to keep a stack

In the computers stack grow **downwards**

More Registers

Gorry Fairhurst (c) 1999

Registers store the intermediate results of programs

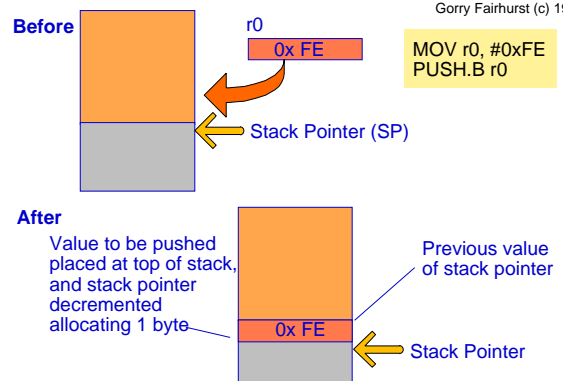
- Rn Most computers have a set integer registers (R0,R1...)
- Fn Most computers also have floating point registers

Some registers have special jobs:

- PC Program Counter (points to next instruction)
- IR Instruction Register ("Value" of current instruction)
- SR Status Register (Status flags after executing instructions)
- SP Stack Pointer (points to next free location on the stack)
- SF Stack Frame (used to locate program variables)

Push Byte

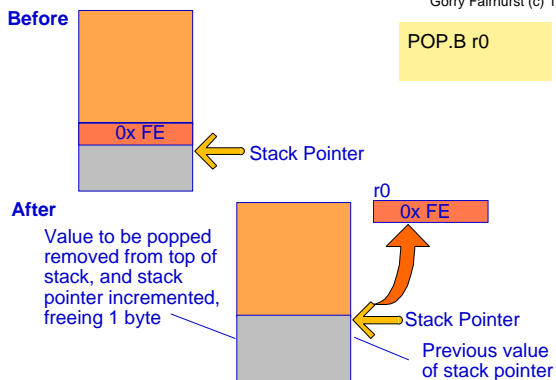
Gorry Fairhurst (c) 1999



MOV r0, #0xFE
PUSH.B r0

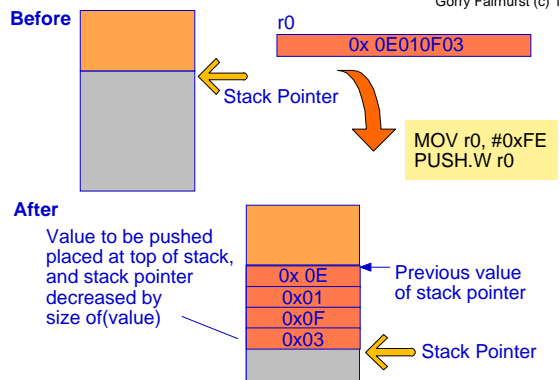
Pop Byte

Gorry Fairhurst (c) 1999



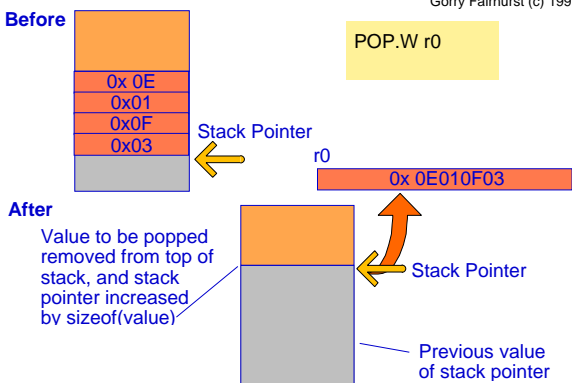
Push Word

Gorry Fairhurst (c) 1999



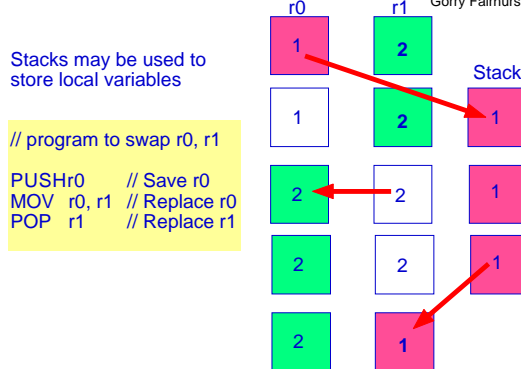
Pop Word

Gorry Fairhurst (c) 1999



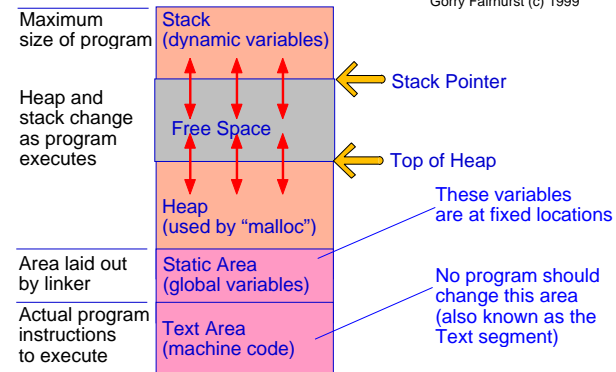
Example using Stack to Swap 2 Values

Gorry Fairhurst (c) 1999

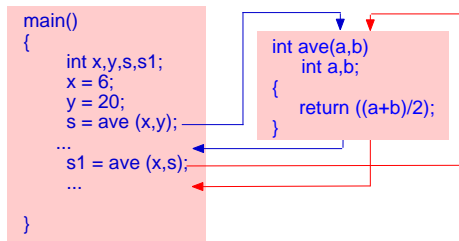


Memory Allocation (Segments)

Gorry Fairhurst (c) 1999



Gorry Fairhurst (c) 1999



Procedures

Why Use Procedures?

Gorry Fairhurst (c) 1999

