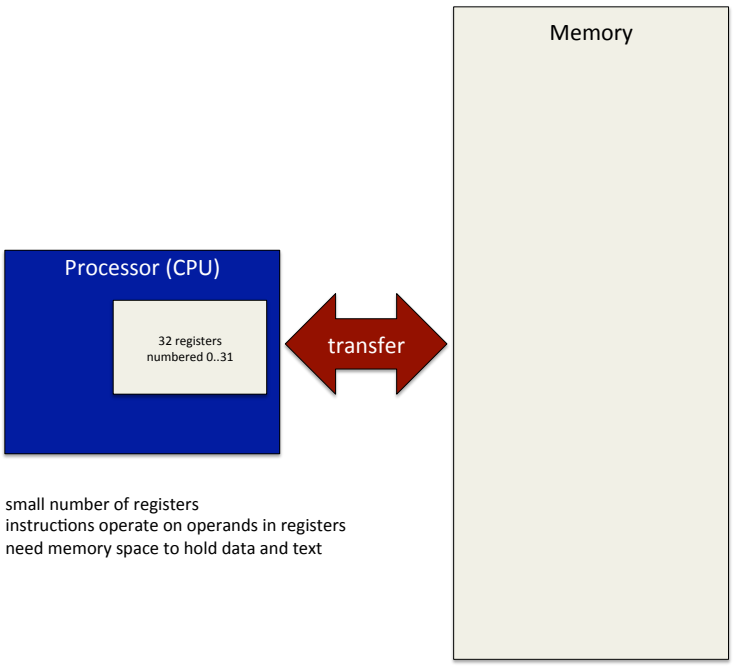
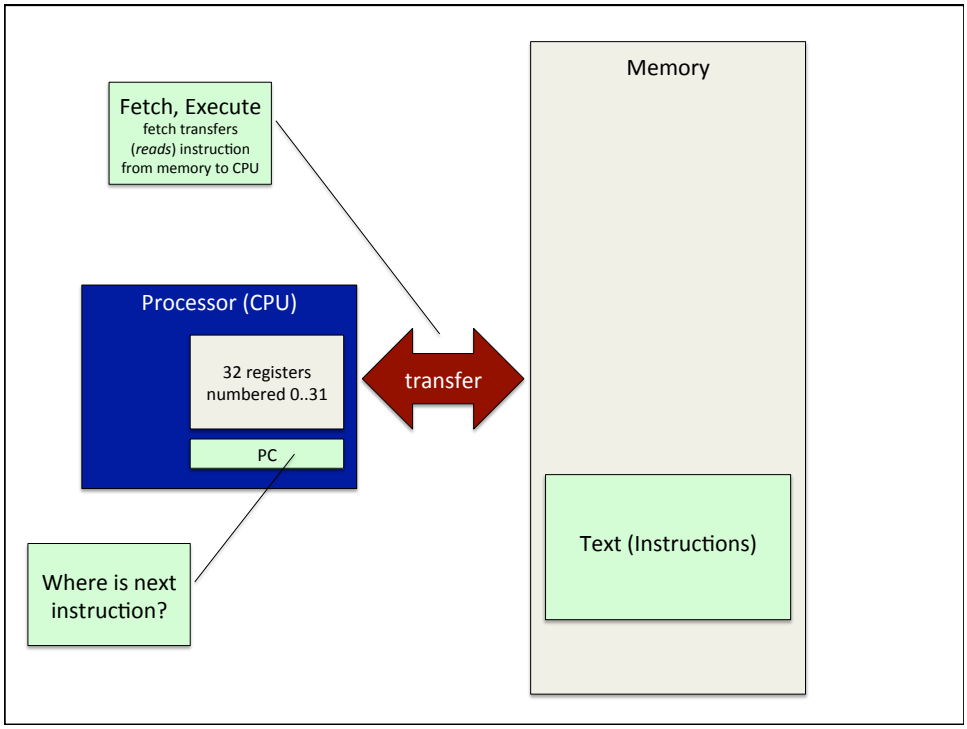
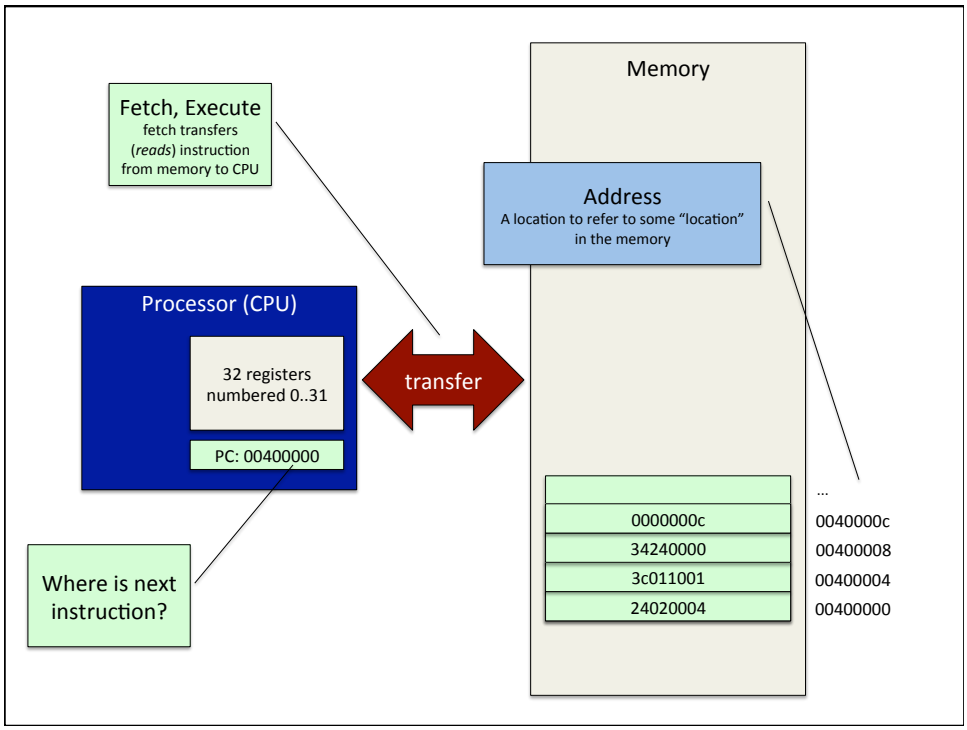
Memory

