Computer Instructions

CSE 410, Spring 2009

Computer Systems

http://www.cs.washington.edu/410

Reading and References

- Readings
  - » *Computer Organization and Design*
    - Section 2.1, Introduction
    - Section 2.2, Operations of the Computer Hardware
    - Section 2.3, Operands of the Computer Hardware
    - Section 2.5, Representing Instructions
- Other References
  - » *See MIPS Run, D Sweetman*
    - section 8.6, Instruction encoding
    - section 10.2, Endianness

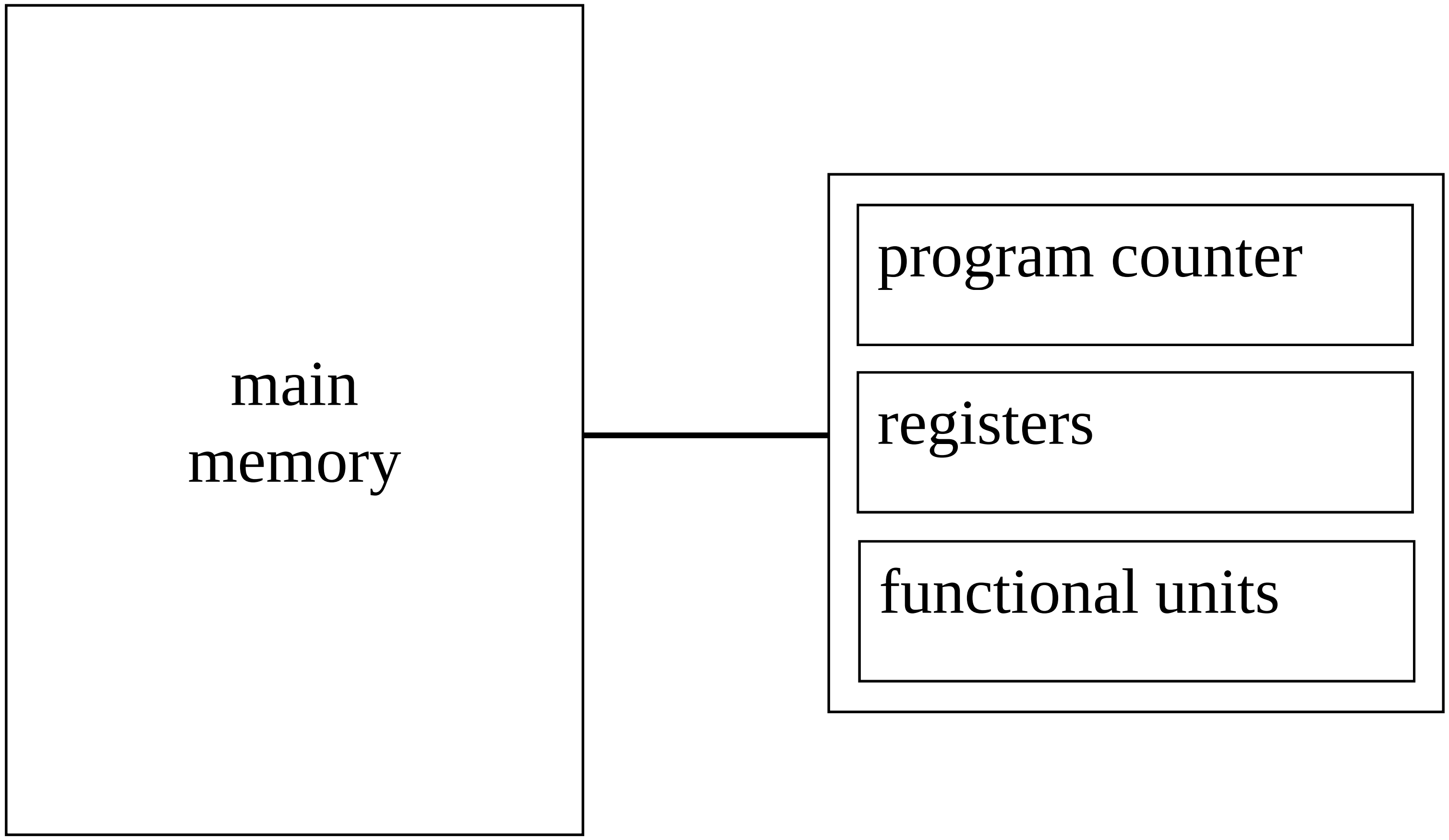
Today’s Agenda

- //pick a language
- A = B + C;   //say, c

- Lets convert this to a much more primitive programming language
  - » Assume A,B,C are associated with $t0,$t1,$t2

