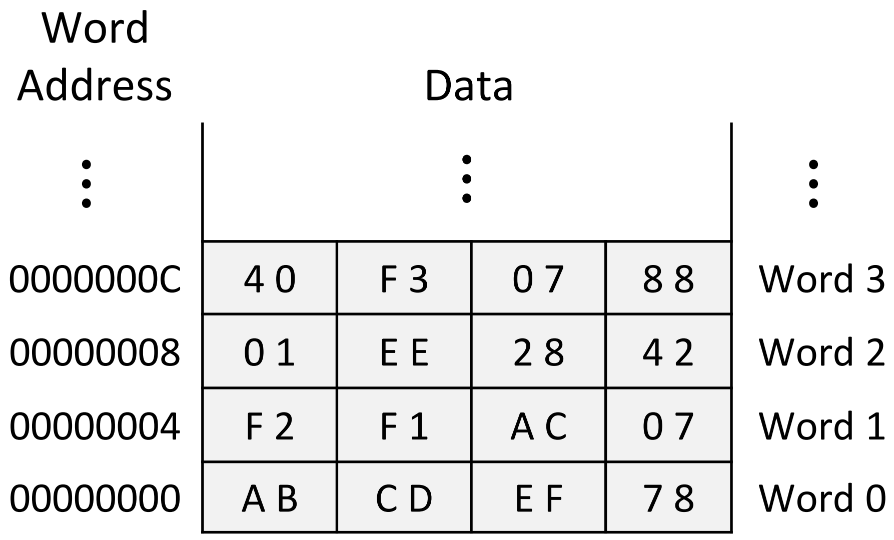
Machine Architecture - Lecture 5



Word Address	Data				
⋮		⋮			⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