Procedures (also known as "functions" and "subroutines")

- Shorten programs
- Reduce use of global variables (fewer mistakes)
- Allow testing of parts of programs (more reliable)
- Allow for software re-use (saves time)

Three types of Procedure

Gorry Fairhurst (c) 1999

Simple Procedures without parameters

(always do the same thing, may return one result)

Procedures which read parameters

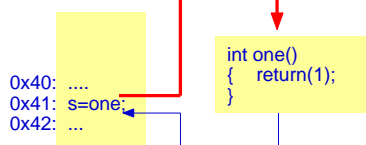
Call by value
(may return one result)

Procedures which modify parameter values

Call by pointer
(may change more than one variable)

Jump To Subroutine (JSR)

Gorry Fairhurst (c) 1999



A **JSR** instruction is used to call a subroutine / procedure

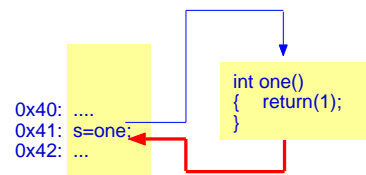
This does three operations:

- Value of PC (incremented) is saved at location SP
- The value of SP is decreased by the size of PC
- The operand (address of procedure) is loaded into PC

N.B. The normal "increment" of PC is not performed.

Return from Subroutine (RTS)

Gorry Fairhurst (c) 1999



The last statement of a procedure is "**return**"

This uses the "**RTS**" instruction

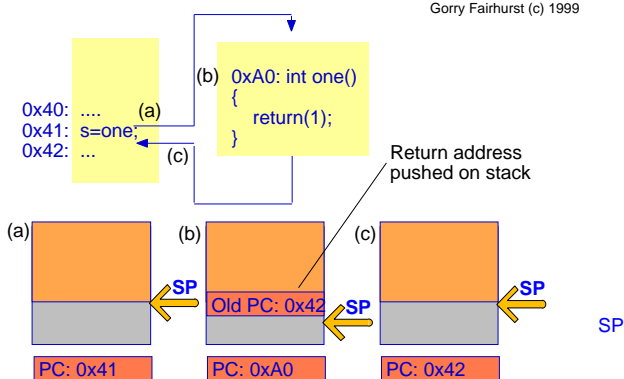
This does two operations:

- The value pointed to by SP is restored to PC
- The value of SP is increased by the size of SP