- add $t0, $t1, $t2   #say, MIPS

A very simple organization

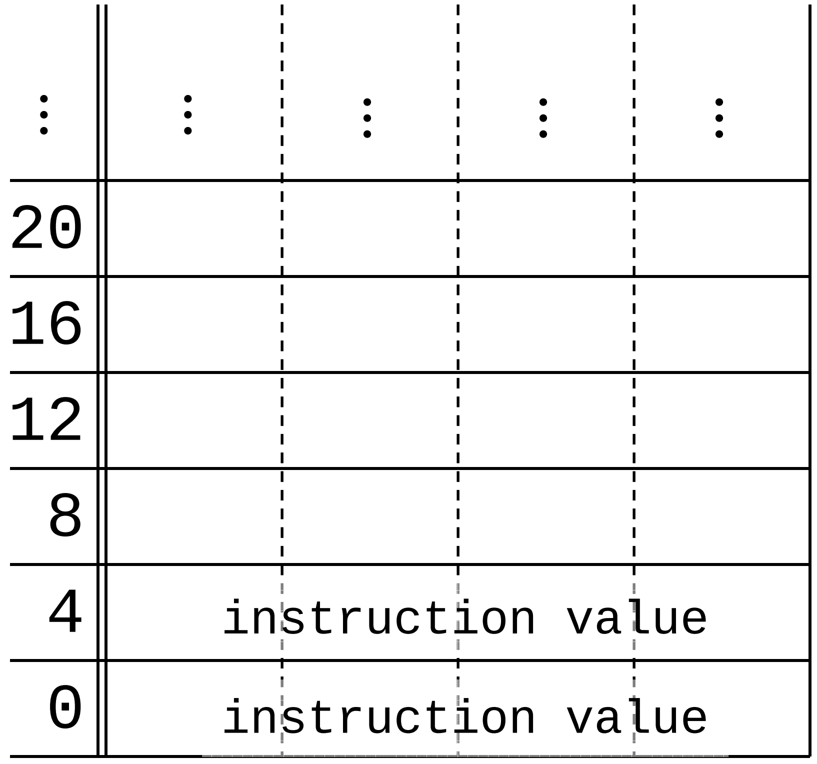


Instructions in main memory

- Instructions are stored in main memory
  - » each byte in memory has a number (an address)
- Program counter (PC) points to the next instruction
  - » All MIPS instructions are 4 bytes long, and so instruction addresses are always multiples of 4
- Program addresses are 32 bits long
  - » 2^{32} = 4,294,967,296 = 4 GigaBytes (GB)

Instructions in memory

instruction
addresses



Fetch/Execute Cycle

- Operation of a computer:

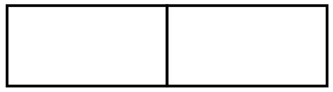
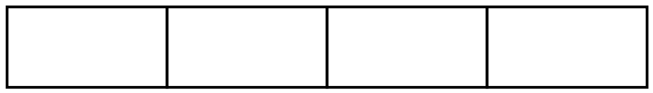
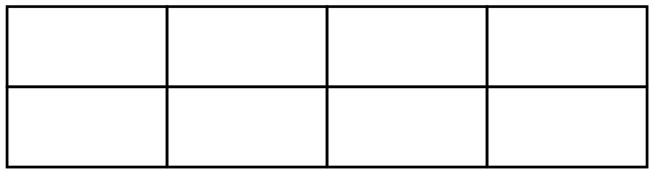
```
while (processor not halted) {
    fetch instruction at memory location (PC)
    PC = PC + 4 (increment to point to next instruction)
    execute fetched instruction
}
```

- Instructions execute sequentially unless a jump or branch changes the PC to cause the next instruction to be fetched from somewhere else

Some common storage units

Note that a *byte* is 8 bits on almost all machines.
The definition of *word* is less uniform (4 and 8 bytes are common today).