Ioannis Ivrissimtzis

ioannis.ivrissimtzis@durham.ac.uk

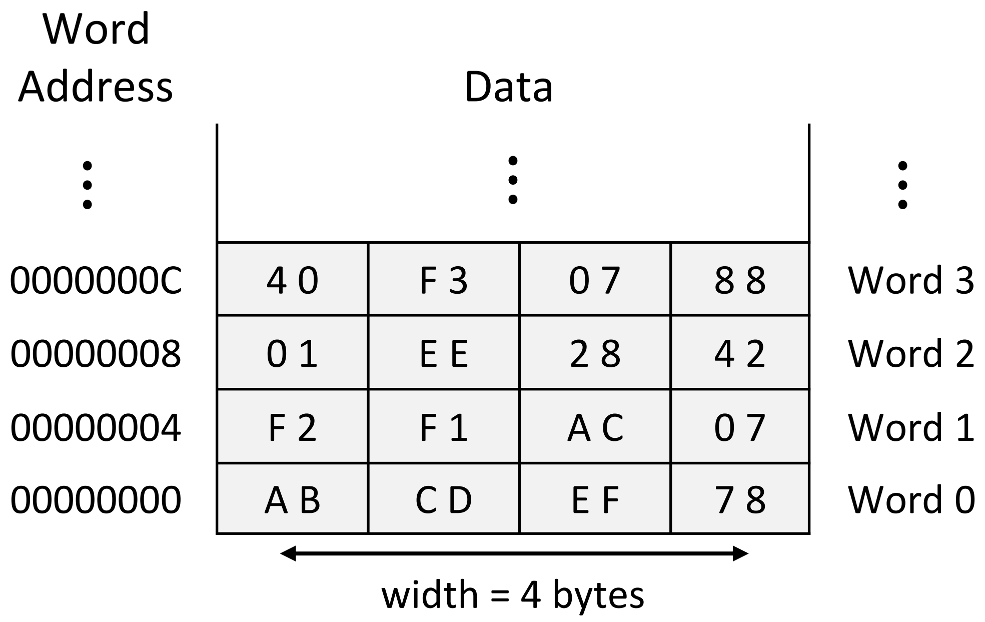
MIPS – memory map and addressing modes

Slide material acknowledgements: Magnus Bordewich

MIPS memory map

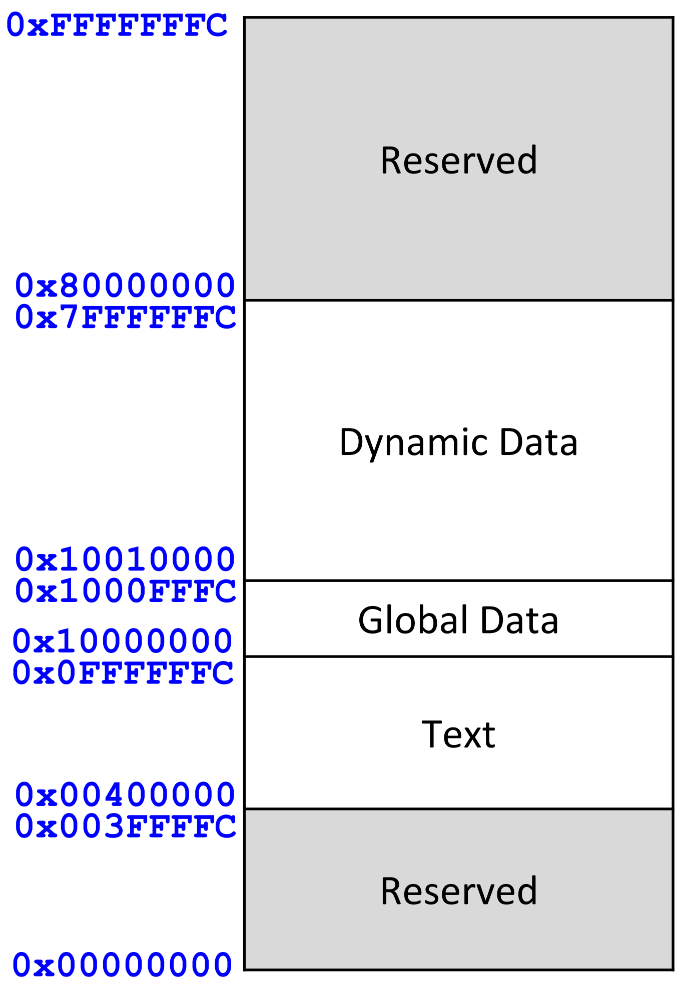
word : 32-bits (in MIPS, high level architectural choice)
byte : 8-bits (always, by the definition)

MIPS uses 32-bit memory addresses and memory is byte addressable.



Word Address	Data				
⋮		⋮			⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

MIPS memory map



With 32-bit addresses, the MIPS address space spans

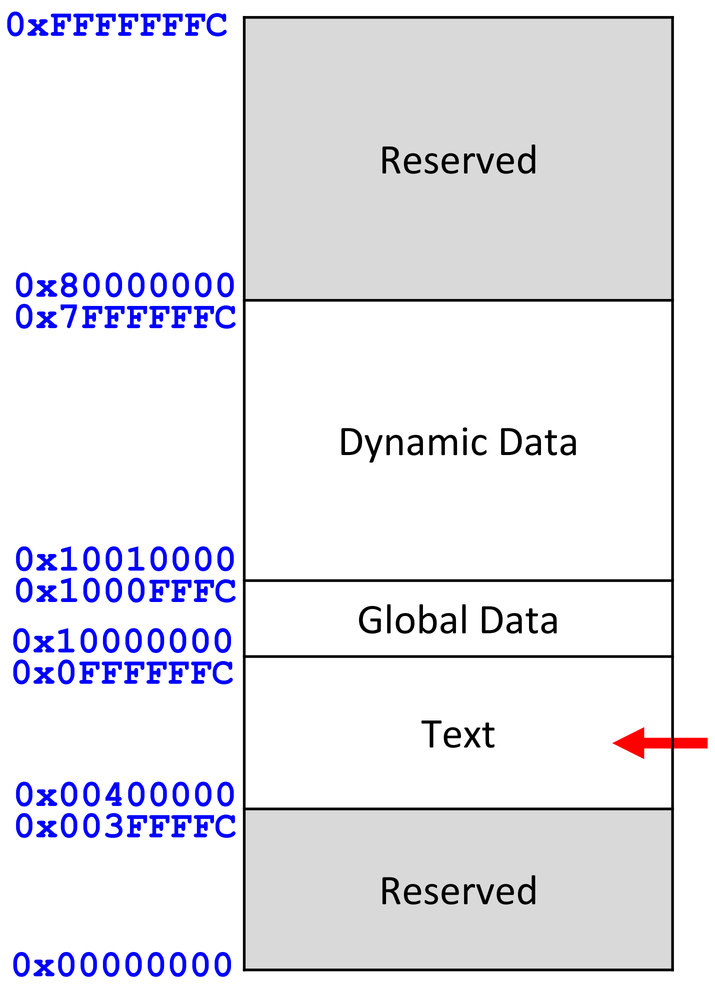
2^{32} bytes = 4 gigabytes (GB).