N.B. The PC is not incremented

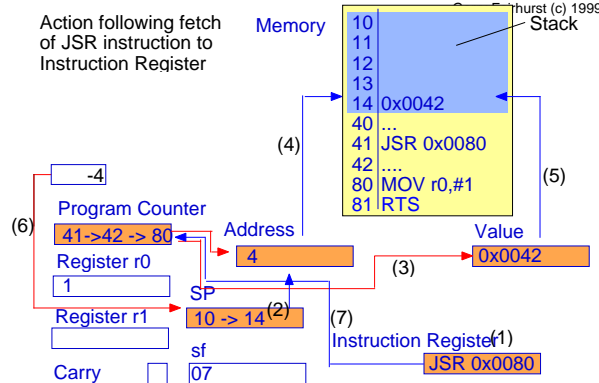
Simple Procedure Call

Gorry Fairhurst (c) 1999

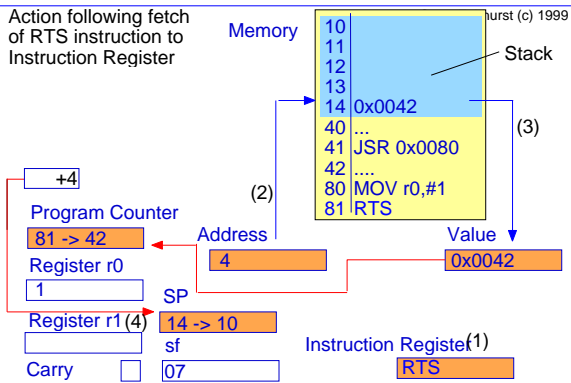


Jump to Subroutine

Gorry Fairhurst (c) 1999



Return from Subroutine



Procedures with Parameters

Gorry Fairhurst (c) 1999

Call by value

The value of each parameter is copied to the STACK
 The procedure does not know the actual variable location
 It **can not** change the variable's value
 The copy is destroyed when the procedure returns

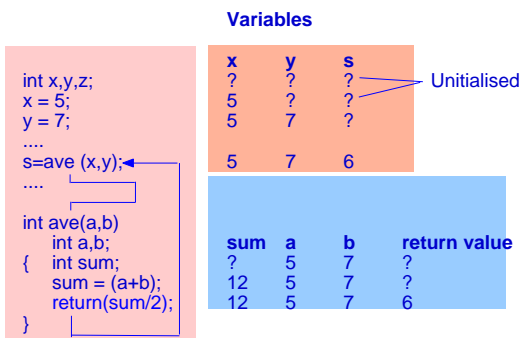
Call by pointer

A pointer to the location is placed on the STACK by the caller
 The procedure **can** change the variable value
 The pointer is destroyed when the procedure returns

N.B. It is usually safer (and quicker) to use **call by value**.

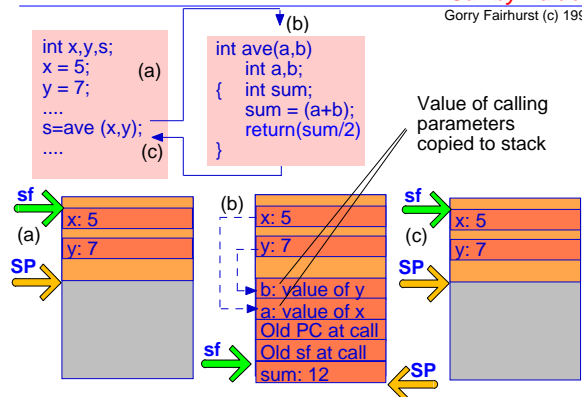
Procedure Calls by Value

Gorry Fairhurst (c) 1999



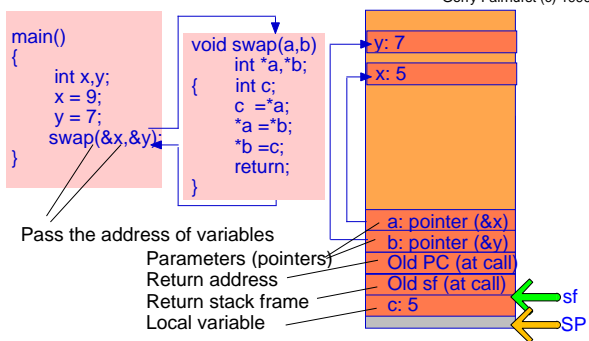
Call by Value

Gorry Fairhurst (c) 1999



Subroutine Call by Pointer

Gorry Fairhurst (c) 1999



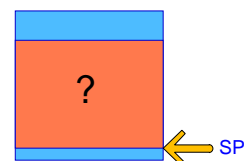
Allocating Space for Local Variables

Gorry Fairhurst (c) 1999

The computer doesn't know *when* the stack is used

So it doesn't know the value of the stack pointer (SP)
 (This changes anyway as the program executes)

How does it know the location of variables on the stack?