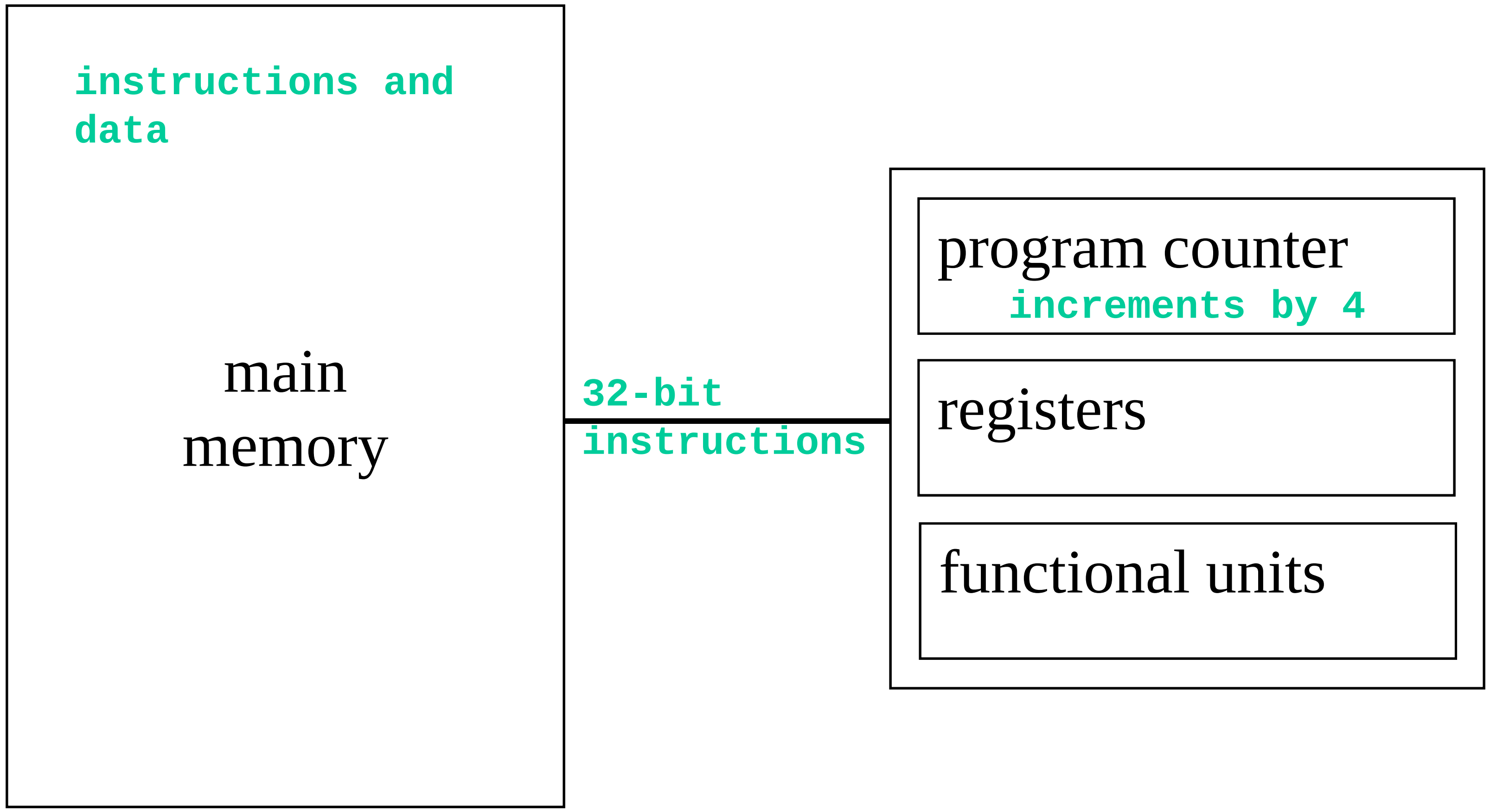
A nibble is 4 bits (half a byte!)

unit	*# bits*	
byte	8	
half-word	16	
word	32	
double word	64	

Alignment

- An object in memory is "aligned" when its address is a multiple of its size
- Byte: always aligned
- Halfword: address is multiple of 2
- Word: address is multiple of 4
- Double word: address is multiple of 8
- Alignment simplifies load/store hardware
  - » And is required by MIPS, but not x86