Word addresses are divisible by 4 (i.e. the two least significant bits are 0) and range from 0 to 0xFFFFFFFC.

The MIPS architecture divides the address space into four parts:

- text segment
- global data segment
- dynamic data segment
- reserved segments

The text segment



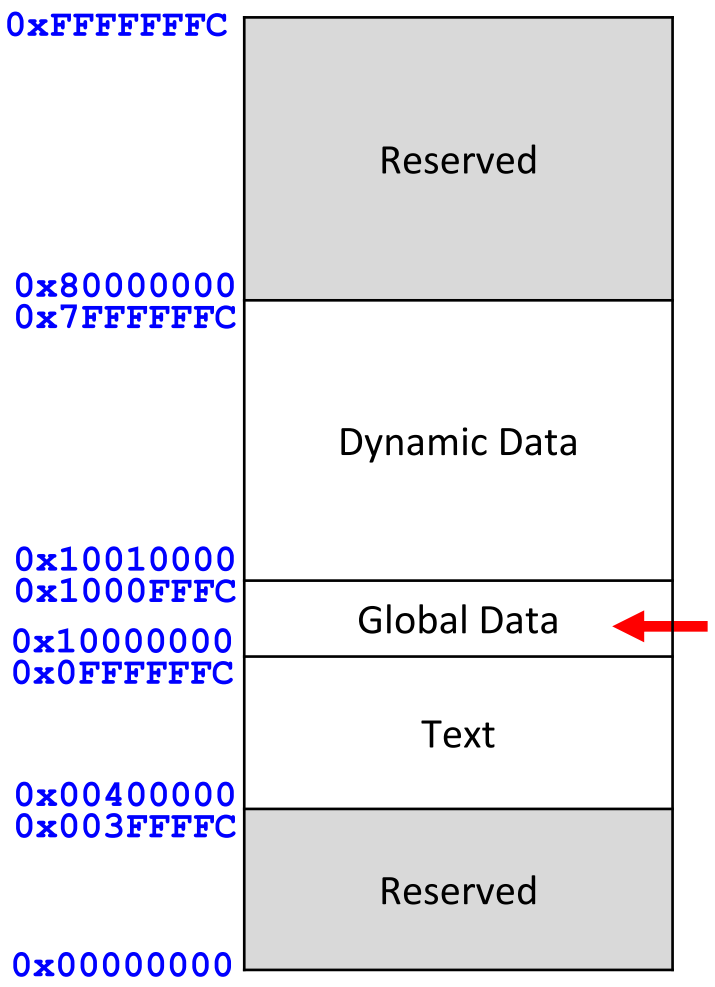
The text segment stores the machine language program.

It can store almost 256 MB of code.

The four most significant bits of any word address in text segment are all 0 (and the two least significant as we saw in the previous slide).

Thus, the 26 bits of the `addr` field of the **`j`** instruction suffice to specify the address of any instruction stored in the text segment.

