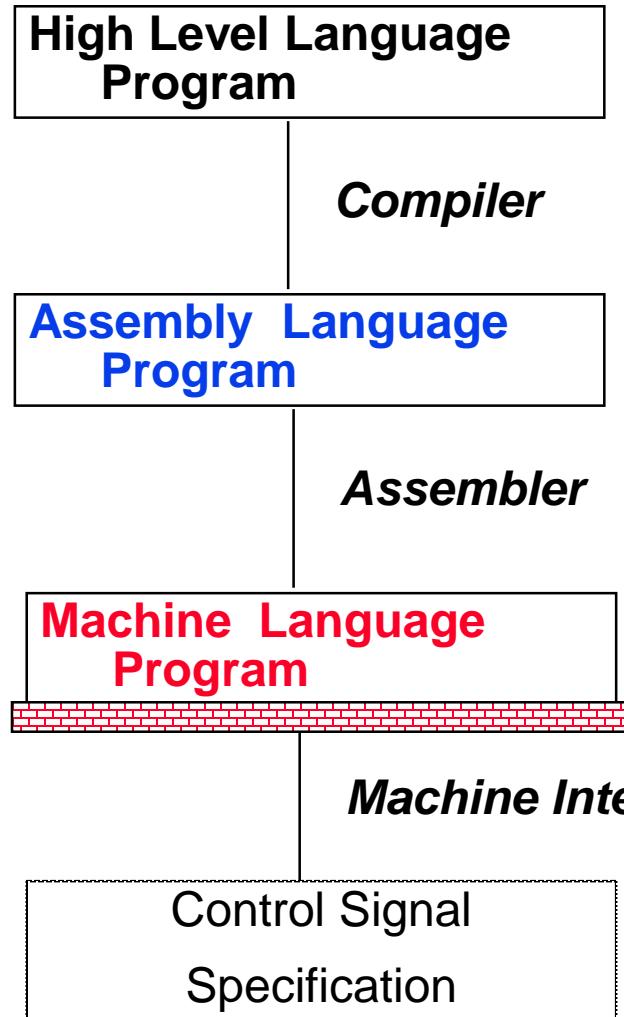

컴퓨터 공학 개론
Lecture 4

Instructions: Language of the Computer

2017

김태성

Levels of Representation



temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

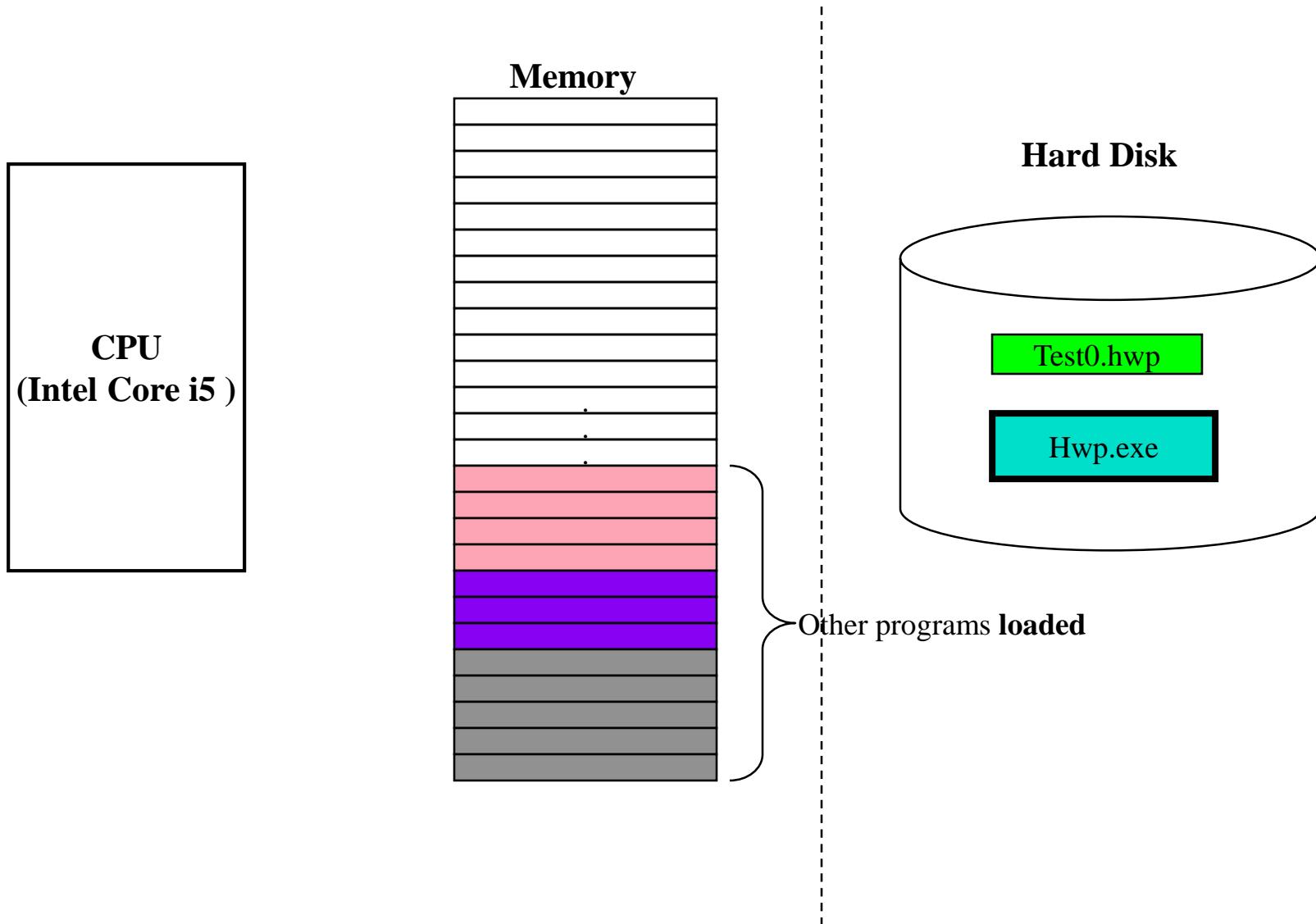
instruction

lw \$15, 0(\$2)
lw \$16, 4(\$2)
sw \$16, 0(\$2)
sw \$15, 4(\$2)

