

Processor and Memory

Peter Rounce
P.Rounce@cs.ucl.ac.uk

What did we learn from the Little Man Computer?

0. The difference between addresses, instructions and data.
1. Memory holds both instructions and data
2. Data and instructions are indistinguishable, once stored in memory.
Computer will happily read and execute data and/or process instructions:
this is what happens when programs go wrong!
3. Instruction pointer (IP) (often called the Program Counter, PC)
identifies memory to be read to get a value to process as an instruction.
The sequence of instructions executed is controlled completely by the
sequence of values in IP/PC.
4. The sequence of values in IP/PC is controlled by the instructions
executed.
**[If a program is constructed properly, it all works out as planned,
if not.....!]**

05/11/2009

GC03 Processor and Memory

2

5. Instructions are simple:-

- Move a value from one place to another
- Perform an operation on one or two values (operands) and store the result somewhere.
- Change the contents of the IP/PC: often conditional on the properties of a value (negative, positive, zero)

6. The operation of computer is just the ceaseless repetition of the sequence:-

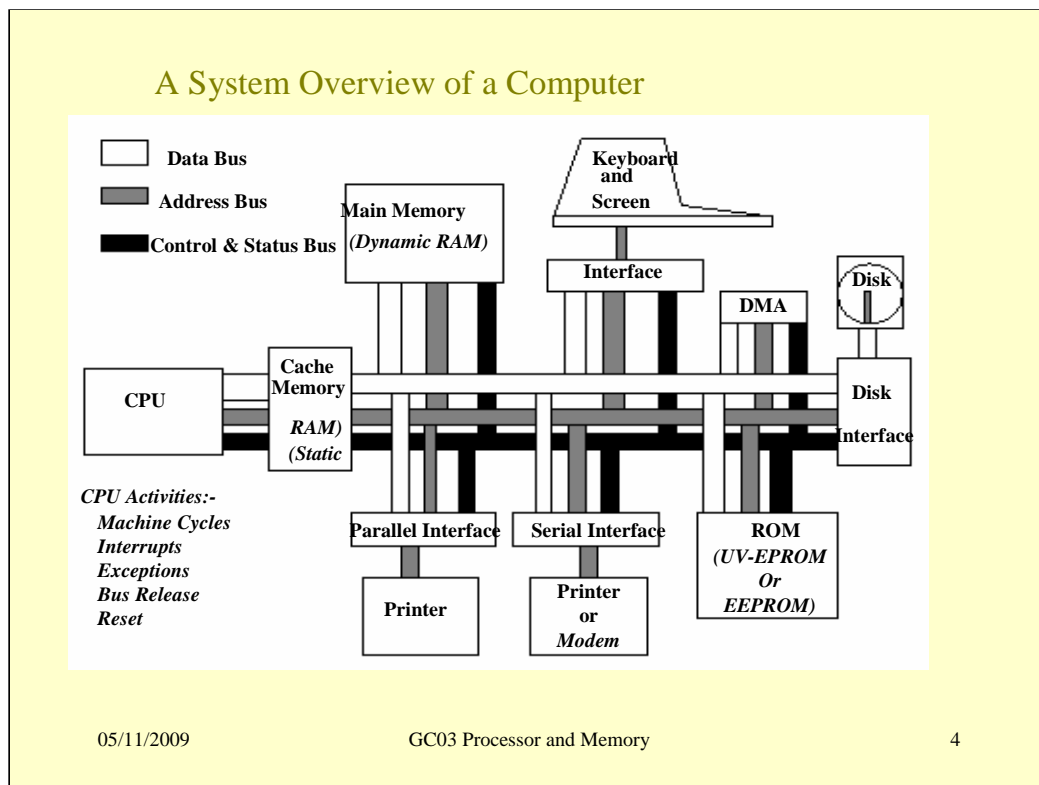
...Fetch instruction---Execute instruction---Fetch---Execute---Fetch...

7. The Fetch Phase is the same for all instructions:

read PC, inc PC, read instruction

8. The Execute Phase is simple because instructions are basically simple

1. read a value (or two) from somewhere
2. test 1 value, or operate on 1 or 2 values, or do nothing
3. store a value somewhere



Processing and Bus Control: CPU (Central Processing Unit, the “processor”)

The CPU controls the computer. It controls the busses to perform data transfers between itself and memory (to read instructions and to read and write data) and also between itself and interfaces (to read and write data to and from input/output devices). It executes instructions and processes data.

Busses: interconnect CPU, Memory, I/O Interfaces

Busses are a bunch of wires connecting the units together so information can be passed between components.

Thus the **address bus** has as many wires as there are bits in an address, e.g. 32 wires in a modern PC, so that all the bits of an address can be output on the address bus at the same time, i.e. in **parallel**: there are other ways the address could be sent, e.g. one bit at a time on a single wire, i.e. **serially**, but parallel is simpler and faster.