System organization so far



MIPS Registers

- 32 bits wide
  - » 32 bits is 4 bytes
  - » same as a word in memory
  - » signed values from -2^{31} to $+2^{31}-1$
  - » unsigned values from 0 to $2^{32}-1$
- easy to access and manipulate
  - » 32 registers (not related to being 32 bits wide)
  - » on chip, so very fast to access

Register addresses

- 32 general purpose registers
- how many bits does it take to identify a register?
  - » 5 bits, because $2^5 = 32$
- 32 registers is a compromise selection
  - » more would require more bits to identify & cost more to save/restore when switching processes
  - » fewer would be harder to use efficiently

Register numbers and names

number	*name*	*usage*
`0`	**`zero`**	always returns 0
`1`	**`at`**	reserved for use as assembler temporary
`2-3`	**`v0, v1`**	values returned by procedures
`4-7`	**`a0-a3`**	first few procedure arguments
`8-15, 24, 25`	**`t0-t9`**	temps - can use without saving
`16-23`	**`s0-s7`**	temps - must save before using
`26,27`	**`k0, k1`**	reserved for kernel use - may change at any time
`28`	**`gp`**	global pointer
`29`	**`sp`**	stack pointer
`30`	**`fp`** `or` **`s8`**	frame pointer
`31`	**`ra`**	return address from procedure

How are registers used?

- Many instructions use 3 registers
  - » 2 source registers
  - » 1 destination register
- For example
  - » **`add $t1, $a0, $t0`**
    - add a0 and t0 and put result in t1
  - » **`add $t1,$zero,$a0`**
    - move contents of a0 to t1 (t1 = 0 + a0)

R-format instructions: 3 registers

- 32 bits available in the instruction
- 15 bits for the three 5-bit register numbers
- The remaining 17 bits are available for specifying the instruction
  - » 6-bit op code - basic instruction identifier
  - » 5-bit shift amount ($2^5 = 32$…)
  - » 6-bit function code

R-format fields

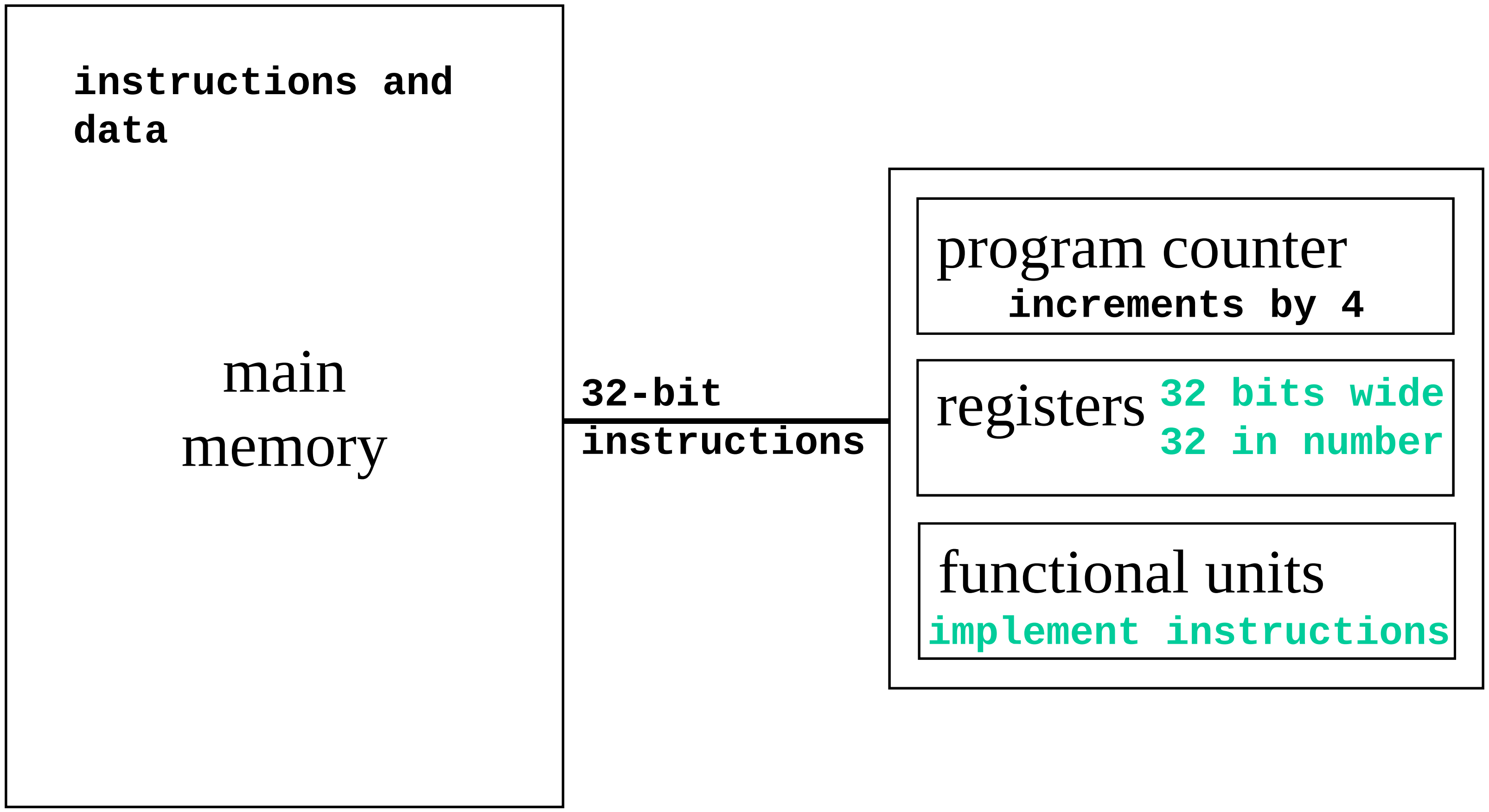
op code	source 1	source 2	dest	shamt	function
6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*