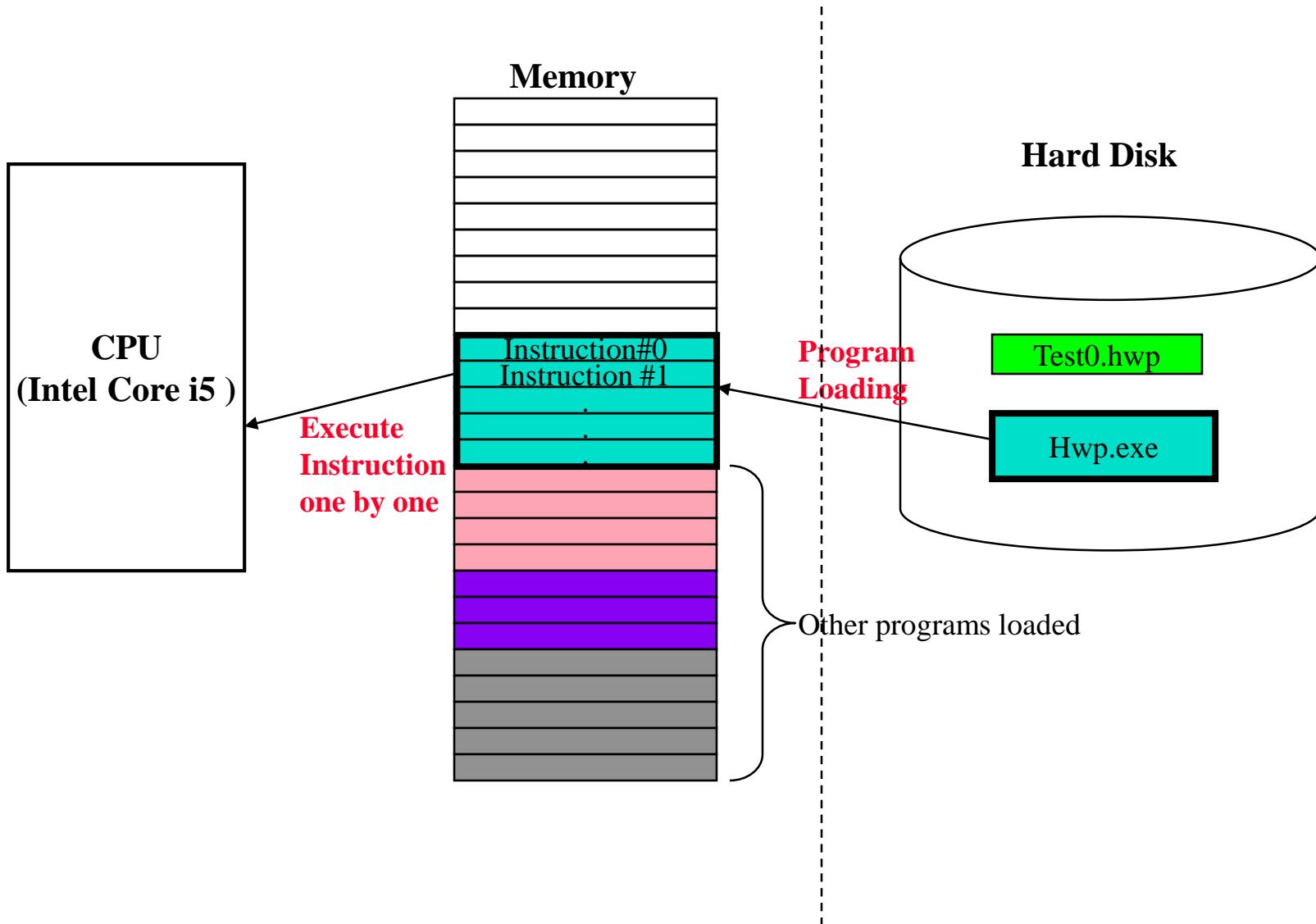
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