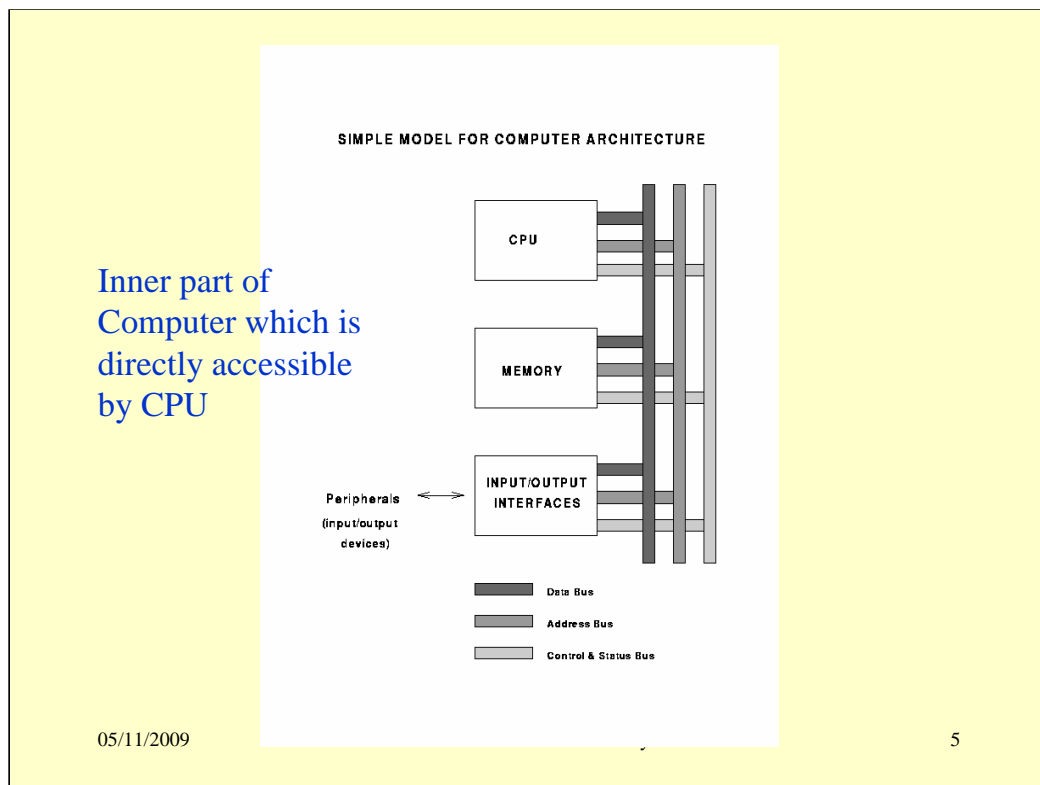
The **data bus** is a bunch of wires, 32 in the PC, so that up to 32 bits of data can be transferred over this bus at the same time. Occasionally, smaller amounts of data may be sent, i.e. in a PC 32 bits, 16 bits or 8 bits of data can be transferred in a single transfer.

The control and status bus contains all the wires that are involved in signalling between components. Some of these will be mentioned during the course, e.g. read control signal, write control signal, interrupt signal.

Other Bus Controllers: DMA devices

A DMA (Direct Memory Access) controller can be set up by the CPU to do data transfers directly between memory and interfaces when large blocks of data have to be transferred, e.g. a program from the disk into memory. When performing the transfers, the DMA controller uses the busses to make the transfers, and at this time the CPU cannot use the busses for transfers. The DMA controller does these block transfers more efficiently than the CPU (later lectures will explain what is meant by more efficiently), and can achieve a higher transfer frequency (number of transfers/second) than the CPU.

Thus the DMA controller is provided for speed reasons and is not a necessary component of a computer – not all computers have one, e.g. not the one in my washing machine.



Memory: types - cache, main, and ROM

There are several memory components in a computer:-

- the main memory is where most programs and data are stored when they are being executed and processed.
- the cache memory is invisible to programs and is purely present for performance reasons, i.e. faster operation, and will be discussed later
- Read-Only-Memory (ROM) keeps its contents when power is removed.

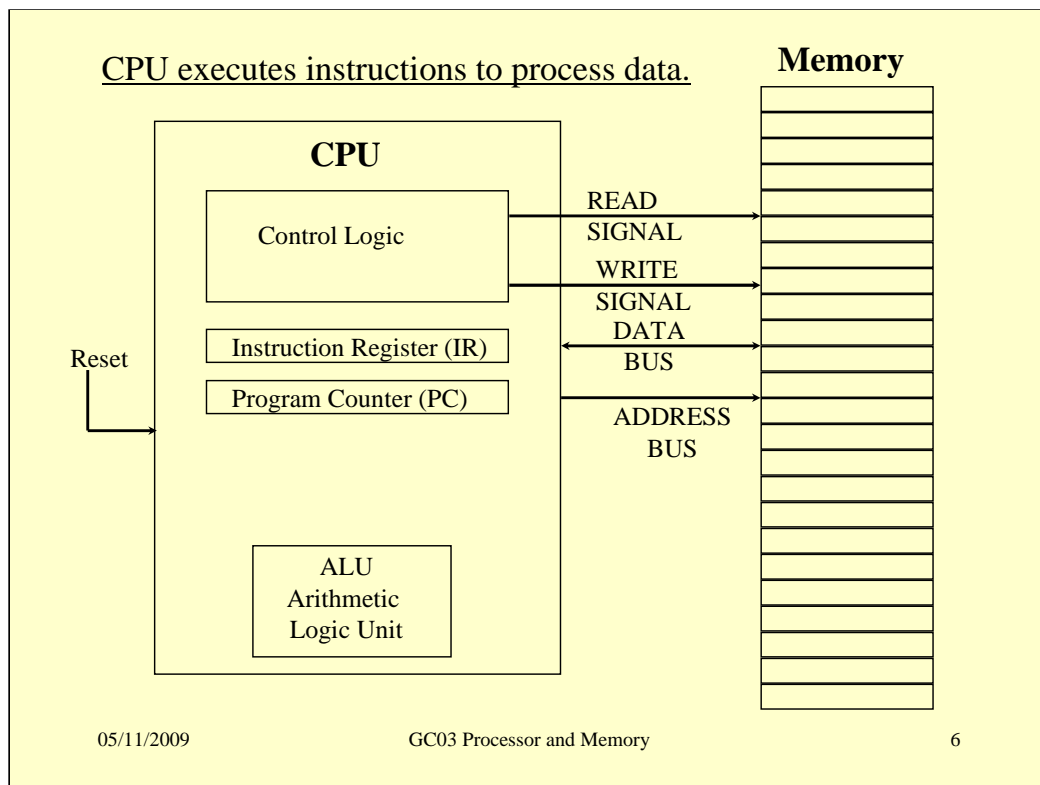
[ROM holds the program to be executed when the computer is turned on: the CPU needs to go somewhere to get instructions to execute and the ROM is where these instructions are held. Modern ROMs can be written, but quite slowly compared to other memory, and this allows ROM contents to be changed when necessary. The BIOS in a PC is held in a ROM on the PC's motherboard, and can, with care, be written to with new software: with care, because if the upgrade does not work properly, the computer will have no program to execute when it starts and will thus not work!]

I/O interfaces: connect I/O devices to Busses

Input-output (I/O) devices are usually complex systems that run autonomously from the CPU, e.g. hard disks, printers, scanners, screen. They may be reliant on the inner parts of the computer (CPU, memory) to send commands or data, or receive data, but they have their own control logic and their own timing domain. Each I/O device is isolated from the inner parts of the computer by an interface, which is a holding point for data in transit between the inner part of the computer and the I/O device.