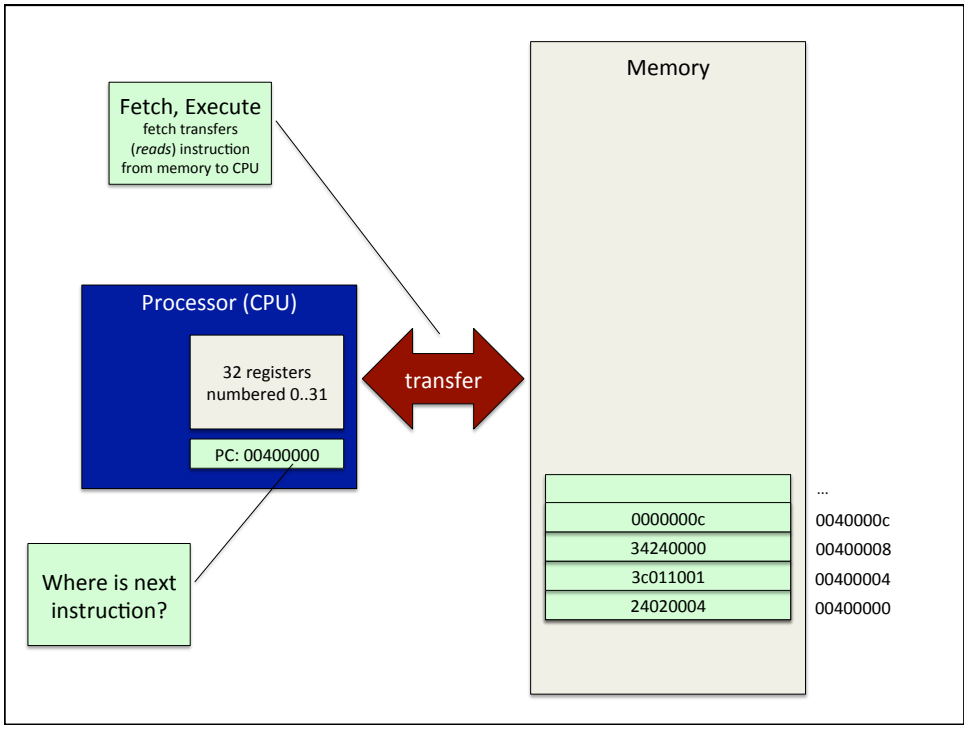
CS 447, Spring 2016

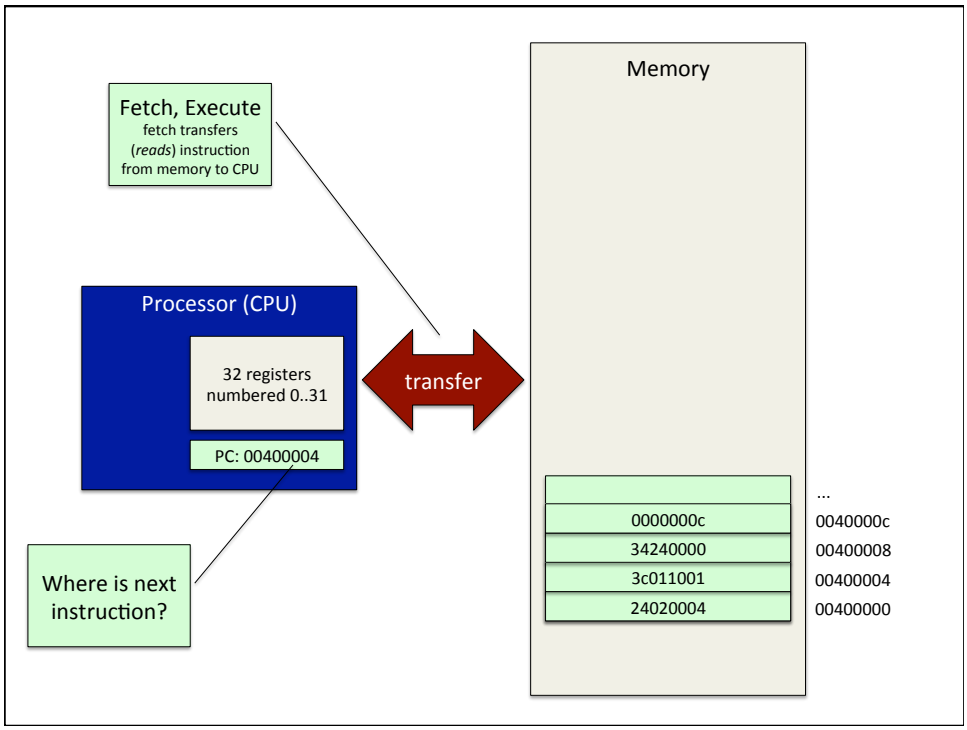


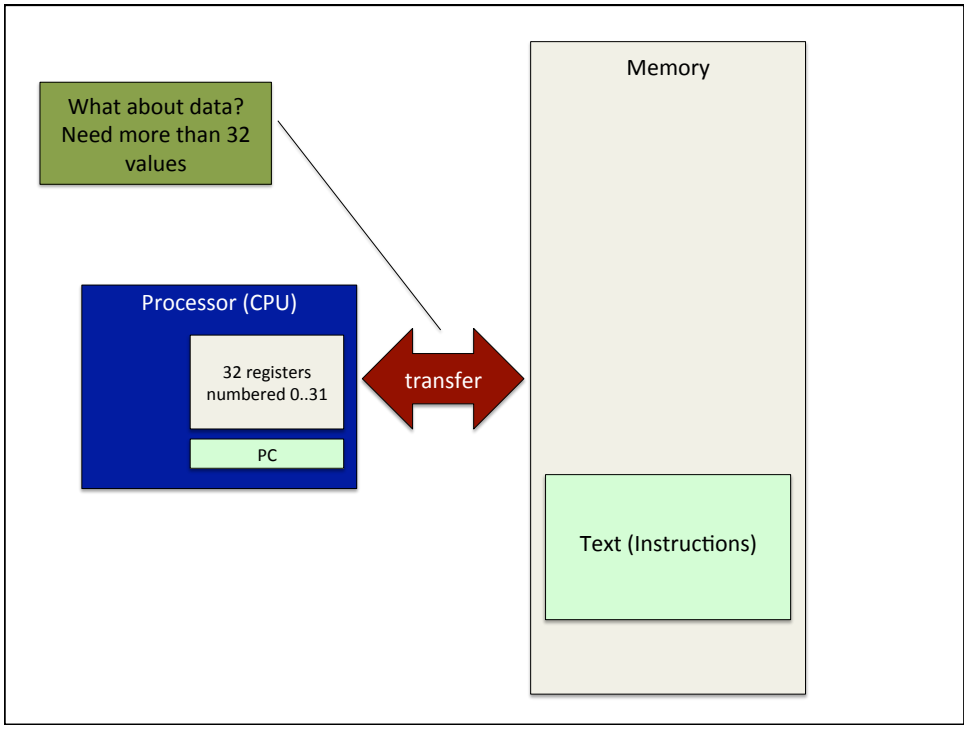
small number of registers
instructions operate on operands in registers
need memory space to hold data and text

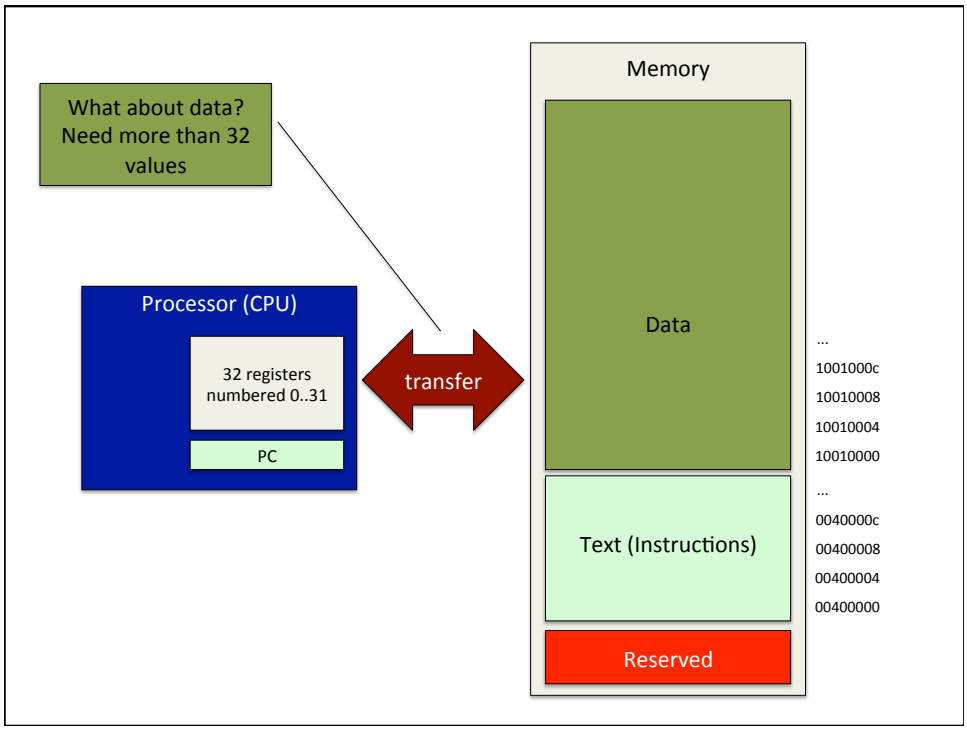

Memory
Fetch, Execute
fetch transfers
(*reads*) instruction
from memory to CPU
Processor (CPU)
32 registers
numbered 0..31
PC
transfer
Where is next
instruction?
Text (Instructions)

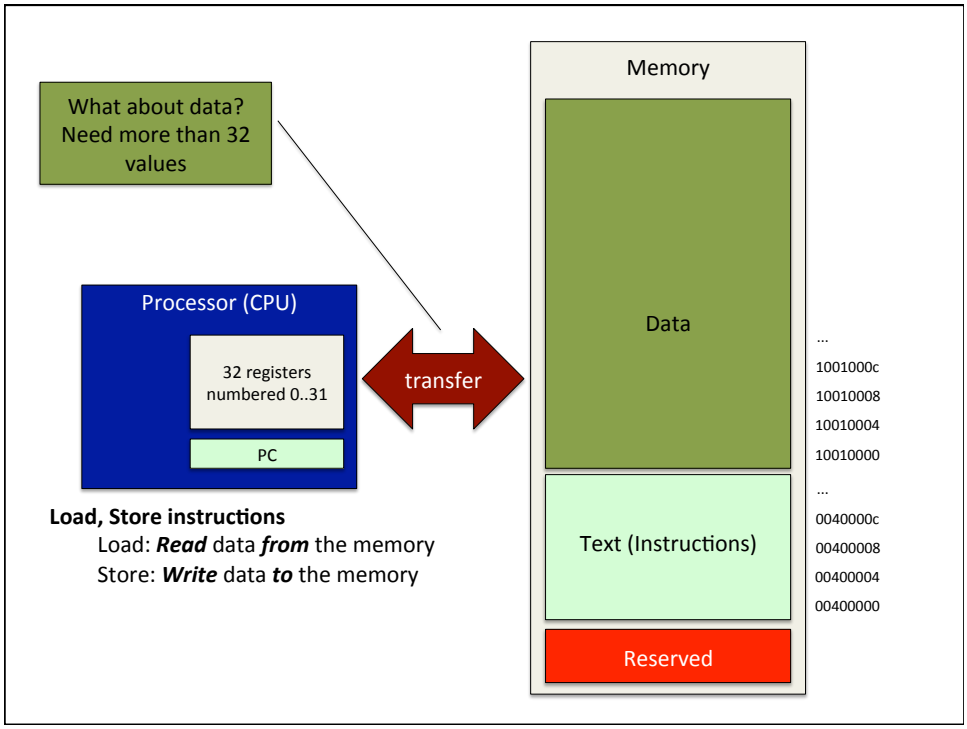

Memory
Fetch, Execute
fetch transfers
(*reads*) instruction
from memory to CPU
Address
A location to refer to some "location"
in the memory
Processor (CPU)
32 registers
numbered 0..31
PC: 00400000
transfer
Where is next
instruction?
0000000c
34240000
3c011001
24020004
...
0040000c
00400008
00400004
00400000


Memory
Fetch, Execute
fetch transfers (reads) instruction from memory to CPU
Processor (CPU)
32 registers numbered 0..31
PC: 00400000
transfer
Where is next instruction?
...
0000000c
34240000
3c011001
24020004
0040000c
00400008
00400004
00400000


Memory
Fetch, Execute
fetch transfers (reads) instruction from memory to CPU
Processor (CPU)
32 registers numbered 0..31
PC: 00400004
transfer
Where is next instruction?
...
0000000c
34240000
3c011001
24020004
0040000c
00400008
00400004
00400000


What about data? Need more than 32 values
Memory
Processor (CPU)
32 registers numbered 0..31
PC
transfer
Text (Instructions)

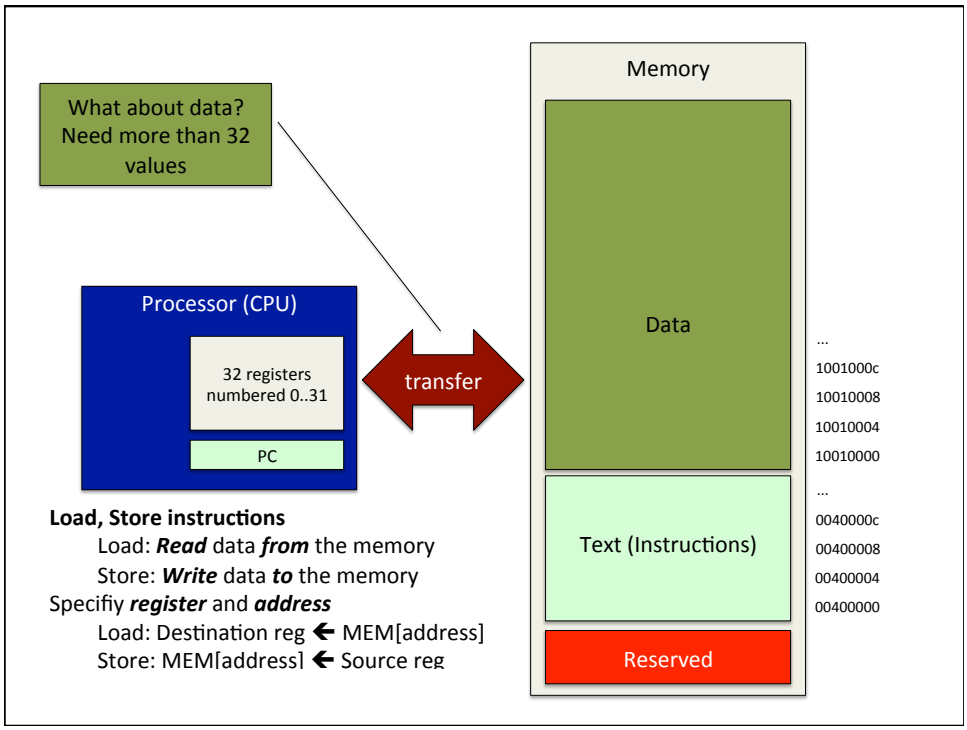

What about data? Need more than 32 values
Memory
Processor (CPU)
32 registers numbered 0..31
PC
transfer
Data
Text (Instructions)
Reserved
...
1001000c
10010008
10010004
10010000
...
0040000c
00400008
00400004
00400000