High and low signals on control lines

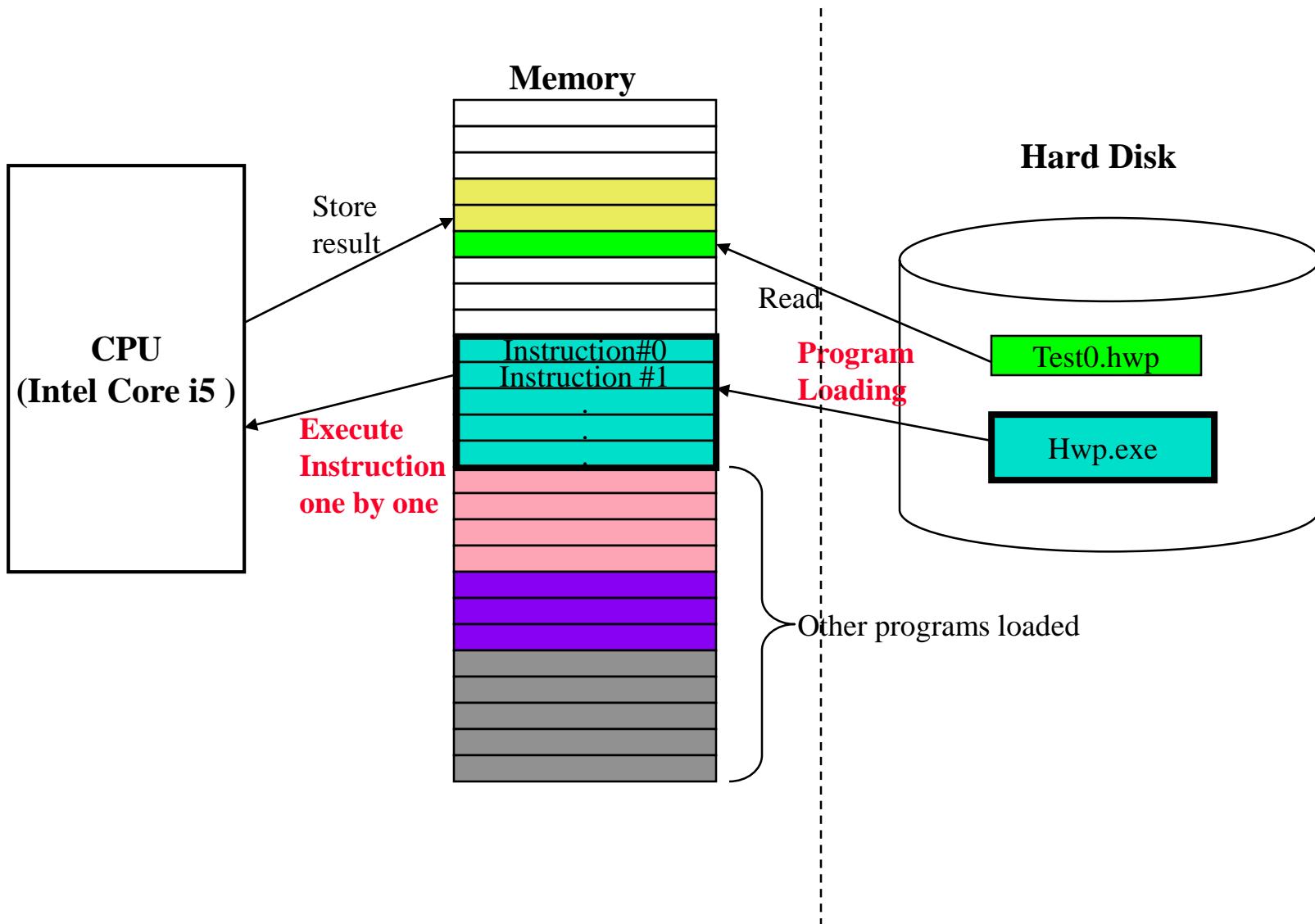
Program Execution



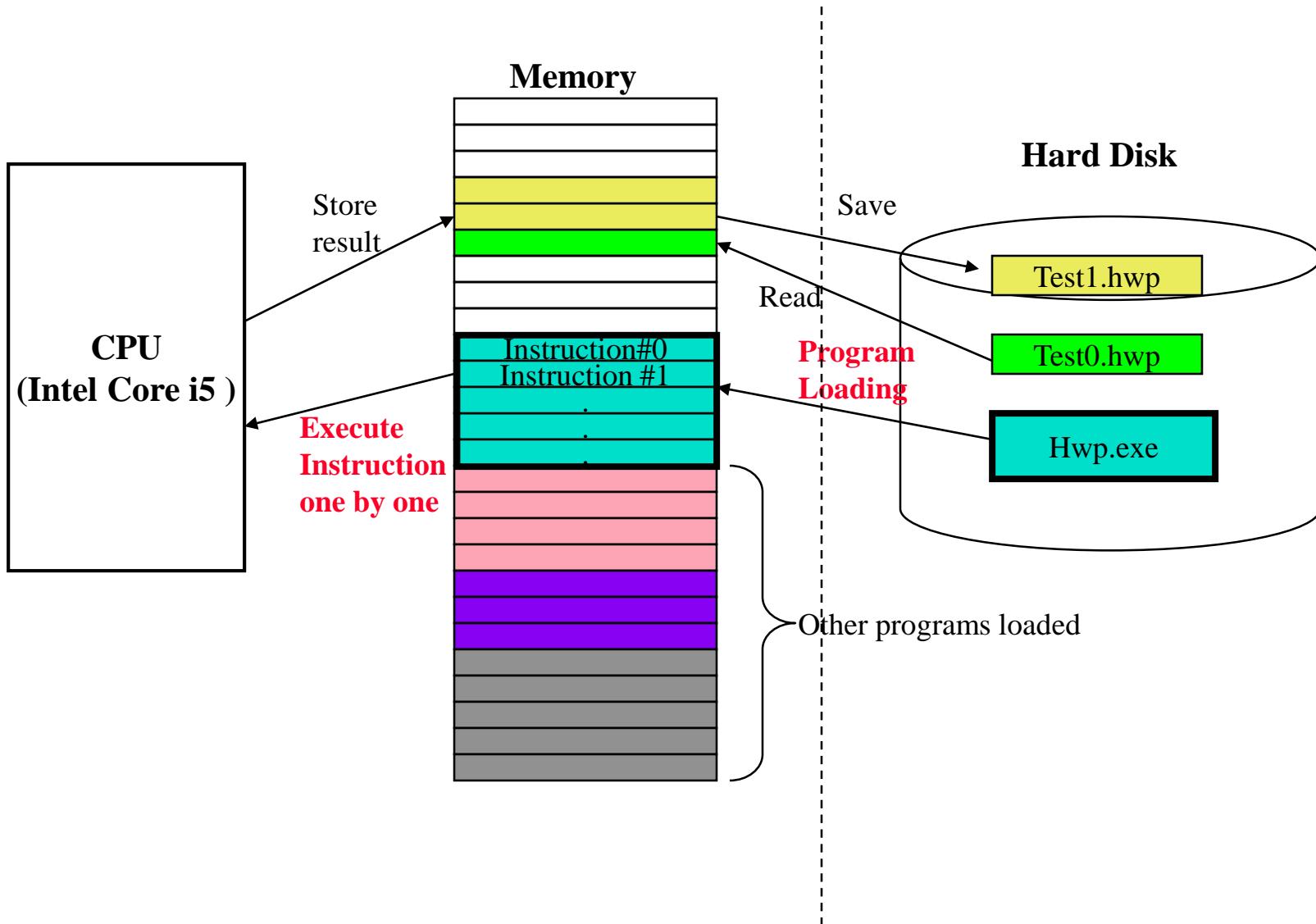
Program Execution



Program Execution



Program Execution



Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions.
 - The set of instructions a particular CPU implements is an *Instruction Set Architecture (ISA)*.
 - Examples:
 - Intel 80x86 (IA32, Pentium 4),
 - IBM/Motorola PowerPC (iMac),
 - ARM (iPhone, Galaxy S, ...),
 - Intel IA64,
 - SPARC,
 - MIPS ...

Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
 - **CISC**: Complex Instruction Set Computing
 - Intel: 80286, 80386, 80486, 80586(Pentium), Pentium2, 3, 4, ...
- **RISC** philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – Reduced Instruction Set Computing
 - Keep the instruction set small and simple, makes it easier to build fast hardware.
 - Let CPU do complicated operations by composing simpler ones.
 - All CPUs designed since 1980
 - MIPS, PowerPC, ARM, Intel Itanium (IA64), SPARC, ...

The MIPS Instruction Set

- Microprocessor without Interlocked Pipeline Stages
- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
 - Including ARM

Arithmetic Operations

- Add and subtract, three operands*

- Two sources and one destination

add a, b, c # a gets b + c

- All arithmetic operations have this form

- *Design Principle 1: Simplicity favors regularity*

- Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Operand: the object of a mathematical operation, a quantity on which an operation is performed