I/O interfaces look the same to the CPU as the memory, i.e. the CPU can read from and write data to storage elements within the interface and each storage element in the interface has an address. All transfers over the busses look the same: this simplifies the hardware design and manufacture.

[The in-tray and out-tray of the LMC are interfaces: they are not the source or destination of the data they hold, they are just holding points between inside and outside.]



The CPU-Memory interface in more detail.

This has all the components of the Little Man Computer except the I/O interfaces.

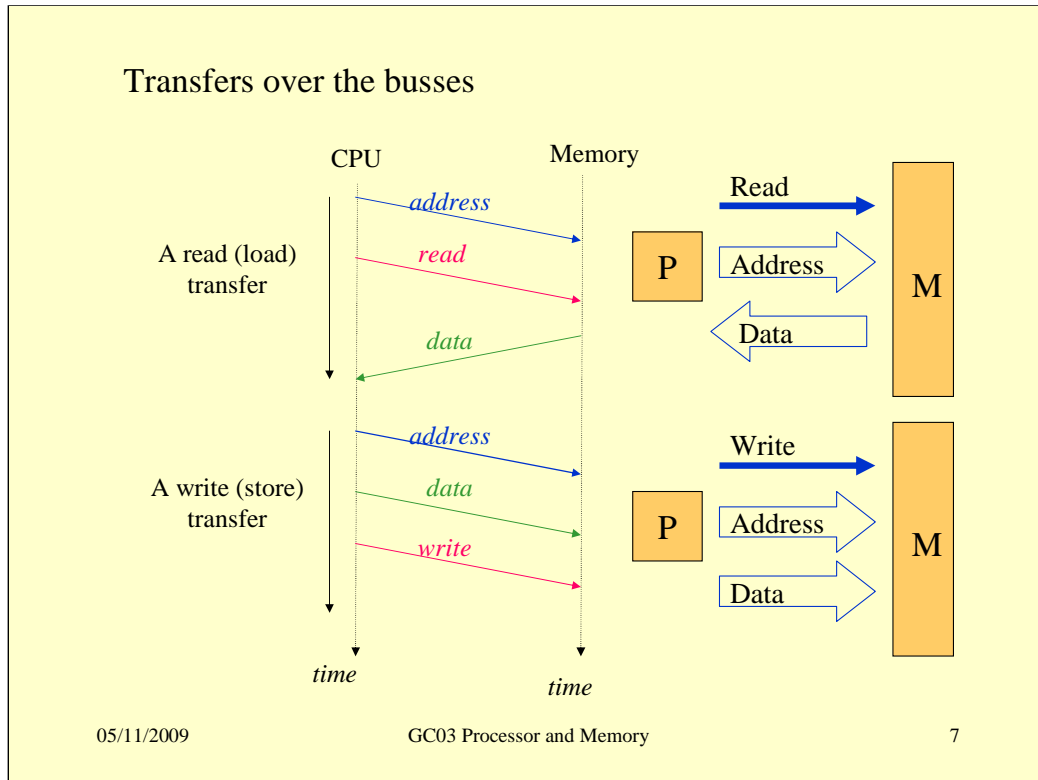
The CPU (Central Processor Unit or just the Processor) holds

- The calculator, normally called the ALU (Arithmetic Logic Unit) – the unit that performs arithmetic (add, subtract) and logic (and, or, not) operations
- The control logic that co-ordinates the CPU's actions and analyses instructions
- The instruction pointer or program counter that indicates the address from which to read the next instruction: it is called the Program Counter by tradition, but it doesn't "count".
- The instruction register (IR) is just a simple store to hold the instruction once it has been read into the CPU, while the instruction is being analysed and executed.
- All CPUs have a Reset signal that operates in a similar way to the Reset button of the LMC

The memory is usually much, much larger than in the LMC.

In current machines, an address will have 32 bits, which means that there are 2^{32} available addresses to the CPU to use. However, most computers do not use all the available addresses – most streets use only a small set of house numbers even though there is a much, much larger range available. Programs should only use the addresses that map to components.

Read and write (control) signals indicate the direction of transfer to the memory: read (into to CPU), write (out of CPU).



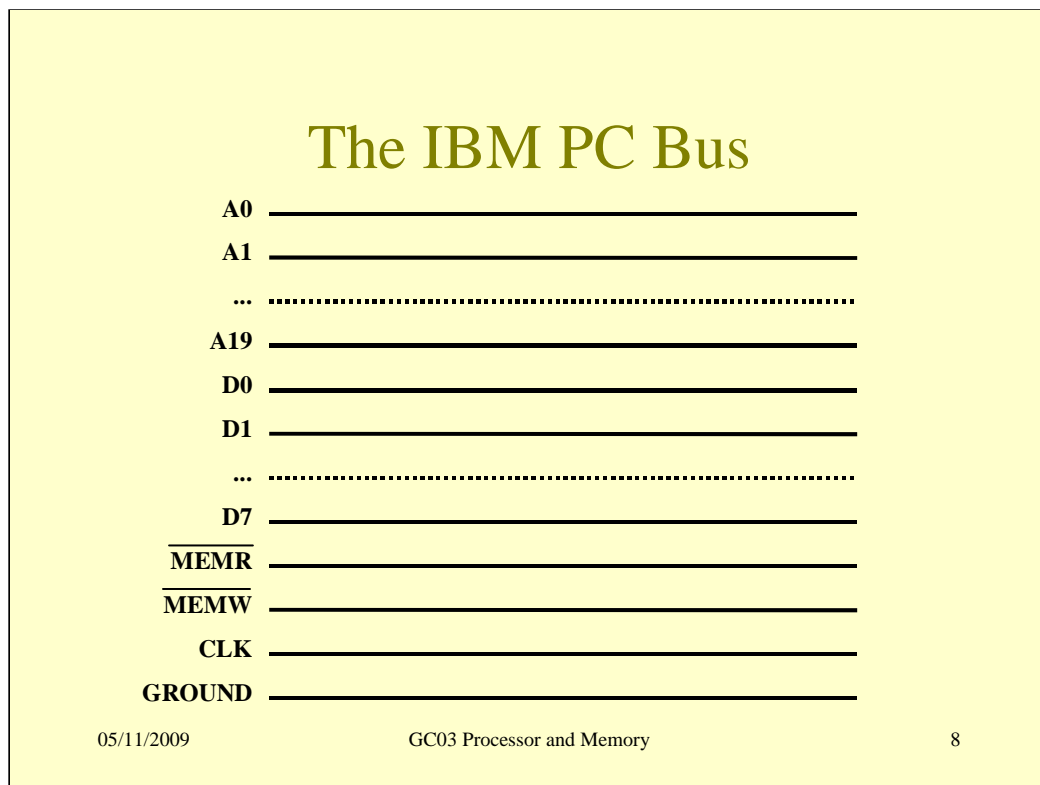
The transfers over the busses are generally performed in a sequence. Both read and write transfers are controlled by the CPU. It is sometimes said that the CPU is the “master” and the memory is the “Slave”.