Load, Store instructions

Load: ***Read*** data ***from*** the memory

Store: ***Write*** data ***to*** the memory



Load, Store instructions

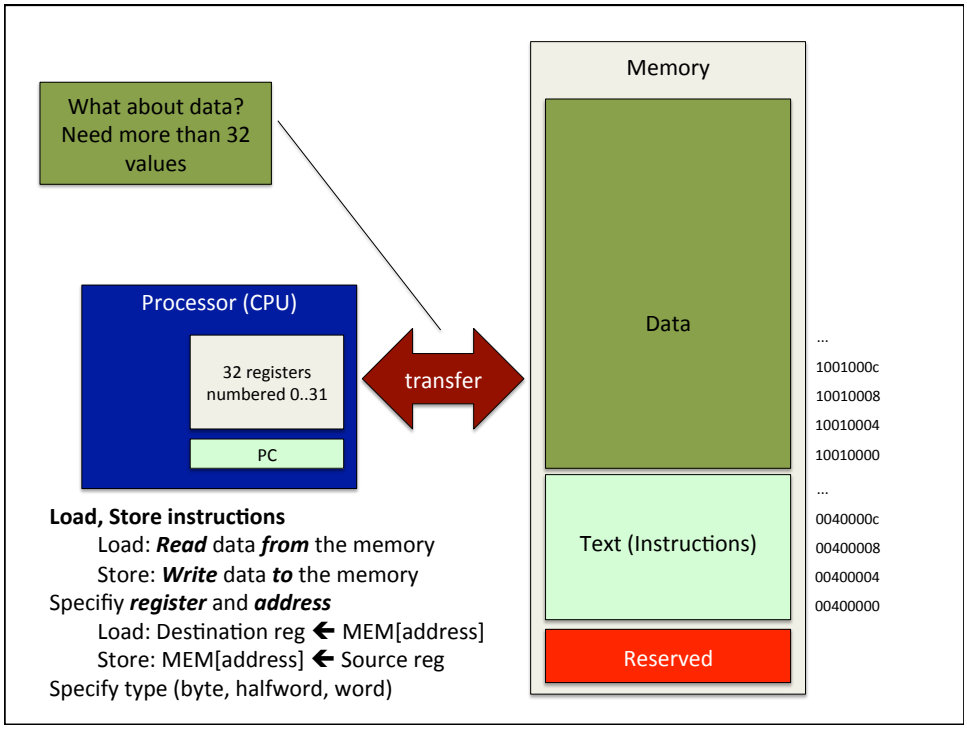
Load: ***Read*** data ***from*** the memory

Store: ***Write*** data ***to*** the memory

Specifiy ***register*** and ***address***

Load: Destination reg ← MEM[address]

Store: MEM[address] ← Source reg



Load, Store instructions

Load: ***Read*** data ***from*** the memory

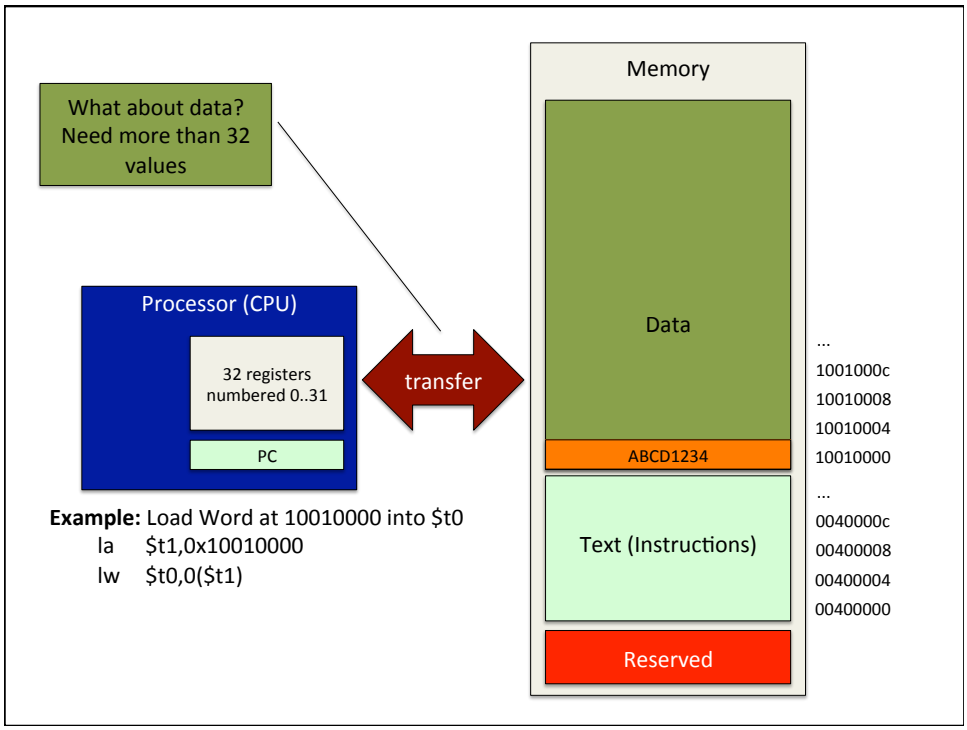
Store: ***Write*** data ***to*** the memory

Specifiy ***register*** and ***address***

Load: Destination reg ← MEM[address]

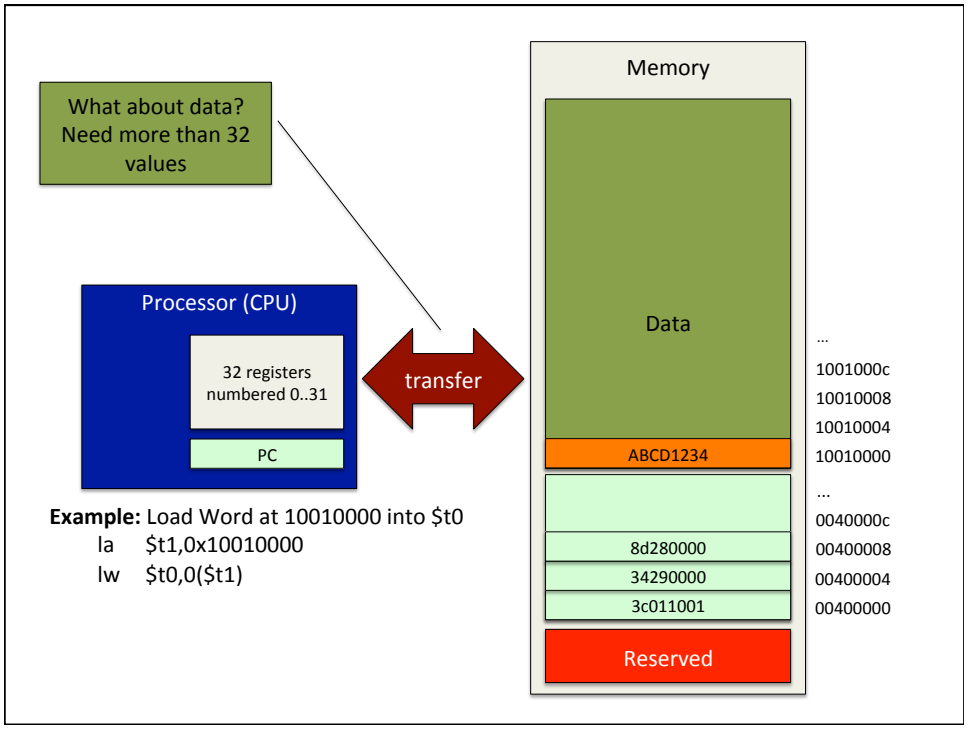
Store: MEM[address] ← Source reg

Specify type (byte, halfword, word)



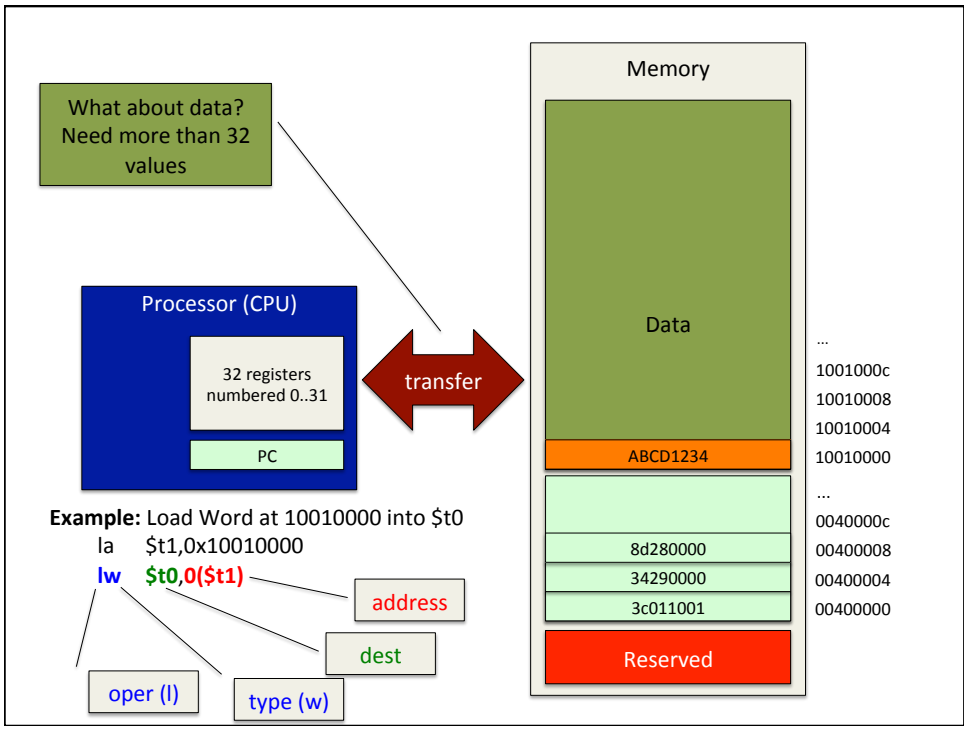
Example: Load Word at 10010000 into $t0

```
la   $t1,0x10010000
lw   $t0,0($t1)
```