Arithmetic Example

- **C code:**

$f = (g + h) - (i + j);$

- **Compiled MIPS code:**

```
add t0, g, h      # temp t0 = g + h
add t1, i, j      # temp t1 = i + j
sub f, t0, t1     # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS processor has a 32×32 -bit register file inside
 - Use for frequently accessed data
 - Numbered 0 to 31 (\$0, \$1, ..., \$31)
 - 32-bit data called a “word”
 - “word” is a basic data unit in MIPS and any other 32-bit machines
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for static variables
- *Design Principle 2: Smaller is faster*
 - c.f. main memory: millions of locations vs. register: 32x32-bits

Register: a small amount of storage available
as part of a digital processor, such as a CPU

Register Operand Example

- **C code:**

a = b + c;

- **a, b, c in registers \$s0, \$s1, \$s2** (*compiler assigns registers*)

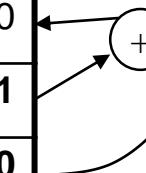
- **Compiled MIPS code:**

add \$s0, \$s1, \$s2 ; register \$s0 gets b + c

Accessing Registers

add \$s0, \$s1, \$s2 # reg \$s0 = reg \$s1 + reg \$s2

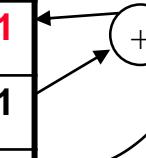
address	Name	Var.	Data
0 (00000)	\$zero		0000 0000 0000 0000 0000 0000 0000 0000
...
16 (10000)	\$s0	a	0000 0000 0000 0000 0000 0000 0000 0000
17 (10001)	\$s1	b	0000 0000 0000 0000 0000 0000 0000 0011
18 (10010)	\$s2	c	0000 0000 0000 0000 0000 0000 0100 0100
...
22 (10110)	\$s6		0110 0000 1101 0010 0000 0100 0000 0000
23 (10111)	\$s7		0000 1101 0010 0100 0110 0000 0000 0000
...
30 (11110)	\$fp		0100 0110 0000 0000 1101 0010 0000 0000
31 (11111)	\$ra		0010 0000 0000 0000 0100 0110 0000 1101



Accessing Registers

add \$s0, \$s1, \$s2 # reg \$s0 = reg \$s1 + reg \$s2

address	Name	Var.	Data
...
16 (10000)	\$s0	a	0000 0000 0000 0000 0000 0000 0100 0111
17 (10001)	\$s1	b	0000 0000 0000 0000 0000 0000 0000 0011
18 (10010)	\$s2	c	0000 0000 0000 0000 0000 0000 0100 0100
...
22 (10110)	\$s6		0110 0000 1101 0010 0000 0100 0000 0000
23 (10111)	\$s7		0000 1101 0010 0100 0110 0000 0000 0000
...
30 (11110)	\$fp		0100 0110 0000 0000 1101 0010 0000 0000
31 (11111)	\$ra		0010 0000 0000 0000 0100 0110 0000 1101



Register Operand Example

- **C code:**

f = (g + h) - (i + j);

- f, g, h, i, j in \$s0, \$s1, \$s2, \$s3, \$s4

- **\$t0 and \$t1 are temporary registers**

- **Compiled MIPS code:**

add \$t0, \$s1, \$s2 # \$t0 gets g+h

add \$t1, \$s3, \$s4 # \$t1 gets i+j

sub \$s0, \$t0, \$t1 # \$s0 gets \$t0-\$t1

Accessing Registers