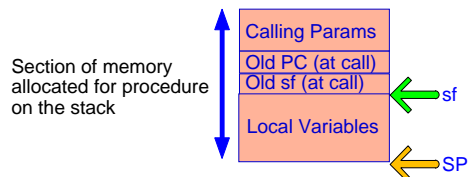


It uses a "stack frame" (sf) pointer

The Stack Frame

Gorry Fairhurst (c) 1999

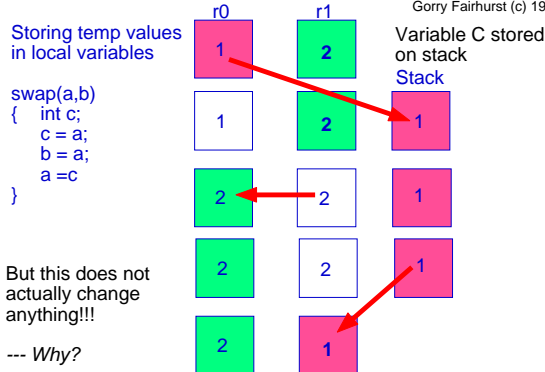
Each subroutine makes it's own little world of variables



- (i) Each procedure saves the old value of the sf when called.
 - (ii) The procedure sets a new sf to point to the local variables
 - (iii) Space is then reserved on the stack for local variables
 - (iv) All local variables are referenced relative to sf
 - (v) When the procedure returns it restores the old sf
- (In practice some compilers use slight variants of this)

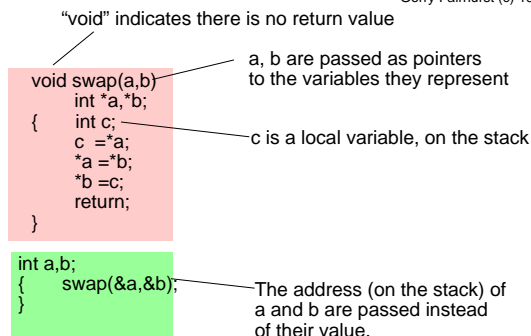
Procedure swap(a,b)

Gorry Fairhurst (c) 1999



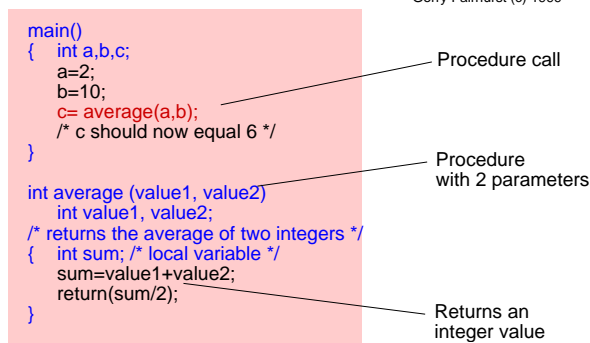
A Useful Swap Procedure

Gorry Fairhurst (c) 1999



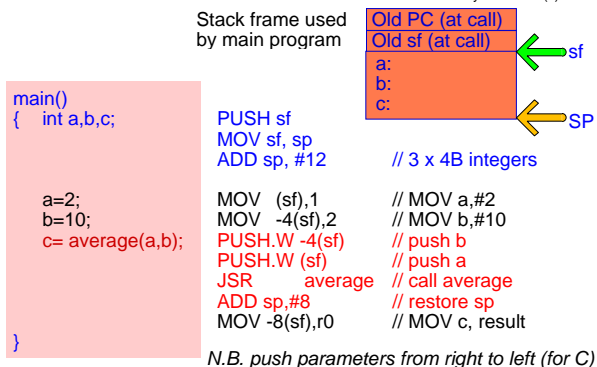
An Example C Program to Average 2 Numbers