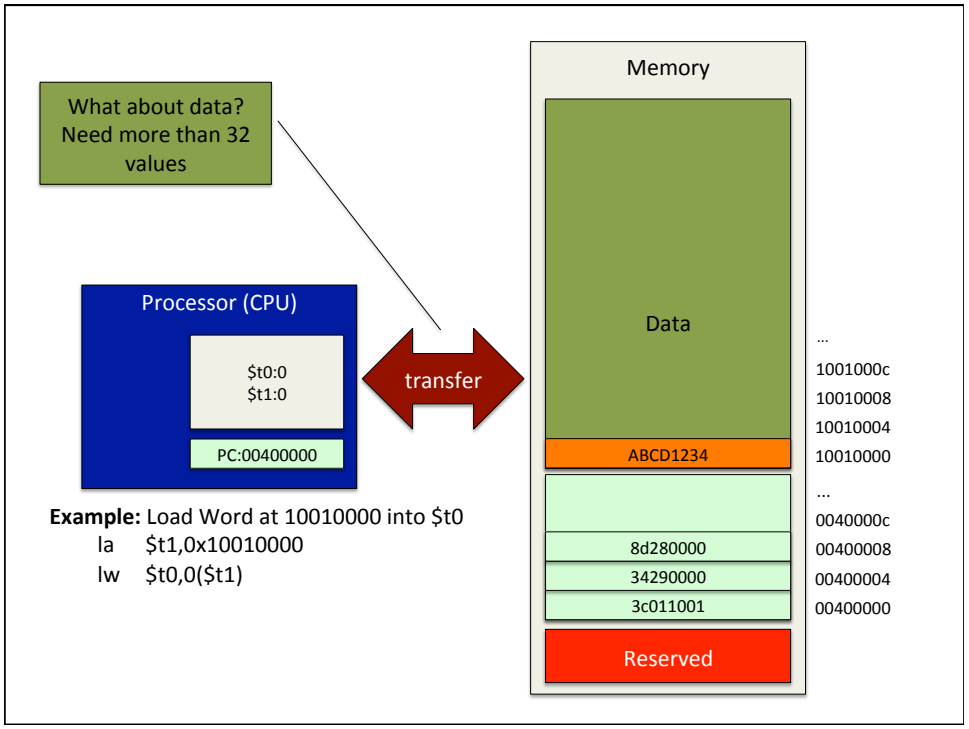
Example: Load Word at 10010000 into $t0

```
la   $t1,0x10010000
lw   $t0,0($t1)
```



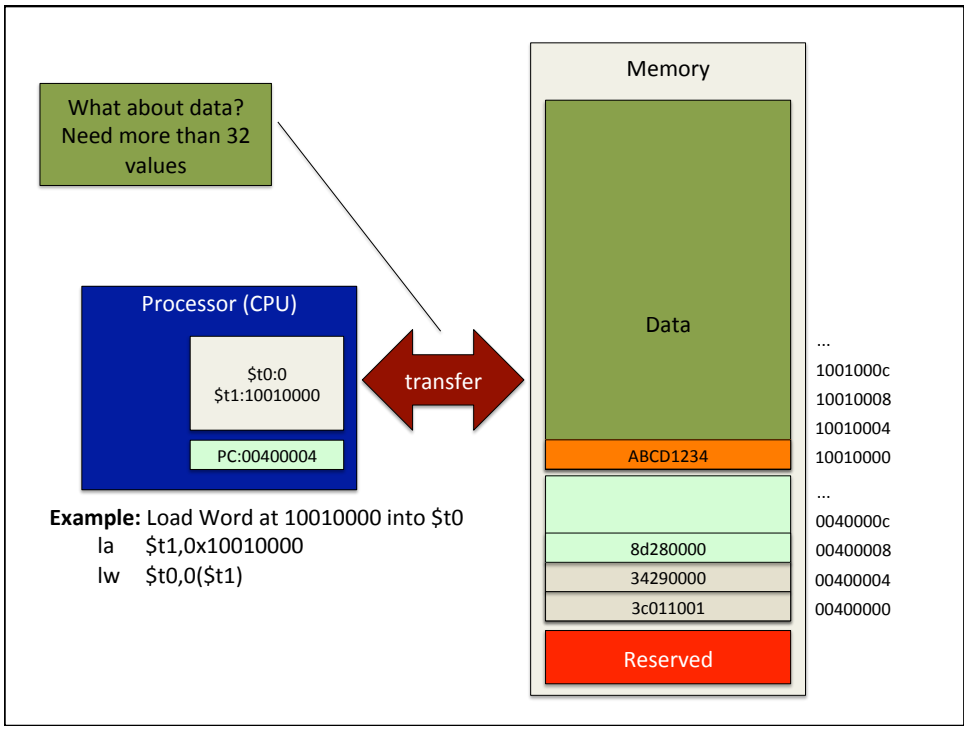
Example: Load Word at 10010000 into $t0

```
la   $t1,0x10010000
lw   $t0,0($t1)
```



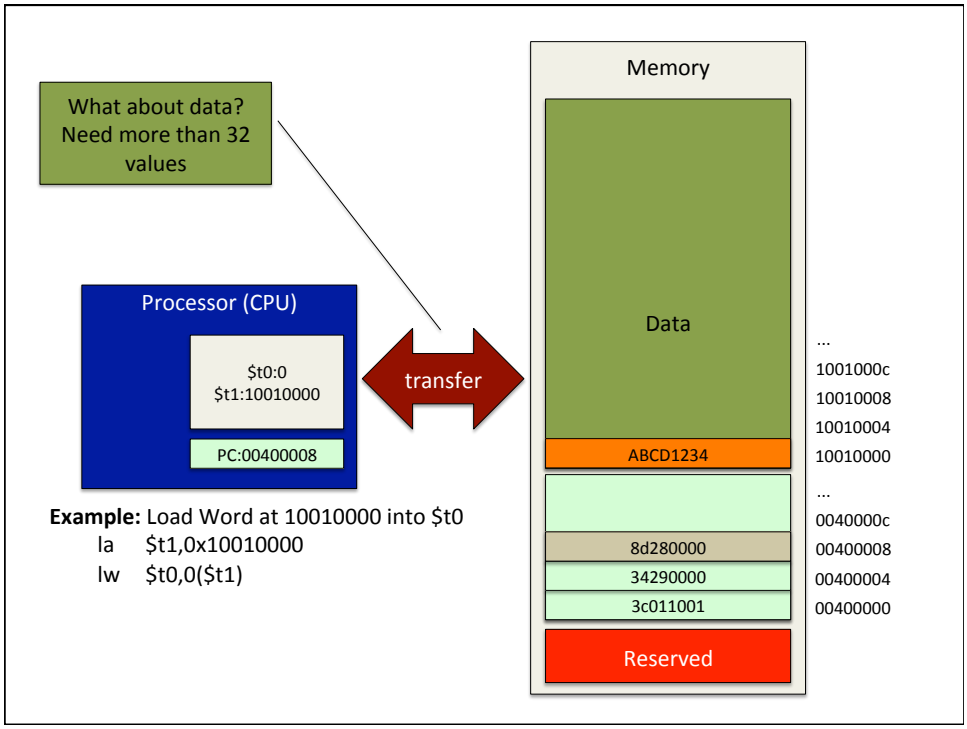
Example: Load Word at 10010000 into $t0

```
la   $t1,0x10010000
lw   $t0,0($t1)
```



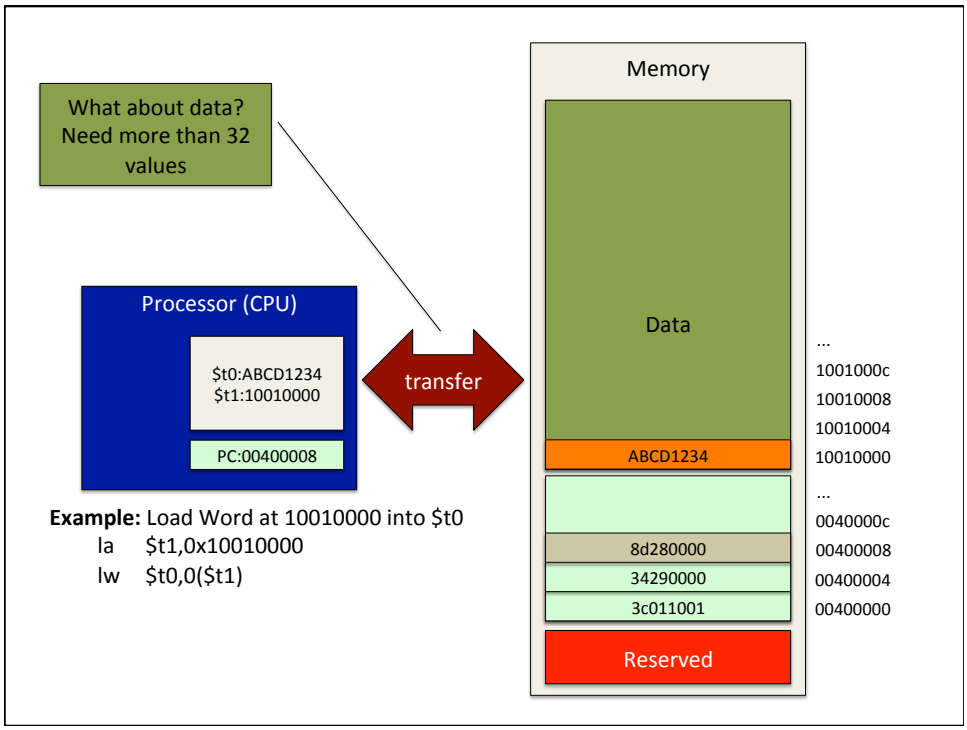
Example: Load Word at 10010000 into $t0

```
la   $t1,0x10010000
lw   $t0,0($t1)
```