```
add $t0, $s1, $s2      # reg $t0 = reg $s1 + reg $s2  
add $t1, $s3, $s4      # reg $t1 = reg $s3 + reg $s4  
sub $s0, $t0, $t1      # reg $s0 = reg $t0 - reg $t1
```

address	Name	Var.	Data
...
8 (01000)	\$t0		...
9 (01001)	\$t1		...
...
16 (10000)	\$s0	f	0000 0000 0000 0000 0000 0000 0000 0000
17 (10001)	\$s1	g	0001 0000 0000 0000 0000 0000 0000 0011
18 (10010)	\$s2	h	0000 0000 0000 0000 0000 0000 0100 0100
19 (10011)	\$s3	i	0000 0000 1101 0010 0000 0100 0000 0000
20 (10100)	\$s4	j	0000 0000 0101 0100 0000 0000 0000 0000
...

Accessing Registers

add \$t0, \$s1, \$s2 # reg \$t0 = reg \$s1 + reg \$s2

add \$t1, \$s3, \$s4 # reg \$t1 = reg \$s3 + reg \$s4

sub \$s0, \$t0, \$t1 # reg \$s0 = reg \$t0 - reg \$t1

address	Name	Var.	Data
...
8 (01000)	\$t0		0001 0000 0000 0000 0000 0000 0100 0111
9 (01001)	\$t1		...
...			...
16 (10000)	\$s0	f	0000 0000 0000 0000 0000 0000 0000 0000
17 (10001)	\$s1	g	0001 0000 0000 0000 0000 0000 0000 0011
18 (10010)	\$s2	h	0000 0000 0000 0000 0000 0000 0100 0100
19 (10011)	\$s3	i	0000 0000 1101 0010 0000 0100 0000 0000
20 (10100)	\$s4	j	0000 0000 0101 0100 0000 0000 0000 0000
...

Accessing Registers

add \$t0, \$s1, \$s2 # reg \$t0 = reg \$s1 + reg \$s2

add \$t1, \$s3, \$s4 # reg \$t1 = reg \$s3 + reg \$s4

sub \$s0, \$t0, \$t1 # reg \$s0 = reg \$t0 - reg \$t1

address	Name	Var.	Data
...
8 (01000)	\$t0		0001 0000 0000 0000 0000 0000 0100 0111
9 (01001)	\$t1		0000 0001 0010 0110 0000 0100 0000 0000
...			...
16 (10000)	\$s0	f	0000 0000 0000 0000 0000 0000 0000 0000
17 (10001)	\$s1	g	0001 0000 0000 0000 0000 0000 0000 0011
18 (10010)	\$s2	h	0000 0000 0000 0000 0000 0000 0100 0100
19 (10011)	\$s3	i	0000 0000 1101 0010 0000 0100 0000 0000
20 (10100)	\$s4	j	0000 0000 0101 0100 0000 0000 0000 0000
...

Accessing Registers