Read transfer

1. The address to be accessed in the read is output on the address bus by the CPU. Each wire will be set to a voltage to reflect the binary value of the bit of the address being sent on this wire, e.g. 0V for ‘0’, 5V for ‘1’.
2. After a short delay, the read signal is set active by the CPU to indicate a read operation: the read signal can be active or in-active, either a read is in progress or there is no read in progress.
3. After a delay, the memory will respond by putting the data from the addressed memory location on to the data bus.
4. At some point after this, the CPU reads the value from the data bus and then sets the Read signal inactive, completing the read transfer, the address and data busses need not be changed as they will change when another transfer occurs.

Write transfer

1. The address to be accessed in the write is output on the address bus by the CPU.
2. After a short delay, the data to be written is put on the data bus.
3. After a further short delay, the write signal is set active by the CPU to indicate a write operation: the write signal can be active or in-active.
4. After a delay, the CPU will make the WRITE signal inactive completing the write operation.



The slide illustrates **some** of the lines on the bus used in the *original* IBM PC. GROUND is a common earth line, the other lines may be at either +5v or -5v with respect to GROUND, corresponding to 1 or 0 values [Normally +5v -> 1 and -5v -> 0. If the reverse is true the name of the signal has a line above it] .

The original PC used an Intel 8086 CPU.

This outputs a 20-bit address – thus the address lines A19-A0 above, so that the address space of the original PC was 1M byte: each address identified a location holding at most a byte.

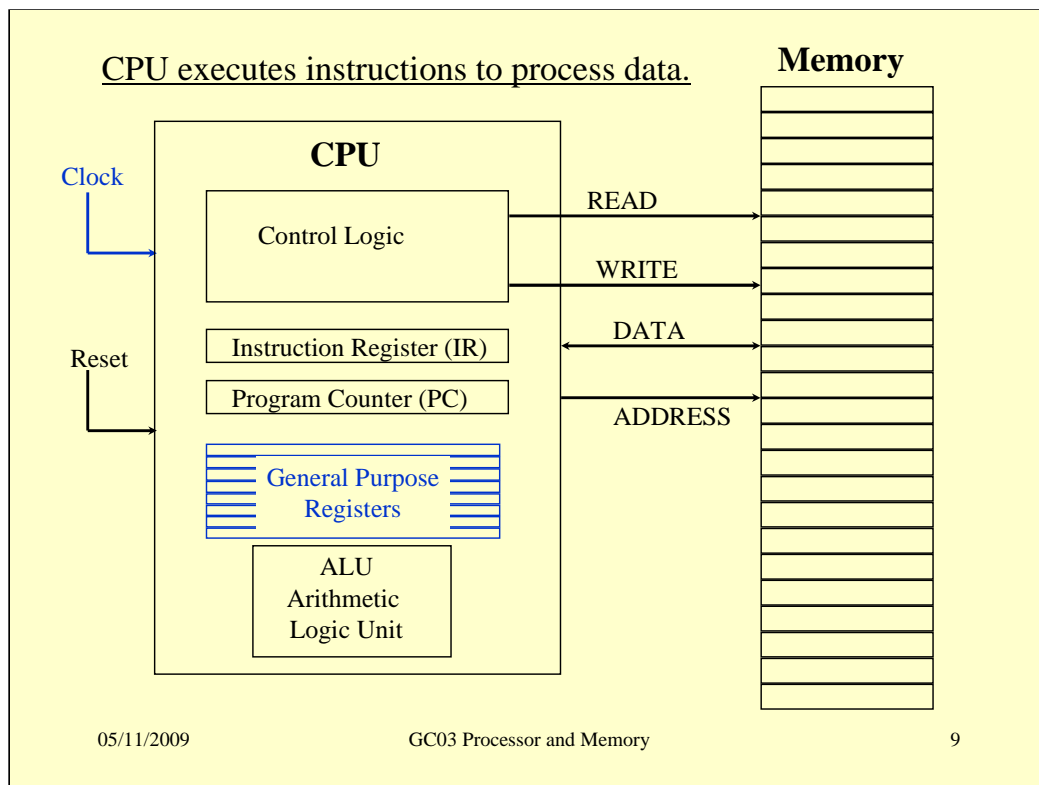
The original PC could only do 8-bit transfers over the above bus, so that there are only 8 data lines in the data bus D7-D0.

Note: the least significant bit of data and address goes on bit 0: D0 for data and A0 for address.

MEMR is the read signal that indicates a read transfer.

MEMW is the write signal that indicates a write transfer.

The bars over these names in the slide are an annotation used by electronic engineers to indicate that these signals indicate an active transfer when they carry a logic '0' value and that a logic '1' on the wire indicates no active transfer.



Most computers have some extra storage inside the CPU.

There are only normally a small number of storage elements, roughly 6 in an Intel Pentium, and 32 in the MIPS.

These elements are called **Registers**.

Registers are provided to improve performance and to reduce program size.

The **performance issue** is that for most of the history of computing, except for a short period in the 1970s, performing a read or write operation to the memory has taken a long time compared the time to do operations within the CPU: memory components are slow compared to the CPU and it takes time for signals to travel between components compared to the time signals take to travel within a component.

To **increase performance**, data values can be stored in registers to make access to them faster. Thus, immediate results of long calculations can be held in registers rather than keeping them in memory. Once the final result is calculated it can be stored to memory or output, and the intermediate results discarded from the registers. Loop counters in **for-loops** can be kept in registers, making it quicker to check and increment them each time around the loop, rather than reading and writing them back and forth to the memory.