Gorry Fairhurst (c) 1999



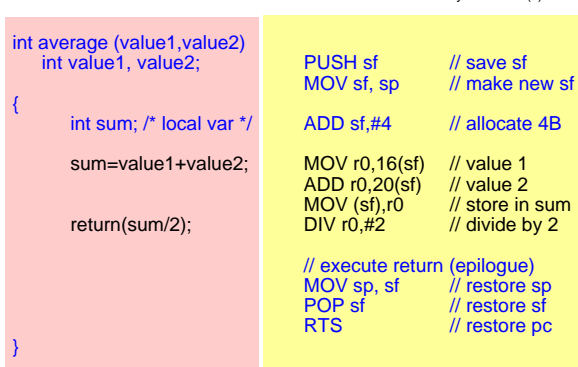
Program calling Procedure Average

Gorry Fairhurst (c) 1999



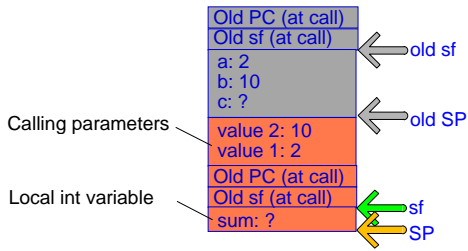
Procedure Average

Gorry Fairhurst (c) 1999



Stack Frame Set Up by Procedure Average

Gorry Fairhurst (c) 1999



Beware of Uninitialised Variables

Gorry Fairhurst (c) 1999

Note a stack frame only allocates space, it does **not** initialise variables!

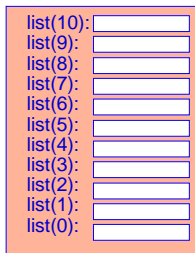
If you don't assign a value you get the last value on the stack!

This is like **Russian Roulette**
It may be OK first time,
But, sooner or later, the program will die!

```
int danger;
{
  int a;
  return(a)
}
```



Gorry Fairhurst (c) 1999

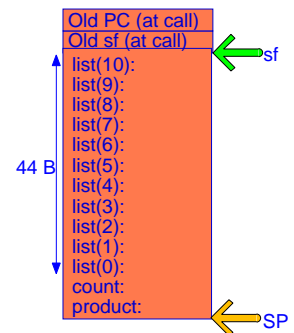


Arrays

Array Elements

Gorry Fairhurst (c) 1999

```
{
  /* An array of 11 numbers */
  /* with n factorial in list(n) */
  int list[11];
  int count;
  int product;
  list[0] = 1;
  count = 1;
  product=1;
loop:
  product=count*product;
  list[count] = product;
  count++;
  if (count <= 10) {goto loop;}
}
```



Array Uses

Gorry Fairhurst (c) 1999

You can create arrays of any type of data (bytes, characters, integers, floating point numbers...)

Arrays may be used for:

- Storing lists of data to be processed
- Storing a series of results
- Performing matrix operations
- Storing strings of characters
- Accessing a lookup-table (to save recalculating complex values)

Sorting an Array of 4 Numbers

Gorry Fairhurst (c) 1999

```
main()
{
  int count;          /* a counter */
  int sorted;        /* a flag */
  int item[4];
  item[0]=1; item[1]=2; item[2]=0; item[3]=3;
loop: /* start of a pass through list of items */
  count = 3; /* sort 4 items in a list */
  sorted = 1; /* assume sorted */
  printf("\nStarting a new pass:\n");
next: if(item[count] < item[count-1]){
      swap(item+count,item+(count-1));
      sorted=0; /* was not sorted */
    }
  printf("item[%d] == %d\n",count,item[count]);
  count--;
  if(count > 0){ goto next;}
  printf("item[%d] == %d\n",count,item[count]);
  if(sorted != 1) { goto loop;}
}
```

Variables

count	0
sorted	0
item(3)	3
item(2)	0
item(1)	2
item(0)	1

Sorting an Array of 4 Numbers

Gorry Fairhurst (c) 1999

item(3)	3	3	3	3	3	3	3	3	3
item(2)	0	2	2	2	2	2	2	2	2
item(1)	2	0	0	1	1	1	1	1	1
item(0)	1	1	1	0	0	0	0	0	0
count	3	2	1	3	2	1	3	2	1
sorted	1	0	0	0	0	0	1	1	1
	Pass 1			Pass 2			Pass 3		