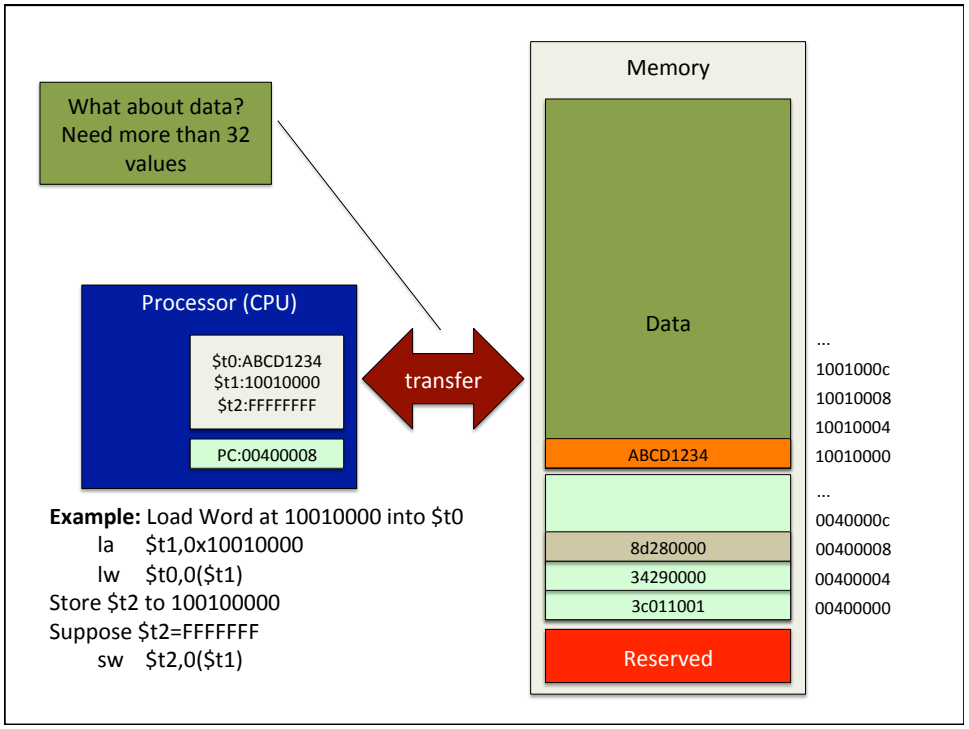
Example: Load Word at 10010000 into $t0

```
la   $t1,0x10010000
lw   $t0,0($t1)
```



Example: Load Word at 10010000 into $t0

```
la   $t1,0x10010000
lw   $t0,0($t1)
```

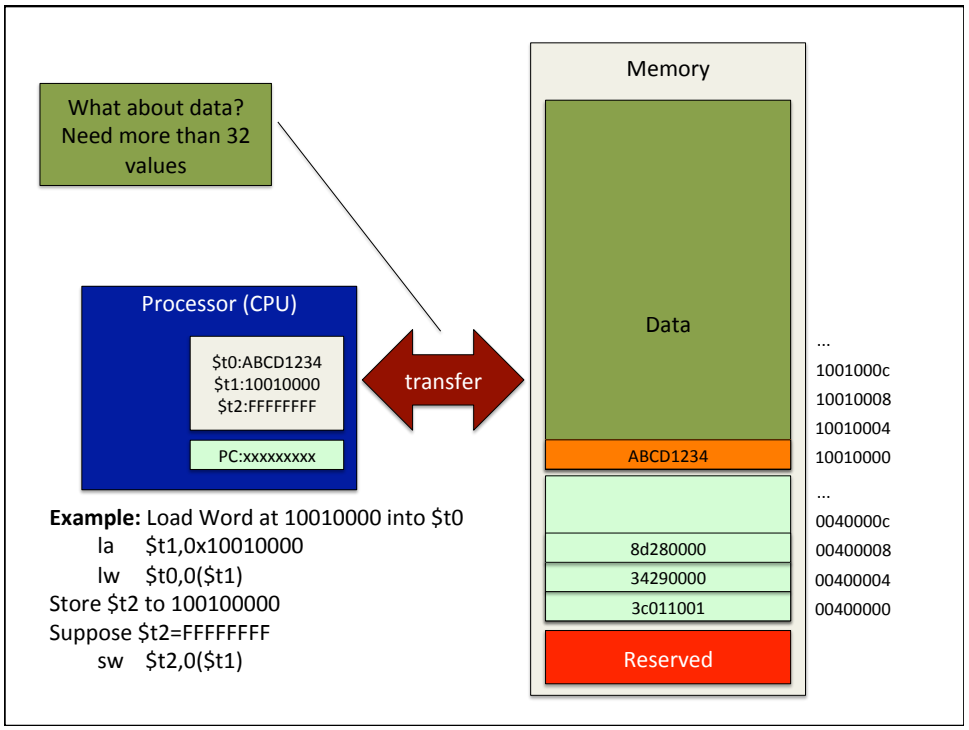


Example: Load Word at 10010000 into $t0

```
    la   $t1,0x10010000
    lw   $t0,0($t1)
```

Store $t2 to 100100000
Suppose $t2=FFFFFFF

```
    sw   $t2,0($t1)
```

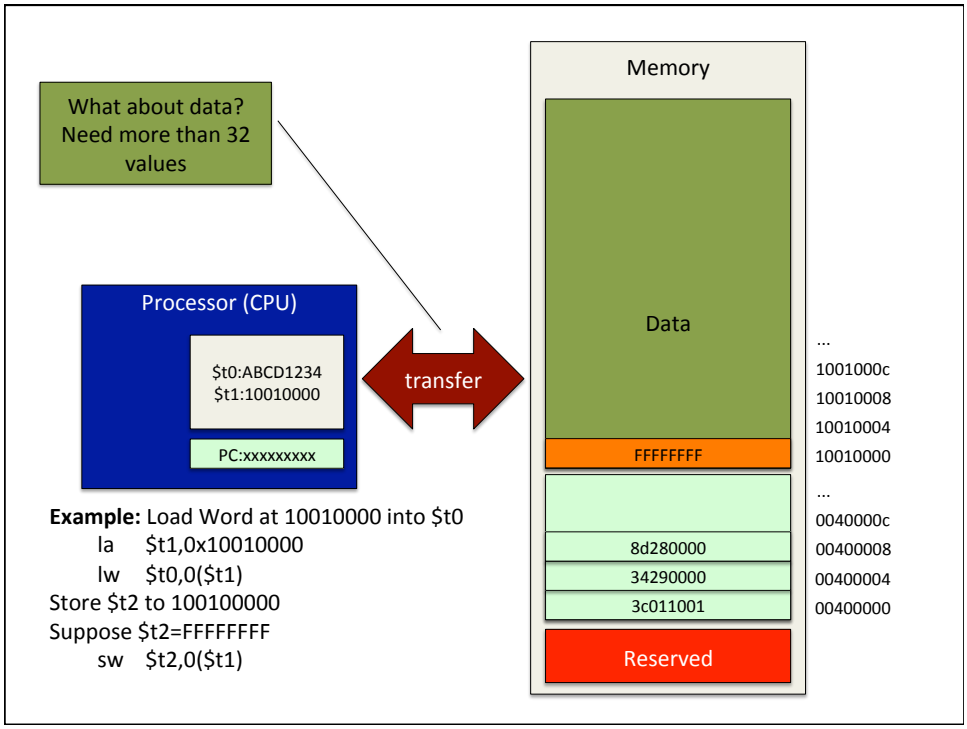


Example: Load Word at 10010000 into $t0

```
    la   $t1,0x10010000
    lw   $t0,0($t1)
```

Store $t2 to 100100000
Suppose $t2=FFFFFFFF

```
    sw   $t2,0($t1)
```

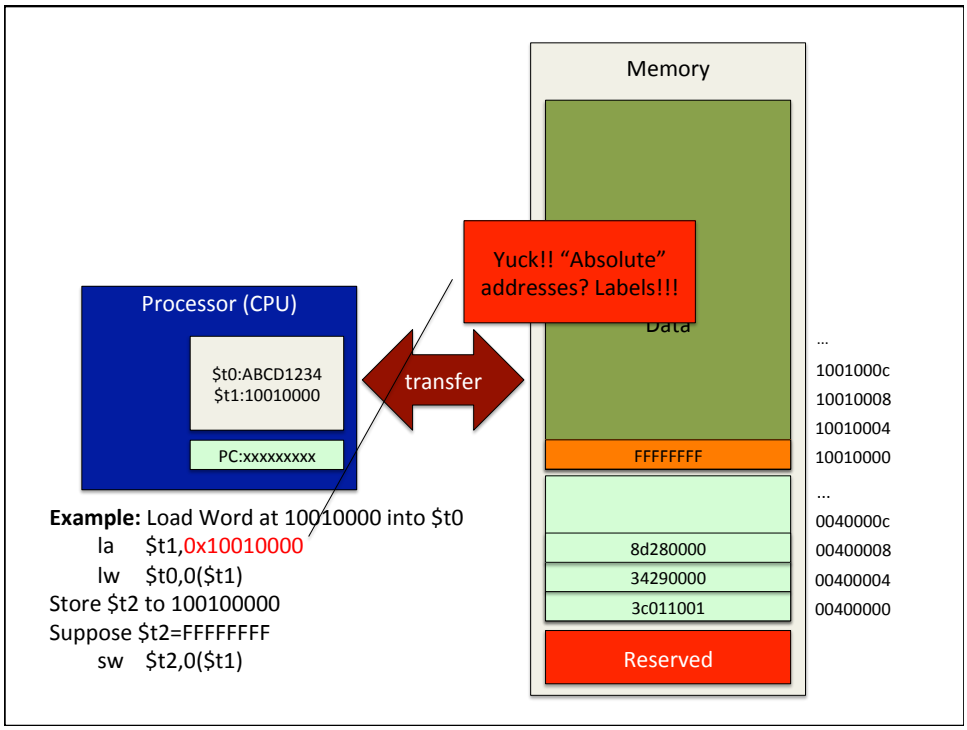


Example: Load Word at 10010000 into $t0

```
    la   $t1,0x10010000
    lw   $t0,0($t1)
```