The **program size** is reduced by registers, since with only 32 registers, only 5 bits of address are needed to select a register, rather than 32 bits to select a memory location.

The **clock** is a square wave signal, changing from 0V to 5V back to 0V etc with regular intervals. The clock drives the CPU through its sequence of operation: the CPU uses the changes to measure time.

The MIPS Processor

- MIPS addresses are 32 bits long
- MIPS instructions are 32 bits long
- in MIPS a single memory location holds 8 bits
- thus PC holds 32 bits
- thus an instruction cannot hold an address
- thus an instruction must be held in 4 consecutive memory locations

05/11/2009 GC03 Processor and Memory 10

The MIPS is a 32-bit processor in that it can perform integer operations on 32-bit values and can transfer 32-bit values in a single transfer—earlier micro-processors worked on 8 bits or 16-bits and had to do operations or transfers on 32-bit quantities in multiple stages. Memory addresses are 32 bits.

All MIPS instructions are 4 bytes long. This is not the case for all processors. Older architectures, such as the Intel Pentium, have instructions of various different lengths. The MIPS design makes instruction fetching and processing simpler and, therefore, faster.

Memory locations in MIPS are expected to store 8 bits, any more will be unused.

Why not 32 bits? 8-bit storage was chosen because the smallest common data unit used in programs is the character ('a', 'b', etc) which is usually 8-bits (it may be 16-bits in Java, which can use **Unicode**), so one memory location will hold one character and character strings are efficiently stored. Larger units -16-bit integers (short integers), 32-bit integers, 32-bit floats, 32-bit instructions, 64-bit doubles) – can be stored over multiple memory locations, using memory efficiently.

[SUN machines have 32-bit memory locations, but storing strings either uses memory inefficiently or needs more complex software to pack and unpack characters from memory locations.]

Each time an instruction is executed by the MIPS CPU, PC increases by 4.

[MIPS is supposed to stand for **M**icroprocessor without **I**nter-**L**ocking **P**ipeline **S**tages, but, although true as a name (even if you don't yet understand this), I suspect that it chosen as a name because Mips also stands for **M**illions of **I**nstructions **p**er **S**econd, and thus MIPS alludes to a fast processor (well it was fast at the time that it was built).]

Arithmetic and registers

\$0	0	0	0	0	0	0	0
\$1	0	0	0	0	0	1	4
\$2	0	0	4	F	2	0	3
⋮							
\$31	F	F	F	F	F	F	E
PC	0	0	0	0	0	1	4

- MIPS – 32 registers (\$0 to \$31)
 - **Arithmetic only allowed on data read from registers**
 - **“Load and store” architecture (only load and store instructions access memory – all other instructions access registers only)**

- To add two numbers in memory requires 4 instructions
 - 1 “load” register ← memory
 - 2 “load” register ← memory
 - 3 “add” register ← register + register
 - 4 “store” register → memory

05/11/2009

GC03 Processor and Memory

11

MIPS processors have 32 registers each of which can store a 32-bit value. Well, except for register \$0, which always holds the value zero, i.e. \$0 is a read-only register. The designers thought that zero was so useful a value in programming the MIPS that it should be readily available in a register at all times.

MIPS is a **“load and store” architecture**. Only special load and store instructions access memory. This means that all other instructions - arithmetic, branching, etc - can **only** act on registers. Thus to do arithmetic on values stored in memory we must first load the values into registers. This is an important feature of the MIPS design (and other modern RISC designs). It enables the design to be simple, because there are fewer and less complex instructions, and arithmetic to be made very fast, the chip is smaller. In most cases movements to/from memory are quite rare since active values can be loaded into registers at the start of a program section and remain there for quite some time.

Why is MIPS a Load & Store Architecture?

With 32 registers, only 5 address bits are needed to select a register. Some MIPS instructions access 3 different registers. This only requires 15 bits, so that there are still 17 bits left in these instructions to specify the operation and other things. This allows for powerful instructions, completing operations in a single operation, that would have need 2 or more operations in other machines. It also means that the logic to analyse instructions is simple: in other processors, an add instruction may access 2 registers, a register and a memory location, or 2 memory locations – the logic for sorting what to do on each variation is much more complex than that for processing a MIPS add. Simplicity reduces the size and gives faster operation – MIPS more than made up in speed for what it lost in instruction complexity.

If we wanted a MIPS instruction that accesses 3 different memory locations within the one instruction, it would have to have at least 96 bits as well as a few more to specify the operation to be done. The special load & store instructions only need to generate one **32 bit address**, which even then is difficult with only 32 bits of instruction & with some bits being required to specify whether it is a load or store.

MIPS Instructions

- MIPS has 3 instruction formats:
 - R-type - register operands – **all arithmetic**
 - I-type - “address” operands – **load, store, some branches**
 - J-type - jump operands – **other branches**
- All instructions are 1 word long (32 bits)
- Examples of R-type instructions:

add	\$8, \$1, \$2	$\$8 \leftarrow \$1 + \$2$
sub	\$12, \$6, \$3	$\$12 \leftarrow \$6 - \$3$
- Register 0 (\$0) always contains zero

add	\$8, \$0, \$0	$\$8 \leftarrow 0 + 0$
add	\$8, \$1, \$0	$\$8 \leftarrow \$1 + 0$ # copy \$1 to \$8

05/11/2009

GC03 Processor and Memory

12

In the Little man computer we saw that some instructions had different formats to other instructions – LOAD, STORE, ADD, SUBTRACT, and BRANCH instructions had a box number in digits 2 and 3, while IN and OUT had the same opcode character and used digits 2 and 3 to specify whether the instruction is IN or OUT. The HALT instruction did not use digits 2 and 3 at all.

MIPS has three instruction formats (see slide). Having several formats allows more effective use of the bits of the instruction. [Some computer have had many more formats than 3, but again this makes the analysis logic much more complex].

MIPS keeps the number of formats as small as possible, whilst making effective use of the bits.

[The designers of MIPS always aimed at performance, and they did a careful analysis of instruction usage statistics in existing computer so that they only included the keys elements that provide for high performance: this had not been done often enough before.]

Representations such as “add \$8, \$1, \$2” are “human readable” representations. These are **not** what is stored in memory (see next slide).