ASCII Characters

Gorry Fairhurst (c) 1999

Printed letters, numbers and symbols are known as characters
Characters are represented by bytes encoded in ASCII

American Standard Code for Information Interchange

A one byte variable

```
e.g.
{
  char letter;
  letter='f';
}
```

This means the ASCII value of f (0x66)

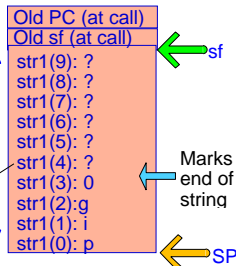
1 byte for each character	
0x00-0x1F	Control characters e.g. 0x0D = end of line \n e.g. 0x00 = end of string
0x20-0x2F	Punctuation
0x30-0x3F	Numbers
0x40-0x5D	Upper case letters
0x60-0x7F	Lower case letters

Character Arrays

Gorry Fairhurst (c) 1999

```
{
  /* create an array of characters */
  char str1[10]; /* up to 9 chars */
  str1[0]='p';
  str1[1]='i';
  str1[2]='g';
  str1[3]=0;
  printf("The animal is a %s\n",str1);
}
```

The animal is a pig



Strings of characters are stored in arrays and terminated by a 0x00

Which Animal is This?

Gorry Fairhurst (c) 1999

```
animal1()
{
  /* create an array of characters */
  char str1[9]; /* up to 8 chars */
  str1[0]='p';
  str1[1]='i';
  str1[2]='g';
  str1[3]=0;
  str1[0]=0x44;
  str1[1]=96+15;
  printf("The animal is a %s\n",str1);
}
```

Use an ASCII chart to lookup these characters

printf is a library routine which asks the operating system to print something

coder.c

Gorry Fairhurst (c) 1999

```
main()
{
  int a;
  char str[30]="SAY HELLO TO FATHER CHRISTMAS\0";
  char outs[30];
  a=0;
  printf("Original string %s\n", str);
loop1:
  outs[a]=str[a]+12;
  if (outs[a]>'Z') {outs[a]=outs[a]-26;}
  a++;
  if (str[a] != 0) {goto loop1;}
  outs[a]=0;
  printf("Encodes to %s \n",outs);
}
```

All strings should end in a NUL

First character is str[0]
This encrypts by adding 12 to the ASCII value

All strings should end in a NUL

Original string SAY HELLO TO FATHER CHRISTMAS

Encodes to EMK,TQXXA,FA,RFMTQD,OTDUEFYME

Beware of Rogue Array Indices

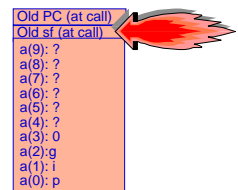
Gorry Fairhurst (c) 1999

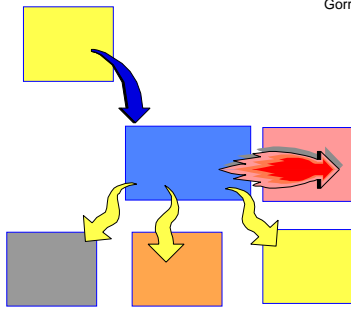
C and Assembler do **not** check the bounds of arrays!

If you use an illegal index you get a bogus value!

It is even more dangerous to assign a bogus value
(This often leads to a "segmentation fault"
i.e. attempt to write to the text area/ segment)

```
{
  int danger;
  int a[10];
  a[10]= danger;
}
```





Compilers

Compiler

- Compiler reads program and checks format (syntax)
- Compiler then identifies each variable and allocates space (N.B. Most variables may need more than one byte)
- It remembers where it puts each variable on the stack (This is called the "symbol table")
- It translates program into object code (or assembly language)
- A listing shows how the program compiled (the list file)
- A map file shows the symbol table
- A linker adds other object code to complete the program

Compiling an Assignment Statement

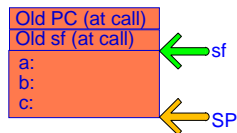
Variable = Constant (e.g. a = 1;)

Variable = Value of another variable (e.g. b=a;)

The computer performs this by using a "move" instruction to move the value from one memory location to another