Store $t2 to 100100000
Suppose $t2=FFFFFFFF

```
    sw   $t2,0($t1)
```

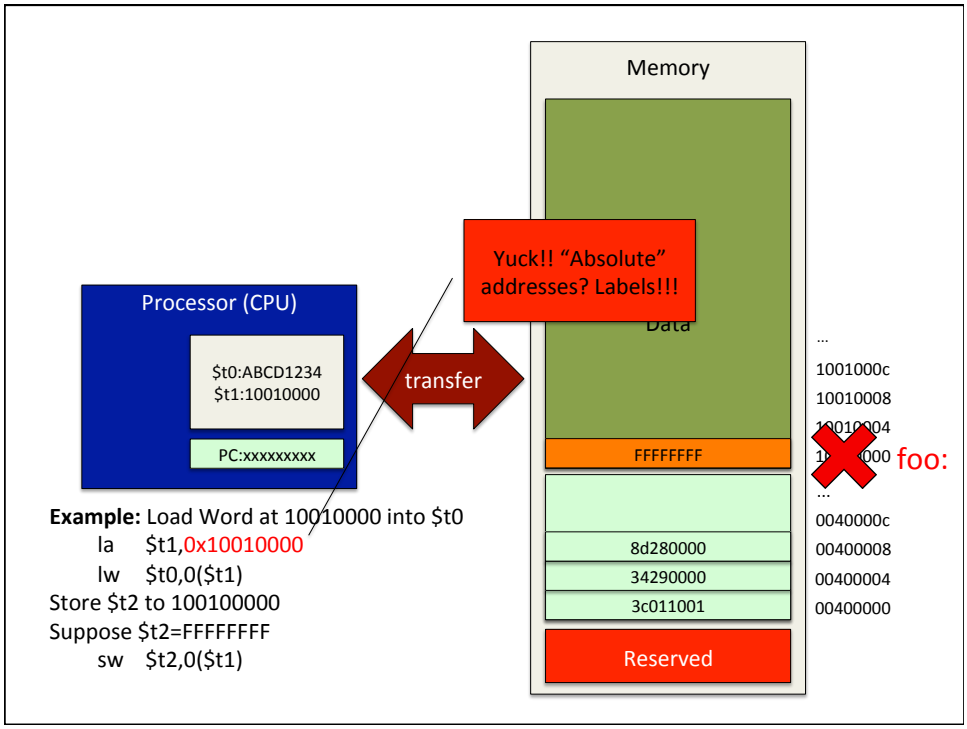


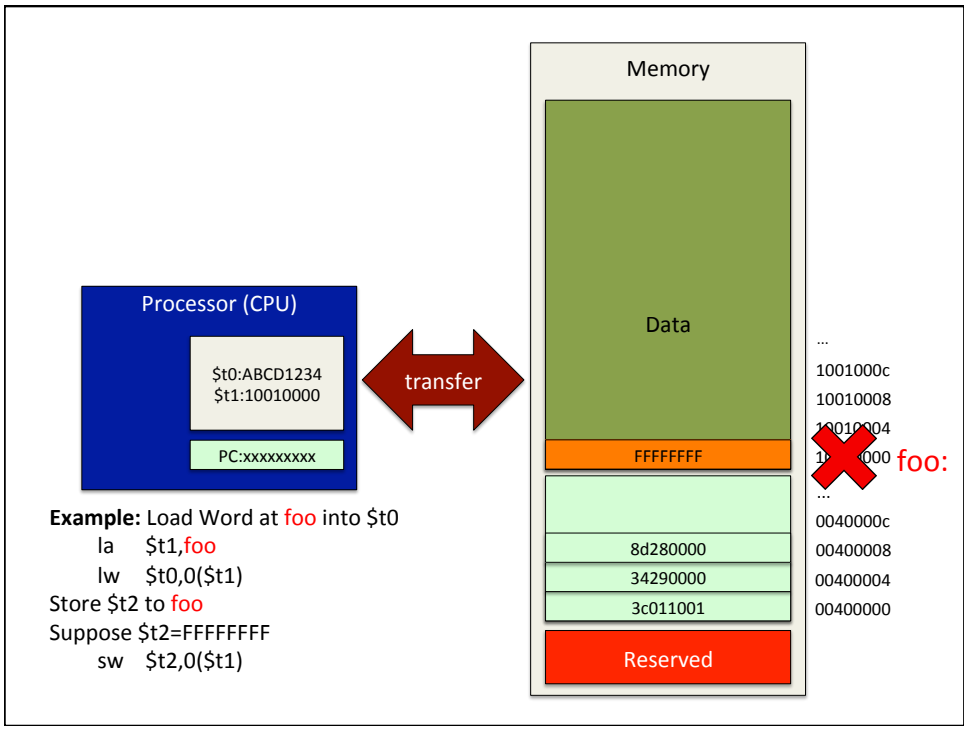
Example: Load Word at 10010000 into $t0

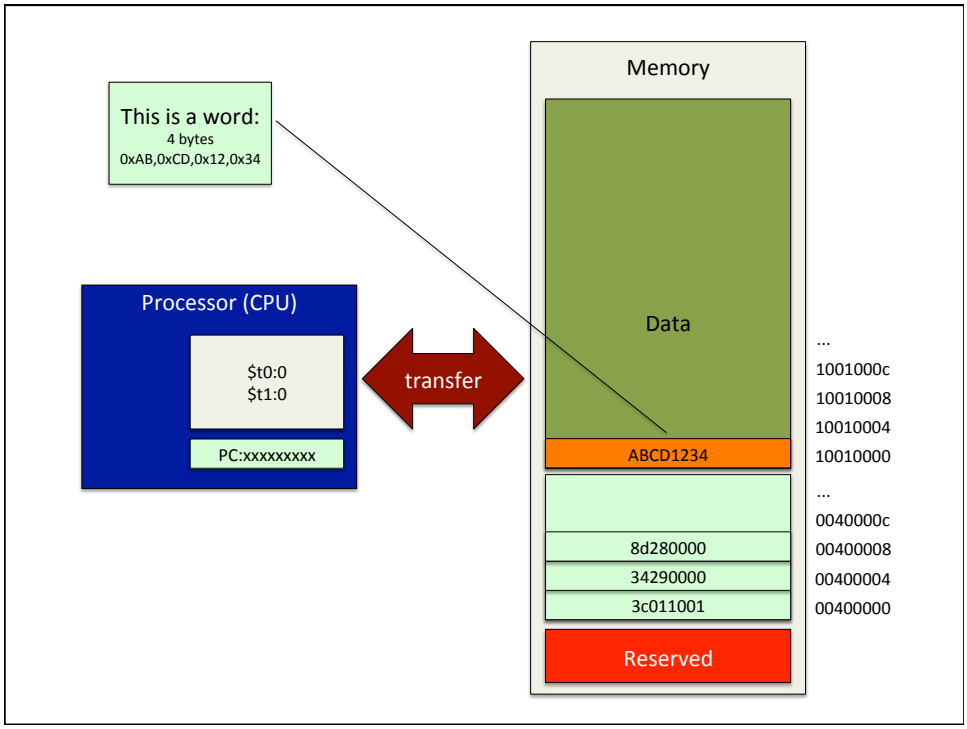
```
    la   $t1,0x10010000
    lw   $t0,0($t1)
```

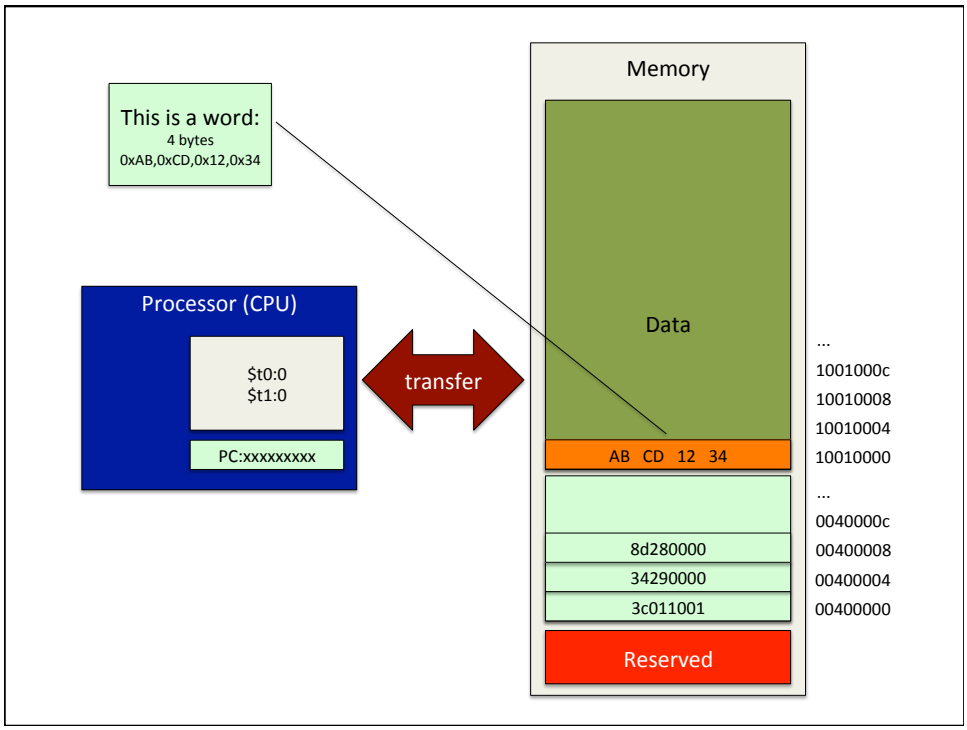
Store $t2 to 100100000
Suppose $t2=FFFFFFFF

```
    sw   $t2,0($t1)
```


Memory
Yuck!! "Absolute" addresses? Labels!!!
Processor (CPU)
$t0:ABCD1234
$t1:10010000
PC:xxxxxxxxx
transfer
Data
FFFFFFFF
8d280000
34290000
3c011001
Reserved
...
1001000c
10010008
10010004
foo:
...
0040000c
00400008
00400004
00400000
Example: Load Word at 10010000 into $t0
la $t1,0x10010000
lw $t0,0($t1)
Store $t2 to 100100000
Suppose $t2=FFFFFFFF
sw $t2,0($t1)


Memory
Processor (CPU)
$t0:ABCD1234
$t1:10010000
PC:xxxxxxxxx
transfer
Data
FFFFFFFF
8d280000
34290000
3c011001
Reserved
...
1001000c
10010008
10010004
foo:
...
0040000c
00400008
00400004
00400000
Example: Load Word at foo into $t0
la $t1,foo
lw $t0,0($t1)
Store $t2 to foo
Suppose $t2=FFFFFFFF
sw $t2,0($t1)


This is a word:
4 bytes
0xAB,0xCD,0x12,0x34
Memory
Data
Processor (CPU)
$t0:0
$t1:0
PC:xxxxxxxxx
transfer
ABCD1234
8d280000
34290000
3c011001
Reserved
...
1001000c
10010008
10010004
10010000
...
0040000c
00400008
00400004
00400000

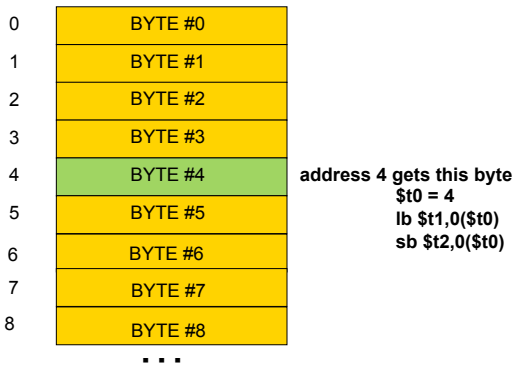

This is a word:
4 bytes
0xAB,0xCD,0x12,0x34
Memory
Data
Processor (CPU)
$t0:0
$t1:0
PC:xxxxxxxxx
transfer
AB CD 12 34
8d280000
34290000
3c011001
Reserved
...
1001000c
10010008
10010004
10010000
...
0040000c
00400008
00400004
00400000