The global data segment

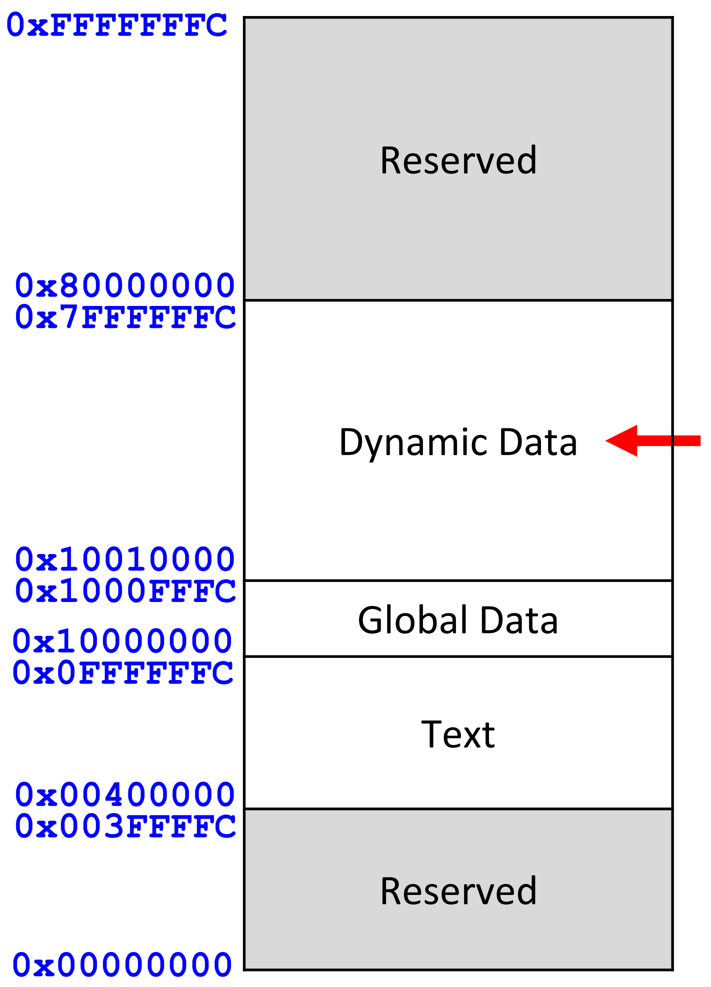


The global data segment stores global variables, which can be seen by all functions in a program. It can store 64 KB of data.

Global variables are accessed using the pointer $gp.

By convention, we initialise **$gp** at the middle of the global data segment at value **0x10008000**. The value of **$gp** stays constant throughout execution, and global variables are addressed as offsets from **0x10008000**.

The dynamic data segment

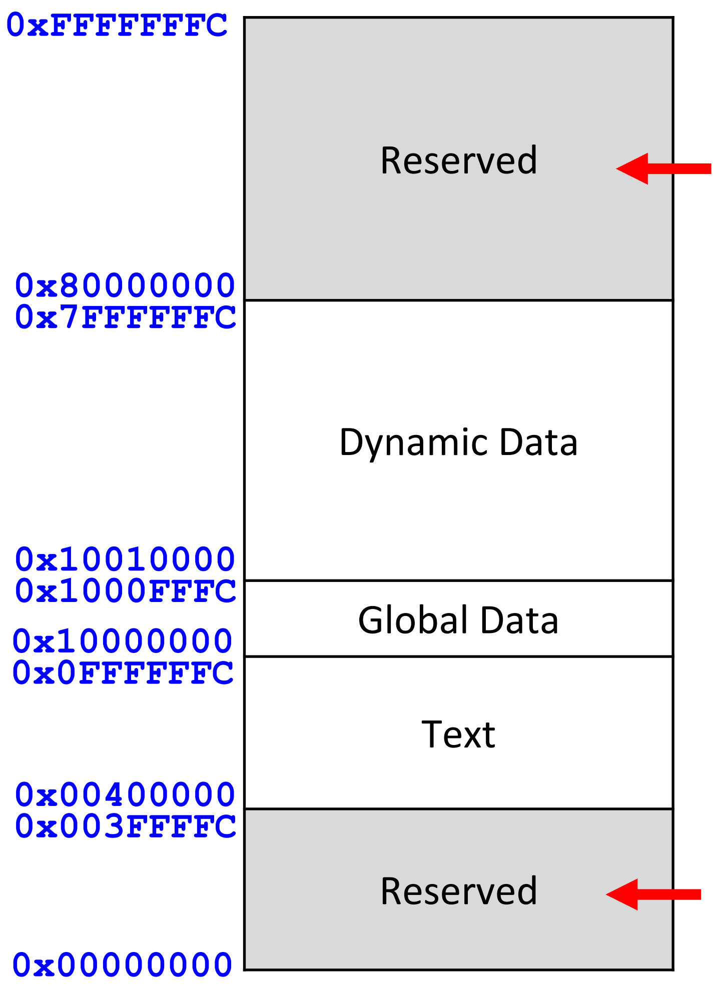


The dynamic data segment stores data that are dynamically allocated and deallocated throughout the execution of the program.

It is the largest segment of memory used by a program, spanning almost 2 GB of the address space.

Data in this segment are stored in a stack and a heap. These two data structures are covered in the ADS module.

The reserved segments



The reserved segments are used by the operating system and cannot directly be used by the program.

MIPS

addressing modes

MIPS addressing modes

$$\left.\begin{array}{l}\text{Register Only}\\ \\ \text{Immediate}\\ \\ \text{Base Addressing}\end{array}\right\}\quad\text{Reading and writing operands}$$

$$\left.\begin{array}{l}\text{PC-Relative}\\ \\ \text{Pseudo-direct}\end{array}\right\}\quad\text{Writing the Program Counter}$$

Register Only and immediate

Register Only

Uses registers for all source and destination operands.

R-type instructions use Register Only addressing.

Immediate

Uses registers and a 16-bit immediate as operands.

Some of the I-type instructions use Immediate addressing (depending on how the 16-bit immediate is used).