We can write programs using the human readable interpretations and run a program (called an assembler) to translate these to the real formats that get stored in memory.

R-Type Instructions

- Represented as:

op	rs	rt	rd	shamt	funct
6 bits	5bits	5bits	5bits	5bits	6 bits

- op = operation or type of the instruction
- rs = first register argument (source)
- rt = second register argument (source)
- rd = result of the operation (destination)
- shamt = shift amount (ignore at present)
- funct = particular type of the operation
 - op = 0, (funct = 32) => **add**, For op = 0, (funct = 34) => **sub**

05/11/2009

GC03 Processor and Memory

13

The slide shows the format for the R-type instructions. Variants of this form cover all arithmetic and logical operations and all combinations of registers.

All MIPS instruction, as you will see, use the top 6 bits (the leftmost bits – bits 31-26) to hold the instruction operand. The analysis logic in the CPU uses these 6 bits to identify the instruction format.

rs, rt and rd fields select registers and they may be all different or all the same or any other combination.

Since there are more than 64 instructions in MIPS, the 6-bits of the **funct** field (bits 5-0) are used in identifying the operation to be performed.

Many R-format instructions have 000000 for the opcode and the funct bits identify the operations; add, subtract, and , etc.

The shamt field provides a shift value for some shift operations:

- logical shift left (lsl)
- logical shift right (lsr)
- arithmetic shift right

R-Type Instructions - Examples

e.g. add \$8, \$1, \$2

op	rs	rt	rd	shamt	funct
000000	00001	00010	01000	00000	100000
0	1	2	8	0	0x20
gen	\$1	\$2	\$8	0	add

→ 00000000001000100100000000100000

e.g. sub \$12, \$6, \$3

op	rs	rt	rd	shamt	funct
000000	00110	00011	01100	00000	100010
0	6	3	12	0	0x22
gen	\$6	\$3	\$12	0	sub

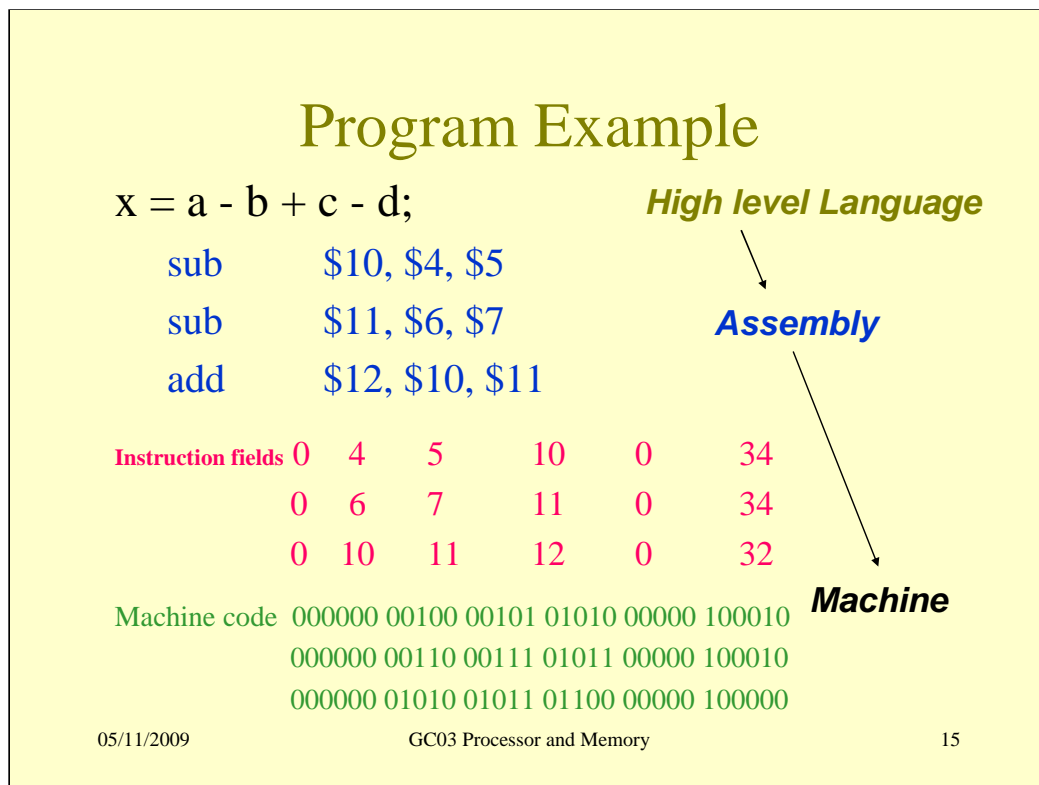
→ 00000000110000110110000000100010

05/11/2009

GC03 Processor and Memory

14

In principle we could write all our programs as a sequence of 32-bit binary words – that, after all, is what the processor actually reads and executes. In practice we will write MIPS programs using the assembly “mnemonics” (add, sub, \$8 etc.).



Of course, modern programs are rarely written in assembly language. Instead a high-level language such as C++ is used. This is translated to assembly language by a program called a **compiler**. The assembly language is then translated to machine language by a program called an **assembler**. Normally we are not interested in the intermediate assembly language version and most compilers subsume the assembly stage and produce machine language directly.

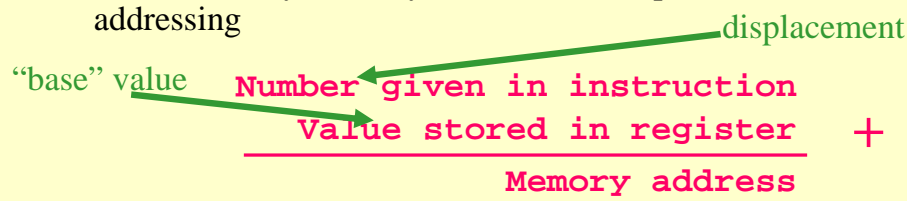
In the slide we assume the following correspondence:

$x \leftrightarrow \$12$
 $a \leftrightarrow \$4$
 $b \leftrightarrow \$5$
 $c \leftrightarrow \$6$
 $d \leftrightarrow \$7$

This example demonstrates the use of registers to hold intermediate results that was referred to in an earlier slide. Using memory to store these would make the code significantly larger.