Memory transfer instructions

- How to get values to/from memory?
  - Also called *memory access* instructions
- Only two types of instructions
  - Load: move data from memory to register ("load the register")
    - e.g., lw $s5, 4($t6) # $s5 ⇐ memory[$t6 + 4]
  - Store: move data from register to memory ("save the register")
    - e.g., sw $s7, 16($t3) # memory[$t3+16] ⇐ $s7
- In MIPS (32-bit architecture) there are memory transfer instructions for
  - 32-bit word: "int" type in C (lw, sw)
  - 16-bit half-word: "short" type in C (lh, sh; also unsigned lhu)
  - 8-bit byte: "char" type in C (lb, sb; also unsigned lbu)

Memory view

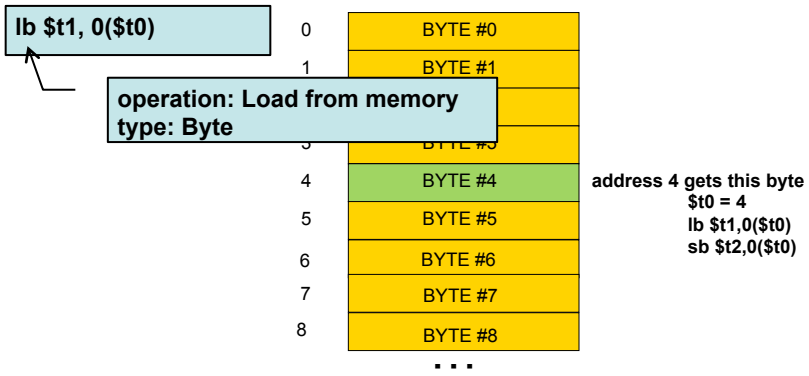
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item ("byte address")
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

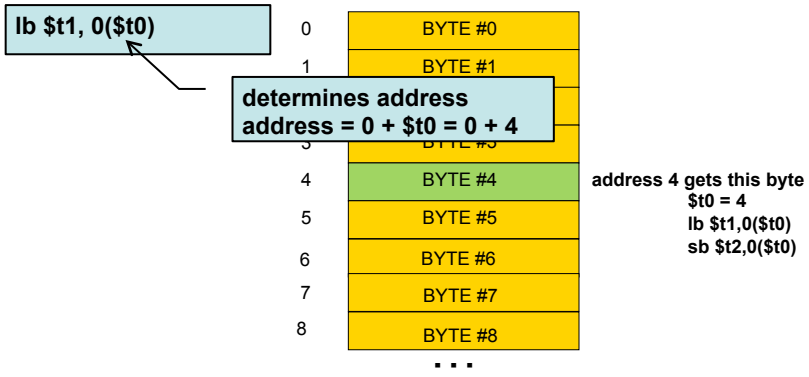
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item ("byte address")
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

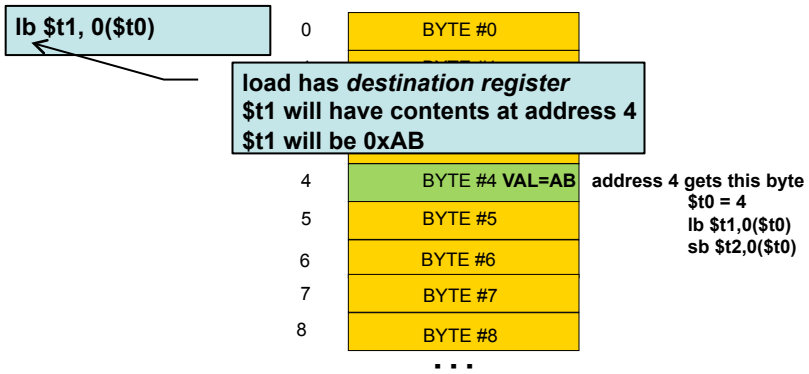
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item ("byte address")
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

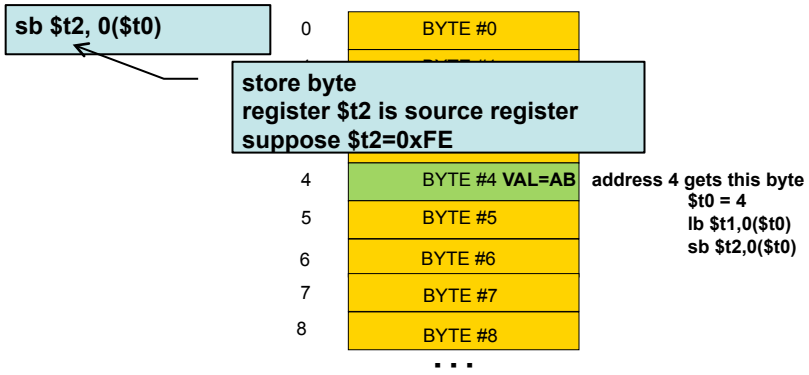
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

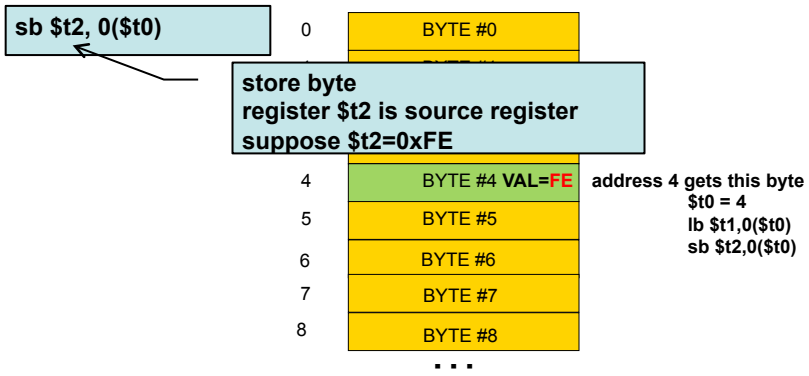
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

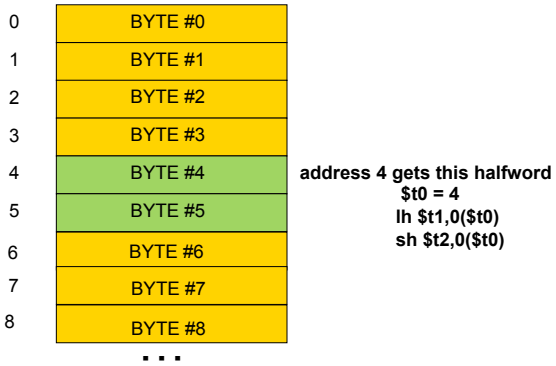
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

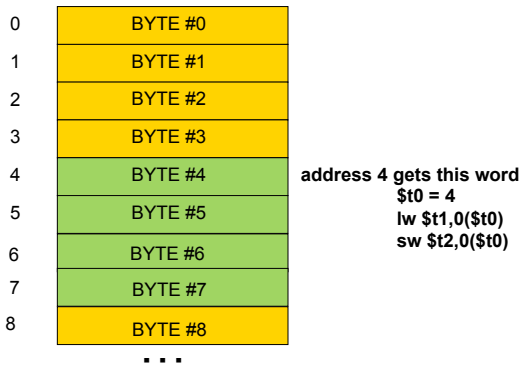
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item ("byte address")
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Effective Address calculation

- ***Effective memory address*** specified as *immediate($register)*
  - Register to keep the ***base address***
  - Immediate to determine an ***offset*** from the base address
  - Thus, address is *contents* of *register* + *immediate*
  - The offset can be positive or negative, 16-bit value (uses I-format)
- Suppose base register $t1=64, then:

```
lw    $t0, 12($t1)      address = 64 + 12 = 76
lw    $t0, -12($t1)     address = 64 - 12 = 52
```

- MIPS uses this simple address calculation; other architectures such as PowerPC and x86 support different methods

Hint on addresses (`la` - load address)

- Often, you need to reference a particular variable.

```
.data
var:   .word        0x1000, 0x2000
```

assembler directive to declare data (word)

- How to reference **`var`**?

```
la   $t0,var
lw   $t1,0($t0)
lw   $t2,4($t0)
```

puts the address of variable "var" into $t0

value at the address in $t0 is loaded ino $t1

- **`la`** is a "pseudo-instruction". It is turned into a sequence to put a large address constant into $t0.

```
lui   $at,upperbitsofaddres
ori   $t0,$1,lowerbitsofaddress
```


Let's try an in-class exercise together!

- Create a word (integer) variable "myVar"
- Give the variable the value 20
- Print the value to the console (Run I/O window)
- Terminate the program
- Extension: Add 10 to the value, store it to myVar, print it

- To do this, we'll need to use:
  - Data segment declaration with a word variable type
  - Instruction segment declaration
  - Load word instruction
  - Syscall instruction
  - Assorted la and li instructions