Base addressing

Used in memory access instructions.

Implemented by I-type instructions.

The address of the memory operand is computed by adding the base address in register `rs` to a 16-bit offset stored in `imm`.

Base addressing

The instructions:

```
lw      # load word
lb      # load byte
sw      # store word
sb      # store byte
```

read and write data from and to the memory.

Base addressing

Word Address	Data				
⋮		⋮			⋮
0000000C	40	F3	07	88	Word 3
00000008	01	EE	28	42	Word 2
00000004	F2	F1	AC	07	Word 1
00000000	AB	CD	EF	78	Word 0

```
lw $s0, 0($0)          # read data word 0 (0xABCDEF78 in the
                       # above example) and load it into $s0
```

Notice the special assembly syntax for this I-Type instruction.

The word address 0 is the sum of the `imm` 0 and the value at register $0, which is always zero (check the list of registers in the previous lecture).

Base addressing

```
sw $s3, 4($0)        # write $s3 to data Word 1

sw $s4, 0x20($0)     # write $s4 to data Word 8
                     # Notice the use of hexadecimal in the
                     # imm, which is supported by the MIPS
                     # assembly

sw $s5, 200($0)      # write $s5 to data Word 50
```

Base addressing

The assembly code

```
lw $t2, 32($0)
```

is translated to the machine language I-Type instruction

op	rs	rt	imm
35	0	10	32
100011	00000	01010	0000 0000 0010 0000
6 bits	5 bits	5 bits	16 bits

Base addressing

The assembly code

```
sw $s1, 4($t1)
```

is translated to the machine language I-Type instruction

op	rs	rt	imm
43	9	17	4
101011	01001	10001	0000 0000 0000 0100
6 bits	5 bits	5 bits	16 bits

MIPS addressing modes

Register Only Immediate Base Addressing	Reading and writing operands
PC-Relative Pseudo-direct	Writing the Program Counter

PC-relative addressing

Branching instructions can use PC-relative addressing to specify the new value of the PC (Program Counter) if the branch is taken.

Consider the following MIPS Assembly code fragment:

```
0x40    loop:  add  $t1, $a0, $s0
0x44           lb   $t1, 0($t1)
0x48           add  $t2, $a1, $s0
0x4C           sb   $t1, 0($t2)
0x50           addi $s0, $s0, 1
0x54           bne  $t1, $0, loop
0x58           lw   $s0, 0($sp)
```

On the left-hand side, in hexadecimal, is the address of the word storing the instruction [just indicative, the addresses are not in the Text segment of the memory that stores the program].

PC-relative addressing

```
0x40    loop:  add  $t1, $a0, $s0
0x44           lb   $t1, 0($t1)
0x48           add  $t2, $a1, $s0
0x4C           sb   $t1, 0($t2)
0x50           addi $s0, $s0, 1
0x54           bne  $t1, $0, loop
0x58           lw   $s0, 0($sp)
```

The branching instruction in this fragment is the I-Type `bne`.

It compares the values of registers `$t1` and `$0` and **b**ranches when they are **n**ot **e**qual.

If the branch is taken, the new value of the PC is 0x40, corresponding to the label `loop`.

PC-relative addressing

```
0x40    loop:  add  $t1, $a0, $s0
0x44           lb   $t1, 0($t1)
0x48           add  $t2, $a1, $s0
0x4C           sb   $t1, 0($t2)
0x50           addi $s0, $s0, 1
0x54           bne  $t1, $0, loop
0x58           lw   $s0, 0($sp)
```

Notice that when we use assembly, we do not have to worry how the new PC value will be computed. We just use labels.

But how is the assembly code translated to machine language?

PC-relative addressing

```
0x40    loop:  add  $t1, $a0, $s0
0x44           lb   $t1, 0($t1)
0x48           add  $t2, $a1, $s0
0x4C           sb   $t1, 0($t2)
0x50           addi $s0, $s0, 1
0x54           bne  $t1, $0, loop
0x58           lw   $s0, 0($sp)
```

To calculate the `imm`, we take the PC value immediately after the branching instruction, here 0x58, we subtract it from the Branch Target Address (BTA), here 0x40, and divide by 4. Here, result will be -6.

Equivalently, we can just count the number of instructions from PC+4 to BTA, using a minus sign if BTA is above PC+4.

Note that, during the execution of `bne`, the value of the PC is 0x58, because we first increment the PC (add 4), and then execute an instruction.