MIPS Addressing

- Instructions that access registers must include **register addresses**; \$0, \$1 etc.
- Instructions that access memory must include **memory addresses**. Various ways of doing this, some processors have up to 8 different ways!
- MIPS has only one way - “Base” or “displacement” addressing



05/11/2009

GC03 Processor and Memory

16

MIPS has **register addresses** (\$0, \$1, \$2,..., \$31) that need 5-bits of address.

MIPS also needs to generate 32-bit **memory addresses** to access components outside the CPU: a 32-bit address won't fit in a 32-bit instruction.

The different sorts of addresses are identified implicitly by the instruction that is used and by the way that **memory addresses** are produced.

Memory addresses are formed by an addition process that is part of load and store instructions

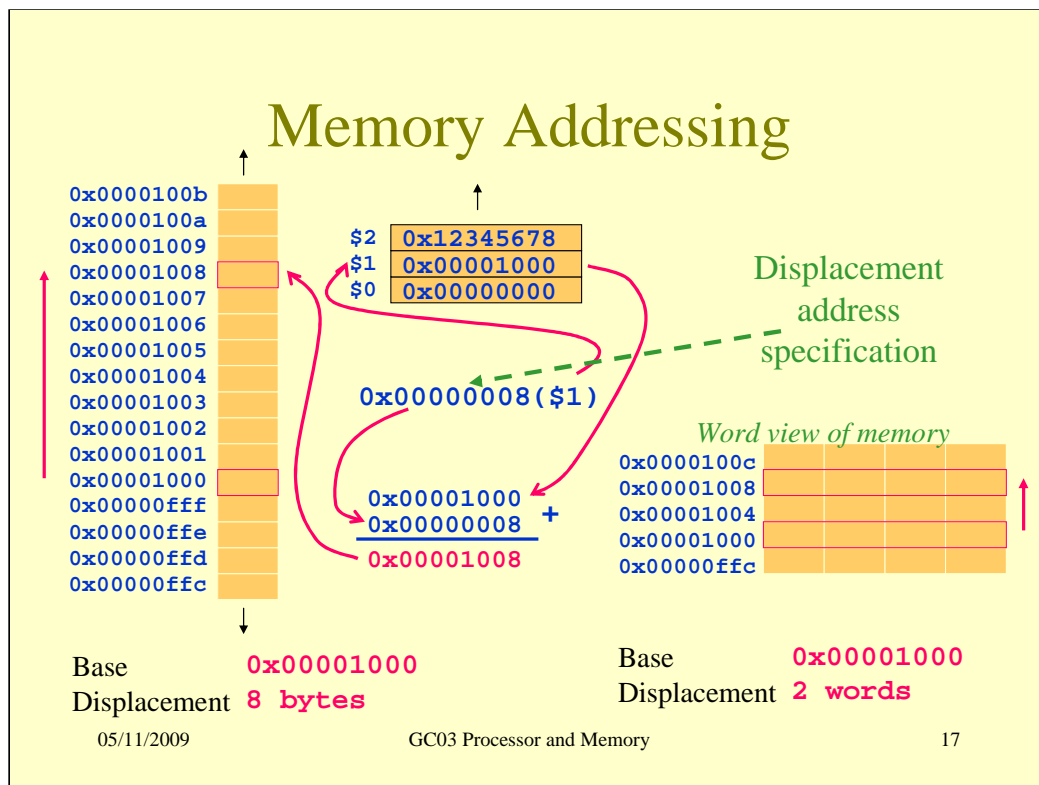
Register addresses are found only within particular bit fields within instructions, e.g. in the *rs*, *rt* and *rd* fields of R-format instructions.

MIPS generates an address by the addition of the contents of one of the 32 registers and a 16-bit signed held in bits 15-0 of load and store instructions.

This address generation method is called displacement or base addressing in architecture text books:

the **register contents** provide a **base** address

the 16-bit field in the instruction provides a **displacement** value that when added to the **base** address specifies an address that is **displaced** from the **base address** by the amount specified by the displacement value.



The slide shows the normal assembly language notation for displacement addressing (e.g. `0x00000008($1)`) and the calculation which is performed by the processor to arrive at a memory address. We can think of the register value as providing a **base** address and the **displacement** as telling the processor how far to look beyond the base. Note:

- It is equally valid to think of the example in the slide as having a base of `0x00000008` and a displacement of `0x00001000`.
- Displacements may be negative (2's complement).

Most often we think of the memory as consisting of 4-byte words. **However, the displacement in the instruction is always measured in bytes.** Thus a displacement of **N words** will appear in an instruction as **4×N bytes**.

A MIPS load or store instruction has to specify:-

- The operation
- The base register for the address calculation
- The 16-bit displacement value
- A 2nd register to receive the data value in a load or supply it in a store

I-Format

6 Bits op	5 Bits rs	5 Bits rt	16-bit offset
---------------------	---------------------	---------------------	--------------------------

Offset may be a 16-bit signed value: $-2^{15} \leq \text{offset} \leq 2^{15}-1$
or an 16-bit unsigned value: $0 \leq \text{offset} \leq 2^{16}-1$
depending on the instruction.

The offset is extended to 32-bits before use: with sign extension for signed values and zero extension for unsigned.

I-Format is used by

- memory access instructions (Load & Store)
- instructions that require a constant value (addi, andi, lui),
- branch instructions

rs and *rt* fields identify registers via the register number.