- some common R-format instructions
  - » arithmetic: **add, sub, mult, div**
  - » logical: **and, or, sll, srl**
  - » comparison: **slt** (set on less than)
  - » jump through register: **jr**

Bits are just bits

- The bits mean whatever the designer says they mean when the ISA is defined
- How many possible 3-register instructions are there?
  - » $2^{17} = 131{,}072$
  - » includes all values of op code, shamt, function
- As the ISA develops over the years, the encoding tends to become less logical

System organization again



Transfer from memory to register

- Load instructions
  - » **`word:      lw  rt, address`**
  - » **`half word: lh  rt, address`**
    `           lhu rt, address`
  - » **`byte:      lb  rt, address`**
    `           lbu rt, address`
- signed load => sign bit is extended into the upper bits of destination register
- unsigned load => 0 in upper bits of register

Transfer from register to memory

- Store instructions

```
» word:      sw  rt, address
» half word: sh  rt, address
» byte:      sb  rt, address
```

The “address” term

- There is one basic addressing mode:

  offset + base register value
- Offset is 16 bits (± 32 KB)
- Load word pointed to by s0, add t1, store

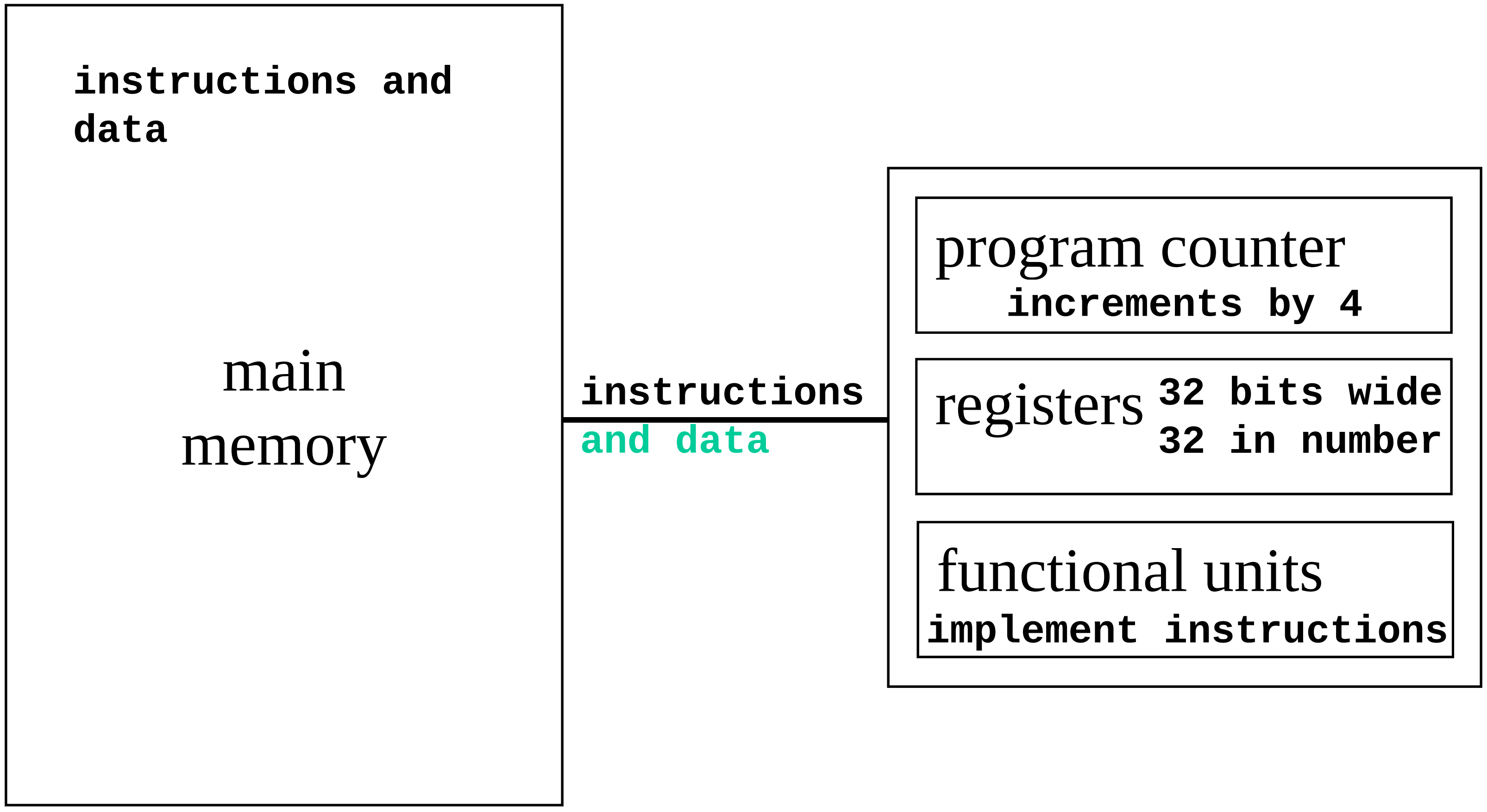
```
lw      $t0,0($s0)
add     $t0,$t0,$t1
sw      $t0,0($s0)
```

I-format fields

op code	base reg	src/dest	offset or immediate value
6 bits	*5 bits*	*5 bits*	*16 bits*

- The contents of the base register and the offset value are added together to generate the address for the memory reference
- Can also use the 16 bits to specify an immediate value, rather than an address

Instructions and Data flow



The eye of the beholder

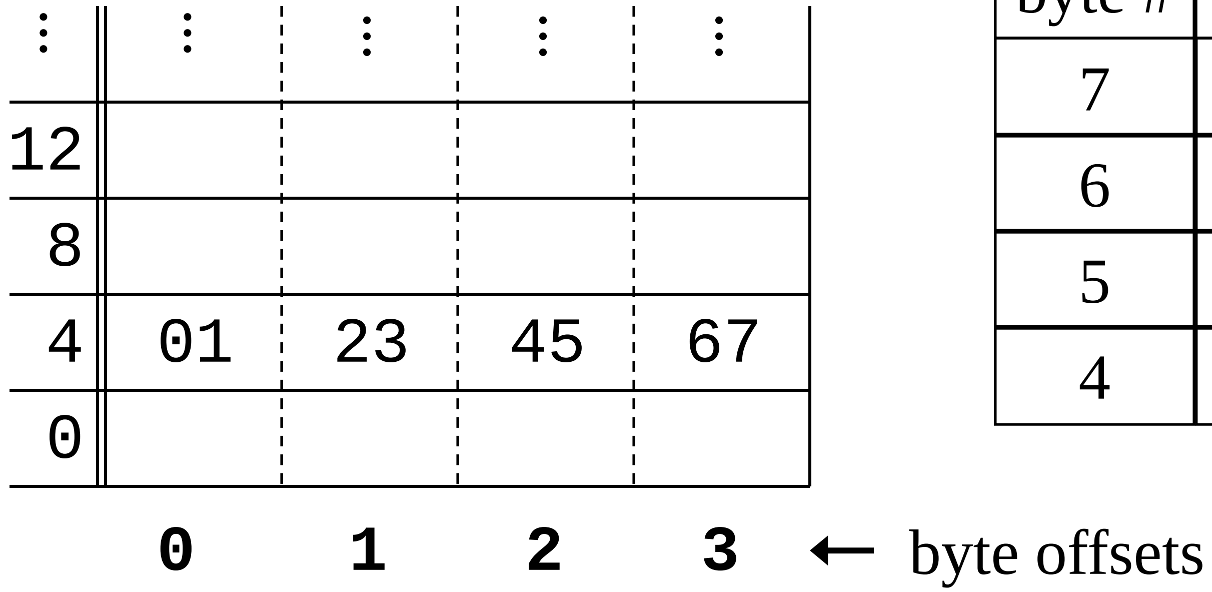
- Bit patterns have no inherent meaning
- A 32-bit word can be seen as
  - » a signed integer (± 2 Billion)
  - » an unsigned integer or address pointer (0 to 4GB)
  - » a single precision floating point number
  - » four 1-byte characters
  - » an instruction