PC-relative addressing

The assembly code

```
bne  $t1, $0, loop
```

is translated to the machine language I-Type instruction

`op`	`rs`	`rt`	`imm`
5	9	0	-6
000101	01001	00000	1111 1111 1111 1010
6 bits	5 bits	5 bits	16 bits

Notice that the 16-bits of the `imm`, field are used to represent integers from −32,768 to 32,767 rather than from 0 to 65,535.

MIPS addressing modes

$$\left.\begin{array}{l}\text{Register Only}\\[1em] \text{Immediate}\\[1em] \text{Base Addressing}\end{array}\right\}\ \text{Reading and writing operands}$$

$$\left.\begin{array}{l}\text{PC-Relative}\\[1em] \text{Pseudo-direct}\end{array}\right\}\ \text{Writing the Program Counter}$$

Pseudo-direct addressing

In direct addressing, an address is specified in the instruction.

MIPS does not support direct addressing, which would need 32-bit for the address and 6-bits for the opcode, while the instruction has 32 bits only.

MIPS uses pseudo-direct addressing in J-Type instructions, calculating the new value of the PC, called Jump Target Address (JTA), as follows:

- The two least significant bits are left to 0 (instructions are word aligned and word addresses are multiples of 4).
- The next 26 bits are taken from the `addr` field of the J-Type instruction.
- The four most significant bits are again left to 0 (recall the slide on the text segment of the memory map).

Pseudo-direct addressing

```
0x0040005C           j  sum
...
...
0x004000A0   sum: add $v0, $a0, $a1
```

The 26-bit `addr` field of the `j` instruction is computed from the JTA, here the address of the instruction labelled `sum`, which is

0x004000A0

In binary, that address is

0000 0000 0100 0000 0000 0000 1010 0000

and we remove the first four and the last two bits, getting

`addr` = 0000 0100 0000 0000 0000 1010 00

Pseudo-direct addressing

```
0x0040005C        j sum
...
...
0x004000A0   sum: add $v0, $a0, $a1
```

The assembly instruction

`j   sum`

is translated to the machine language J-Type instruction

op	addr
000101	00000100000000000000101000
6 bits	26 bits

Pseudo-direct addressing

Conversely, when that `j` instruction is executed, the new PC value will be computed from its 26-bit `addr` field, here

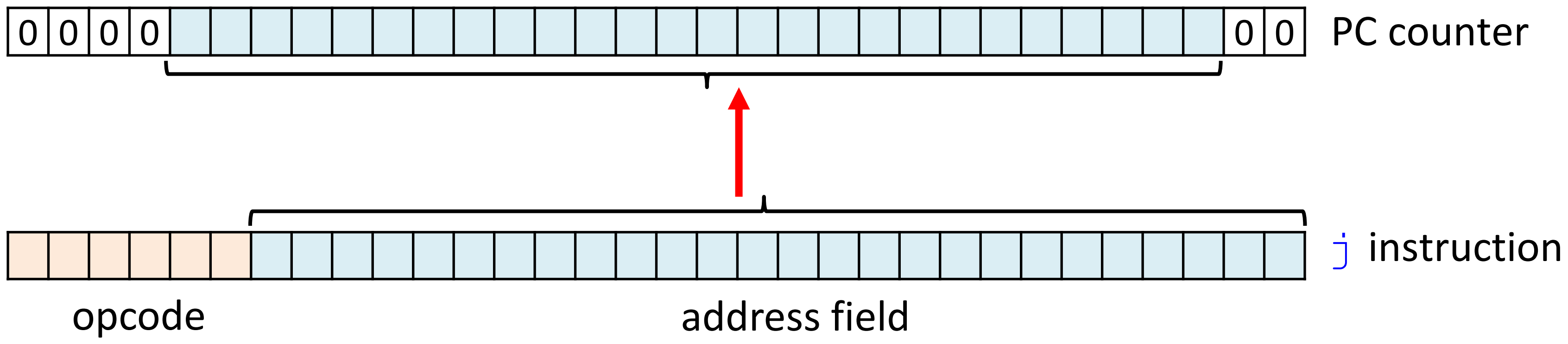
0000 0100 0000 0000 0000 1010 00

by appending four 0's as its most significant bits, and two 0's as its least significant bits, getting

0000 0000 0100 0000 0000 0000 1010 0000

Pseudo-direct addressing

As a hardware implementation on the MIPS microprocessor, the `j` instruction just copies the 26 bits of its `addr` field, over to 26 bits of the PC.


0 0 0 0
0 0
PC counter
j instruction
opcode
address field