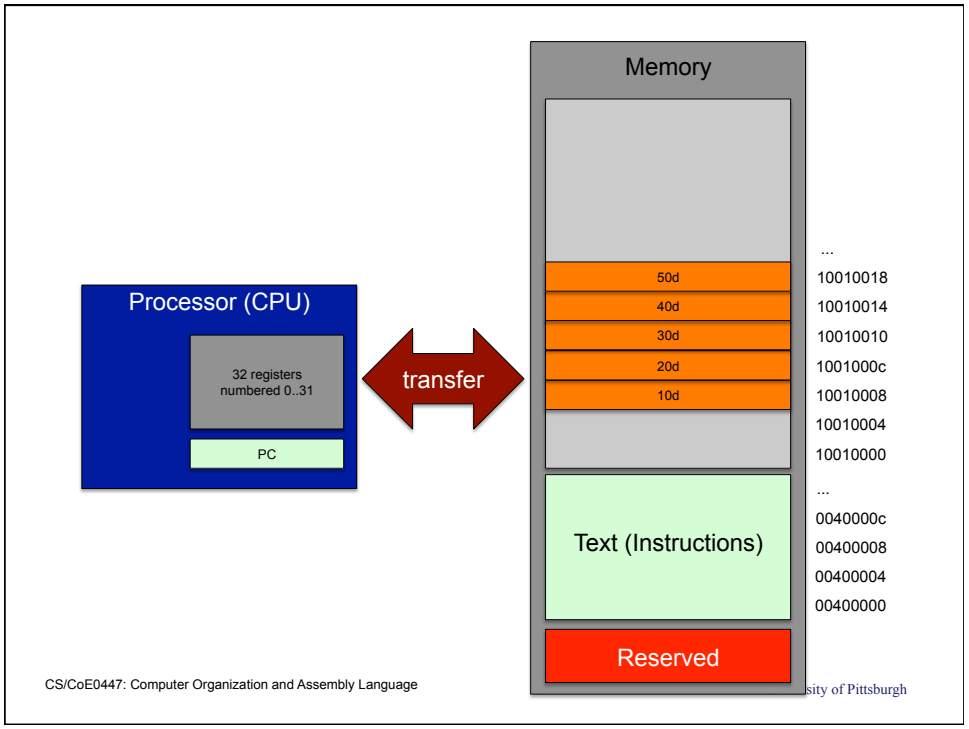
In-class Example

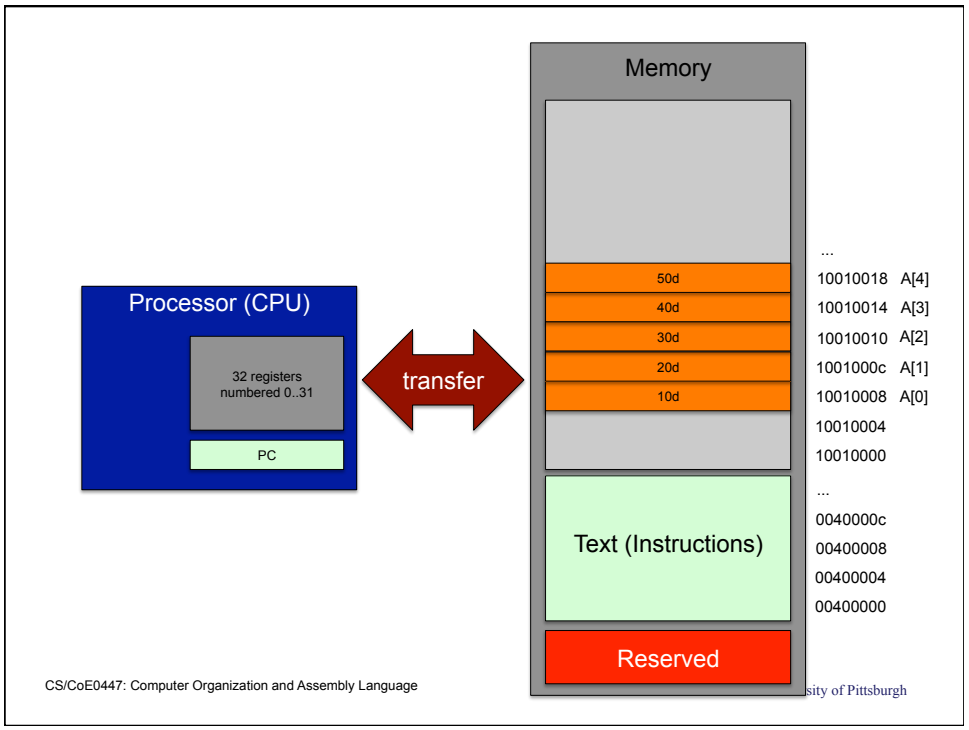
- set myVar=20, print myVar, terminate

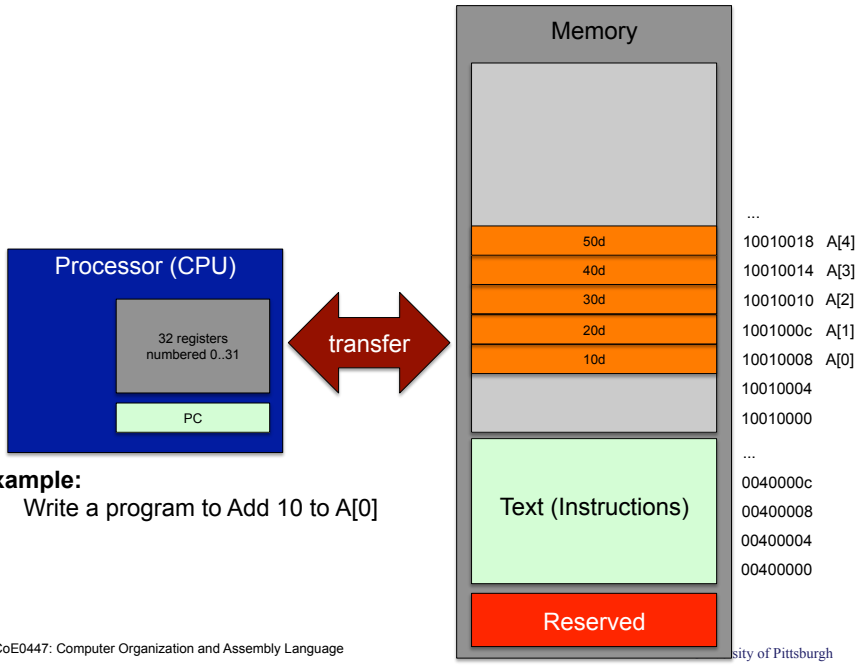
```
        .data
myVar:  .word   20
        .text
        la      $t0,myVar       # address of "myVar"
        lw      $a0,0($t0)      # load value into $a0
        li      $v0,1           # print integer service
        syscall                 # call operating system
        li      $v0,10          # terminate service
        syscall                 # call operating system
```


Another example

- What about an ***array***?
  - A sequence of data elements
  - Each element can be accessed ("references") by an index
  - 10 elements: Index 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- E.g., declare a C array of integers with 5 numbers
  - int A[5];
  - 5elements: A[0], A[1], A[2], A[3], A[4]
- This is data. How is it represented?
  - 5 integers in memory; an integer is a word in MIPS (4 bytes)
  - Element 0: at lowest address
  - Element 4: at highest address


Memory
Processor (CPU)
32 registers
numbered 0..31
PC
transfer
50d
40d
30d
20d
10d
Text (Instructions)
Reserved
...
10010018
10010014
10010010
1001000c
10010008
10010004
10010000
...
0040000c
00400008
00400004
00400000
CS/CoE0447: Computer Organization and Assembly Language
sity of Pittsburgh


Memory
Processor (CPU)
32 registers
numbered 0..31
PC
transfer
50d
40d
30d
20d
10d
Text (Instructions)
Reserved
...
10010018 A[4]
10010014 A[3]
10010010 A[2]
1001000c A[1]
10010008 A[0]
10010004
10010000
...
0040000c
00400008
00400004
00400000
CS/CoE0447: Computer Organization and Assembly Language
sity of Pittsburgh



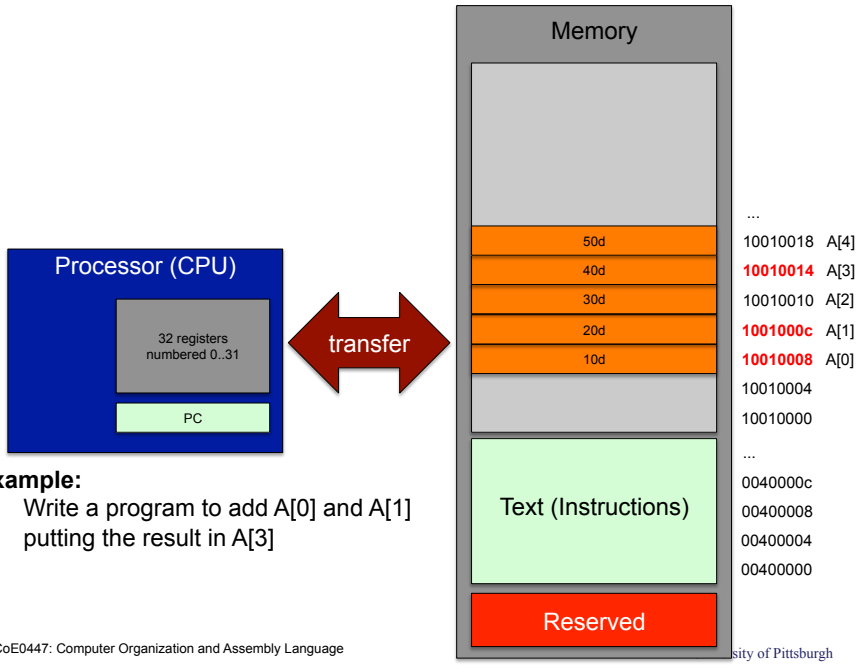
Example:
Write a program to Add 10 to A[0]


```
# Example: add 10 to A[0]
    .data
    array_A:    .word    10,20,30,40,50
    .text
    la   $t0,array_A
    lw   $t1,0($t0)
    add  $t1,$t1,10
    sw   $t1,0($t0)
```

Use label to reference array

load A[0]'s value into register

store new A[0]'s value in memory



Example:

Write a program to add A[0] and A[1] putting the result in A[3]


```
# Example:   Write a program to add A[0] and A[1]
#                putting the result in A[3]
        .data
        array_A:        .word     10,20,30,40,50
        .text
        la    $t0,array_A
        lw    $t1,0($t0)
        lw    $t2,4($t0)
        add $t1,$t1,$t2
        sw    $t1,12($t0)
```

- Use label to reference array
- load A[0]'s value into register
- load A[1]'s value into register
  $t0+4 is next array element
- Why $t0+12????
- A[3] is 12 bytes from A[0]
  3 integers * 4 bytes each = 12

Memory Organization

32-bit byte address:

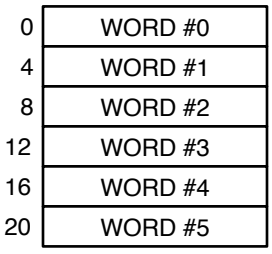
- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, …, 2^{32}-4

Words are aligned

- 2 least significant bits (LSBs) of an address are 0s

Half words are aligned

- LSB of an address is 0



Addressing within a word:

- **Which byte appears first and which byte last?**
- Big-endian vs. little-endian
  - Little end (LSB) comes first (at low address)
  - Big end (MSB) comes first (at low address)

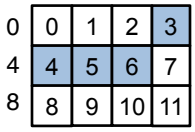
Alignment

A misaligned access

- Assume $t0=0, then `lw $s4, 3($t0)`

How do we define a word at an address?

- Data in byte 0, 1, 2, 3
  - If you meant this, use the address 0, not 3.
- Data in byte 3, 4, 5, 6
  - If you meant this, it is indeed misaligned!
  - Certain hardware implementation may support this; usually not.
  - If you still want to obtain a word starting from the address 3 – get a byte from address 3, a word from address 4 and manipulate the two data to get what you want



Alignment issue does not exist for byte access.

Easy rule: **Aligned if: ADDRESS mod TYPESIZE == 0**

E.g., Is 13 aligned for a word? 13 mod 4 != 0 ➔ not aligned

Is 14 aligned for a halfword? 14 mod 2 == 0 ➔ aligned