Big-endian, little-endian

- A 32-bit word in memory is 4 bytes long
- But which byte is which address?
- Consider the 32-bit number 0x01234567
  - » four bytes: 01, 23, 45, 67
  - » most significant bits are 0x01
  - » least significant bits are 0x67

Data in memory- big endian

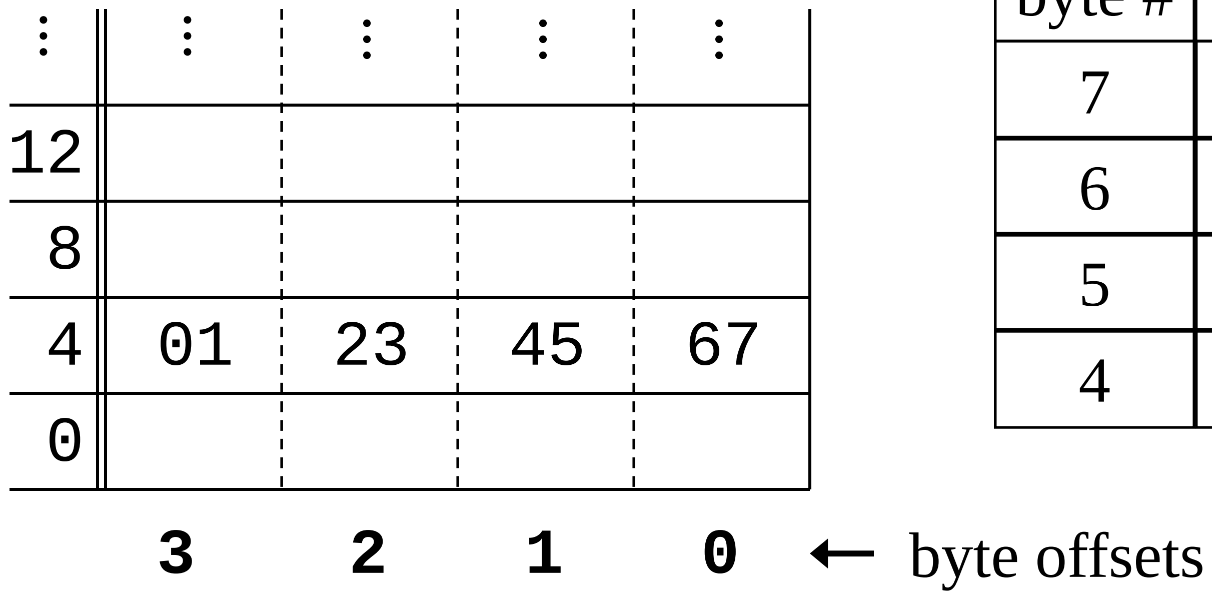
Big endian - **most** significant bits are in byte 0 of the word



byte #	contents
7	67
6	45
5	23
4	01

Data in memory- little endian

Little endian - **least** significant bits are in byte 0 of the word



byte #	contents
7	01
6	23
5	45
4	67