I-Format

6 Bits op	5 Bits rs	5 Bits rt	16-bit Signed offset
---------------------	---------------------	---------------------	--------------------------------

Load word (32 bits read from memory)

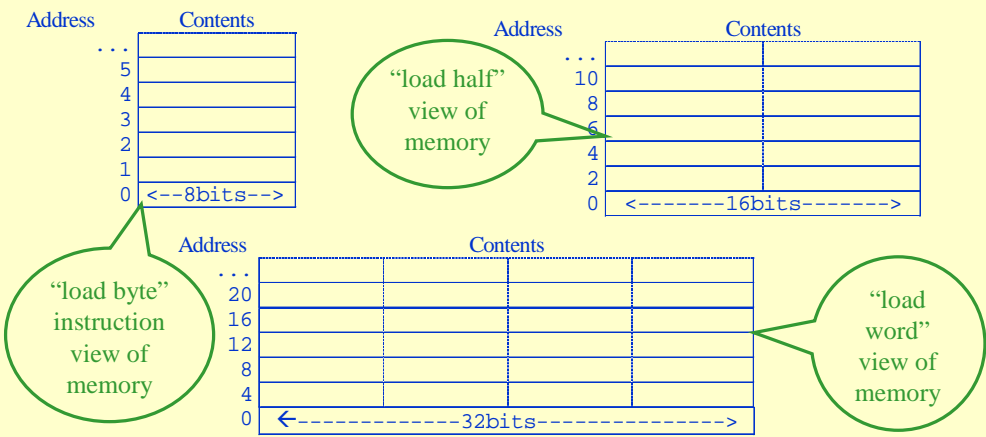
100011 00101 11000 0001 0010 0011 0100 lw \$20, 0x1234(\$5)
 0x23 5₁₀ 20₁₀ 0x1234 Mem[\$5 + 0x00001234] → \$20

Store word (32 bits written to memory)

101011 01111 01000 1000 0000 000 0000 sw \$8, 0x8000(\$15)
 0x2B 15₁₀ 8₁₀ 0x8000 \$8 → Mem[\$15 + 0xFFFF8000]

Grouping bytes

- Three ways of looking at memory



Load Word Instruction (lw)

- **Suppose register \$9 contains 0x12345678**
- **lw \$10, 0(\$9)**
 - copies the word starting at address 0x12345678 into register \$10
- **lw \$10, 80(\$9)**
 - copies the word from address 20 word offsets from 0x12345678 into register \$10
- **Suppose \$9 holds address 0x12345679**
 - lw \$10, 0(\$9) causes an exception!! Why?

05/11/2009

GC03 Processor and Memory

21

This instruction copies data from memory to a register. **The contents of the memory locations are unaltered.**

Store Word (sw)

- **Store word** copies the contents of a register into an address in memory
- Suppose **Z** is an address on a word boundary, and **\$8** contains **Z**
- **sw \$8, 0(\$8)**
 - copies the contents of \$8 into memory starting at address Z
- **sw \$8, 80(\$8)**
 - copies the contents of register \$8 into memory starting at the address which is 20 words offset from Z, i.e. **Z[20]**

05/11/2009

GC03 Processor and Memory

22

This instruction copies data from a register to memory. **The contents of the register are unaltered.**

I-Format

6 Bits op	5 Bits rs	5 Bits rt	16-bit offset
--------------	--------------	--------------	------------------

Add immediate (signed)

001000 00101 11000 1010 1011 1100 1101 **addi \$20, \$5, 0xabcd**

0x8 5₁₀ 20₁₀ 0xabcd [\$5 + 0xffffabcd] → \$20

Add immediate unsigned

001001 00101 11000 1010 1011 1100 1101 **addiu \$20, \$5, 0xabcd**

0x9 5₁₀ 20₁₀ 0xabcd [\$5 + 0x0000abcd] → \$20

Andi immediate (unsigned) – bitwise and

001100 00101 11000 1010 1011 1100 1101 **andi \$20, \$5, 0xabcd**

0xc 5₁₀ 20₁₀ 0xabcd [\$5 and 0x0000abcd] → \$20

I-Format - branch instructions

6 Bits op	5 Bits rs	5 Bits rt	16-bit offset
--------------	--------------	--------------	------------------

Branch equal (beq)

000100 00101 11000 1010 1011 1100 1101 beq \$5, \$20, 0xabcd

0x4 5₁₀ 20₁₀ 0xabcd if (\$5 == \$20)
pc ← pc + 0xffffabcd*4

Branch greater than or equal to zero (bgez)

000001 00101 00000 1010 1011 1100 1101 bgez \$5, 0xabcd

0x4 5₁₀ 0₁₀ 0xabcd if (\$5 >= 0)
pc ← pc + 0xffffabcd*4

Offset = number of instructions to branch over, starting from instruction following branch!

pc value used in addition is value of pc after increment by 4.

I-Format - loading addresses into registers

6 Bits op	5 Bits rs	5 Bits rt	16-bit offset
--------------	--------------	--------------	------------------

Load upper immediate (lui)

001111 00000 10001 0001 0010 0011 0100 **lui \$17, 0x1234**

0xf 0₁₀ 17₁₀ 0x1234 **\$17 ← 0x12340000**

Or immediate (ori)

001101 00000 10001 0001 0010 0011 0100 **ori \$17, \$17, 0x5678**

0xd 17₁₀ 17₁₀ 0x5678 **\$17 ← \$17 | 0x00005678**

The result of these 2 instructions in sequence is:-

\$17 ← 0x12345678

Example

- To sum 3 consecutive elements of a sequence a_0 , a_1 and a_2 , and put the result in a_3 .
- Suppose that these values are stored in memory at an address which is stored in $\$4$ and that each value is 32-bits, i.e. 4 bytes.

```
lw      $10, 0($4)      # $10 =  $a_0$ 
lw      $3, 4($4)       # $3 =  $a_1$ 
add     $10, $10, $3     # $10 =  $a_0 + a_1$ 
lw      $3, 8($4)       # $3 =  $a_2$ 
add     $10, $10, $3     # $10 =  $a_0 + a_1 + a_2$ 
sw      $10, 12($4)     #  $a_3$  gets $10
```

05/11/2009

GC03 Processor and Memory

26

For example, if the 3 elements are stored at addresses 0x00010000, 0x00010004 and 0x00010008, we would begin by setting $\$4$ to 0x00010000.

Related Instructions

- **Load/Store**

lb load byte
lh load halfword
sb store byte
sh store halfword

- **Arithmetic/Logical**

div divide
mult multiply
rem remainder
nor logical nor
xor logical xor
sll shift left logical
sra shift right arith.
srl shift right logical

- **There are variations on some of these - some of which we will discuss later**

05/11/2009

GC03 Processor and Memory

27

Summary

- The “fetch-execute cycle”
- Registers
 - Arithmetic and logical instructions
 - MIPS instructions
 - Storing instructions in memory
- Load and store instructions
- Memory addressing
 - MIPS displacement addressing
- Further reading: Patterson & Hennessey pp. 105 - 122

05/11/2009

GC03 Processor and Memory

28