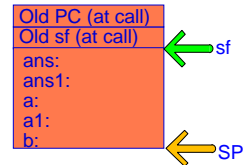
The compiler has first to figure out where each of the variables are located.

```
b=1; /* MOV -4(sf), #1 */
c=a; /* MOV -8(sf), (sf) */
```



Optimising Groups of Statements

```
a=a+2; /* MOV r1, -8(sf) */
      /* ADD r1,#2 */
b=b+a; /* MOV -8(sf),r1 */
      /* ADD r1,-20(sf) */
a=b;   /* MOV -20(sf),r1 */
      /* MOV r1,-20(sf) */
      /* MOV -8(sf),r1 */
```



N.B. most compilers are smart enough to realise this may be **optimised** by effectively using registers:

```
a=a+2; /* MOV r1, -8(sf) */
      /* ADD r1,#2 */
b=b+a; /* MOV -8(sf),r1 */
      /* ADD r1,-20(sf) */
a=b;   /* MOV -20(sf),r1 */
      /* MOV -8(sf), r1 */
```

Compiling IF... { ... }

IFexpression {action}

Compiles to:

```
test condition
if condition is FALSE then {GOTO forward}
action
forward:
```

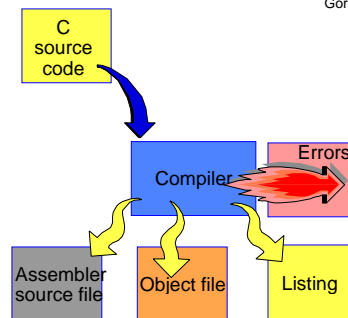
e.g.

```
if (a&0x01==0) { /* a is even */ }
```

Compiles to

```
AND r1, r2 // assume a is already in r2 //
BRA.EQ forward
/* a is even */
forward:
```

Compilers



Syntax Errors Detected by Compilers

Gorry Fairhurst (c) 1999

type-here> **cat compiling.c**

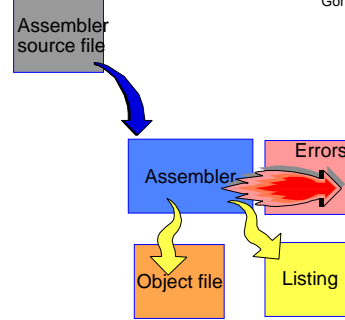
```
main()
{
  int a,b;
  a=12; b=11;
  printf("a==0x%x, b==0x%x\n",a,b);
  a=a&0x01; b=b&0x01;
  printf("a==0x%x, b==0x%x\n",a,b);
}
```

type-here> **cc -c compiling.c**

```
ucbcc: Invalid input file name compiling,
no output generated for this file.
"compiling.c", line 5: syntax error before or at: printf
```

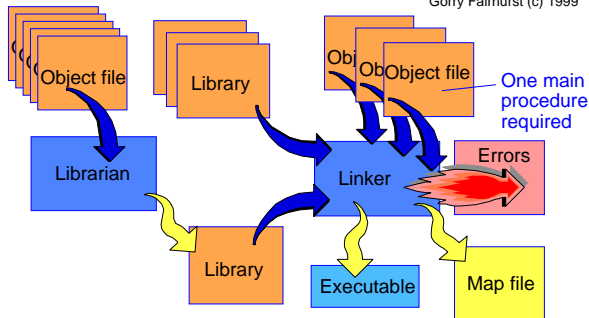
Assemblers

Gorry Fairhurst (c) 1999



The Linker

Gorry Fairhurst (c) 1999



The executable contains a copy of the "text segment" (i.e. the actual machine code for the program)

strcpy Library Procedure

Gorry Fairhurst (c) 1999

```
void strcpy(char in[], out[])
/* procedure to copy the string pointerd to by in[] */
/* to the string out[], the procedure accepts two */
/* pointer parameters, each points to a start of string */
/* note that the last NUL (0x00) must also be copied */
{
  int i;
  i= -1;
next:
  i++;
  out[i] = in[i]
  if in[i] != 0) {goto next;}
  return;
}
```

Programming is like Cooking

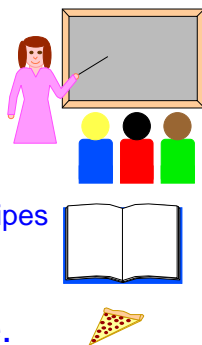
Gorry Fairhurst (c) 1999

A good cook needs to know about:

Good techniques

Some standard recipes

Good taste.



Ten Steps to Perfect Programs

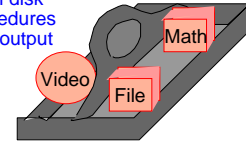
Gorry Fairhurst (c) 1999

- 1) Think first about DATA not the ALGORITHM
- 2) Break the problem down into smaller tasks
(The best procedures are 5-30 lines)
- 3) Don't use global variables!
- 4) Avoid GOTOs (use for, while, case of)
- 5) Beware of pointers and initialised variables
- 6) You are allowed to copy algorithms
- 7) Test procedures thoroughly when you write them
- 8) Add lots and lots of comments - you will forget
- 9) Save the best procedures - you can use them again.
- 10) Don'tbe afraid to throw away your first attempt



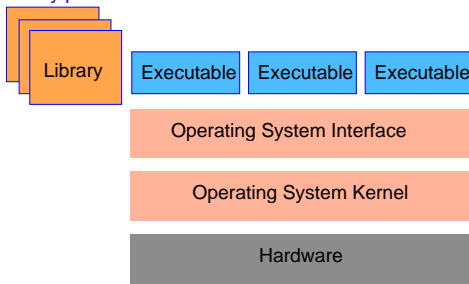
Library procedures act like a **tool box for programmers:**

- Procedures to draw on screen
- Procedures to use menus & mouse
- Procedures to input from keyboard
- Store/Read files on disk
- Mathematical procedures
- Audio/Visual input/output etc.



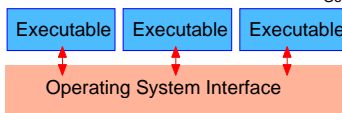
Using library procedures can:
 Save a lot of time
 Take advantage of toolbox updates

Library procedures



Operating Systems

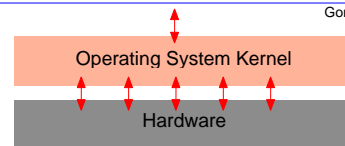
Operating System Interface



The operating system **interfaces** to computer hardware

- Simplifies high-level programs (allows them to ignore details of hardware)
- Allows programs to be moved between computers (provides consistent procedures)
- Allows several programs to execute on same CPU (co-ordinates which one runs when)
- Protects peripherals (stops unauthorised use)

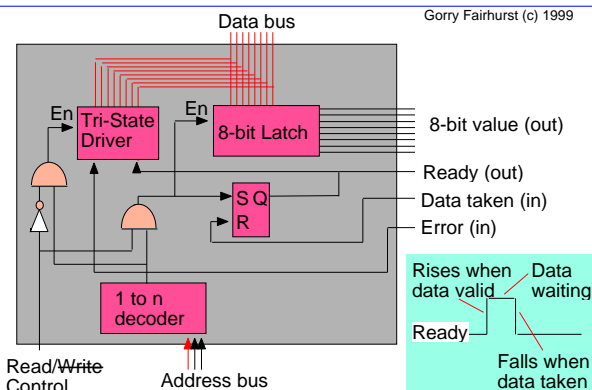
Operating System Kernel



The operating system Kernel **hides** the real hardware

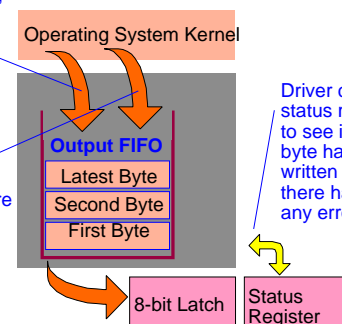
- The kernel is optimised for particular hardware
- Handles all input/out to peripherals (keeps the keyboard, screen, etc. running and allows programs to access data when needed)
- Protects memory (stops programs corrupting one another)

Parallel Port Hardware Interface



Operating System Parallel Port Device Driver

Program calls "printf" library procedure which places all the output data in the parallel port driver output FIFO
 OS calls periodically calls "write" procedure in each driver (polls) to attempt to send the next byte waiting in the output FIFO



Driver checks status register to see if last byte has been written and if there has been any errors