```
add $t0, $s1, $s2      # reg $t0 = reg $s1 + reg $s2  
add $t1, $s3, $s4      # reg $t1 = reg $s3 + reg $s4  
sub $s0, $t0, $t1      # reg $s0 = reg $t0 - reg $t1
```

address	Name	Var.	Data
...
8 (01000)	\$t0		0001 0000 0000 0000 0000 0000 0100 0111
9 (01001)	\$t1		0000 0001 0010 0110 0000 0100 0000 0000
...			...
16 (10000)	\$s0	f	0000 1110 1101 1001 1111 1100 0100 0111
17 (10001)	\$s1	g	0001 0000 0000 0000 0000 0000 0000 0011
18 (10010)	\$s2	h	0000 0000 0000 0000 0000 0000 0100 0100
19 (10011)	\$s3	i	0000 0000 1101 0010 0000 0100 0000 0000
20 (10100)	\$s4	j	0000 0000 0101 0100 0000 0000 0000 0000
...

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- MIPS arithmetic instructions **only operate on registers**
- To apply arithmetic operations on data in main memory
 - Load values from memory into registers => lw (load word)
 - Execute arithmetic operations on registers
 - Store result from register to memory => sw (store word)
- Memory is byte-addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4

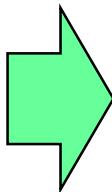
Memory Structure

Physical

Address 32-bit (hex)	Data 8-bit (1 byte)
...	...
0x0000 0014	...
0x0000 0013	...
0x0000 0012	0001 1010
0x0000 0011	0010 0110
0x0000 0010	0100 0010
0x0000 000f	0010 1010
0x0000 000e	0010 0010
0x0000 000d	1000 1110
0x0000 000c	0100 0011
0x0000 000b	0010 0010
0x0000 000a	1110 0000
0x0000 0009	0010 0010
0x0000 0008	0011 1011
0x0000 0007	0010 0010
0x0000 0006	0000 0110
0x0000 0005	0010 1000
0x0000 0004	0010 0010
0x0000 0003	0100 0010
0x0000 0002	1110 0010
0x0000 0001	0010 0010
0x0000 0000	0110 0011

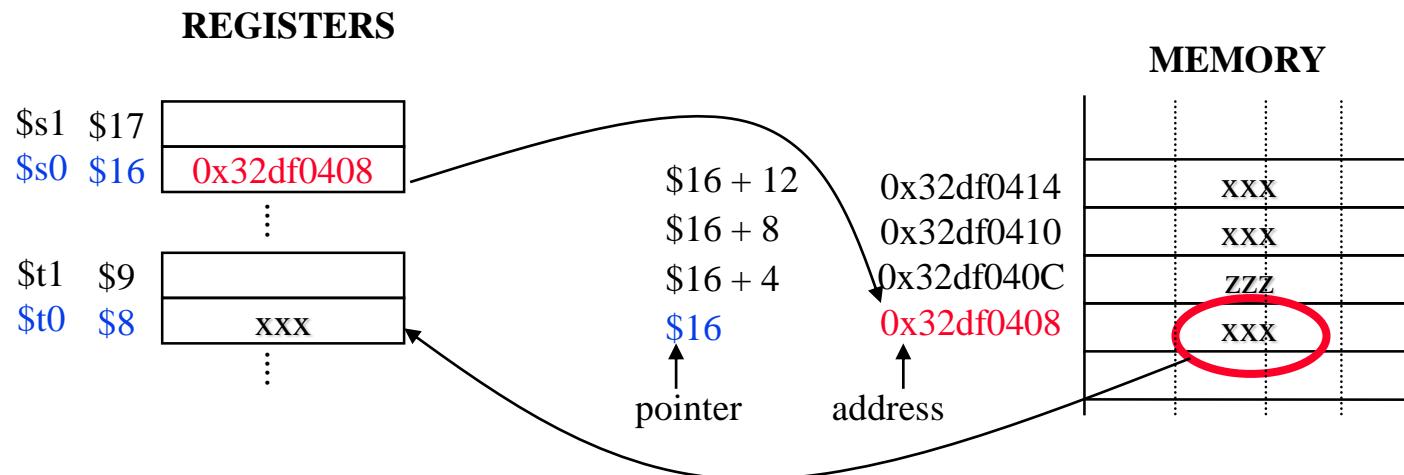
In a 32-bit MIPS ISA

Address 32-bit (hex)	Data 32-bit (4 bytes = 1 word)
...	
0x0000 0014	
0x0000 0013	0000 0000
+4 0x0000 0012	0001 1010
0x0000 0011	0010 0110
+4 0x0000 0010	0100 0010 0010 0110 0001 1010 0000 0000
0x0000 000f	0010 1010
+4 0x0000 000e	0010 0010
0x0000 000d	1000 1110
+4 0x0000 000c	0100 0011 1000 1110 0010 0010 0010 1010
0x0000 000b	0010 0010
+4 0x0000 000a	1110 0000
0x0000 0009	0010 0010
+4 0x0000 0008	0011 1011 0010 0010 1110 0000 0010 0010
0x0000 0007	0010 0010
+4 0x0000 0006	0000 0110
0x0000 0005	0010 1000
+4 0x0000 0004	0010 0010 0010 1000 0000 0110 0010 0010
0x0000 0003	0100 0010
+4 0x0000 0002	1110 0010
0x0000 0001	0010 0010
0x0000 0000	0110 0011 0010 0010 1110 0010 0100 0010



Data Transfer: Memory to Reg (1/2)

- To transfer a word of data, we need to specify two things:
 - Register: specify this by # (\$0 - \$31) or symbolic name (\$s0,..., \$t0,...)
 - Memory address: more difficult
 - think of memory as a single one-dimensional array
 - Address to memory location is specified as a 32 bit word in MIPS
 - The 32 bit address is too long to be included within an instruction which also has 32 bits in it
 - so we address it **indirectly** by supplying a **pointer** to a **memory address**.
 - other times, we want to be able to **offset** from this **pointer**.



Data Transfer: Memory to Reg (2/2)

- Load Instruction Syntax:

lw \$t0, 4(\$s0)

- where

lw: meaning Load Word, so 32 bits(one word) are loaded at a time

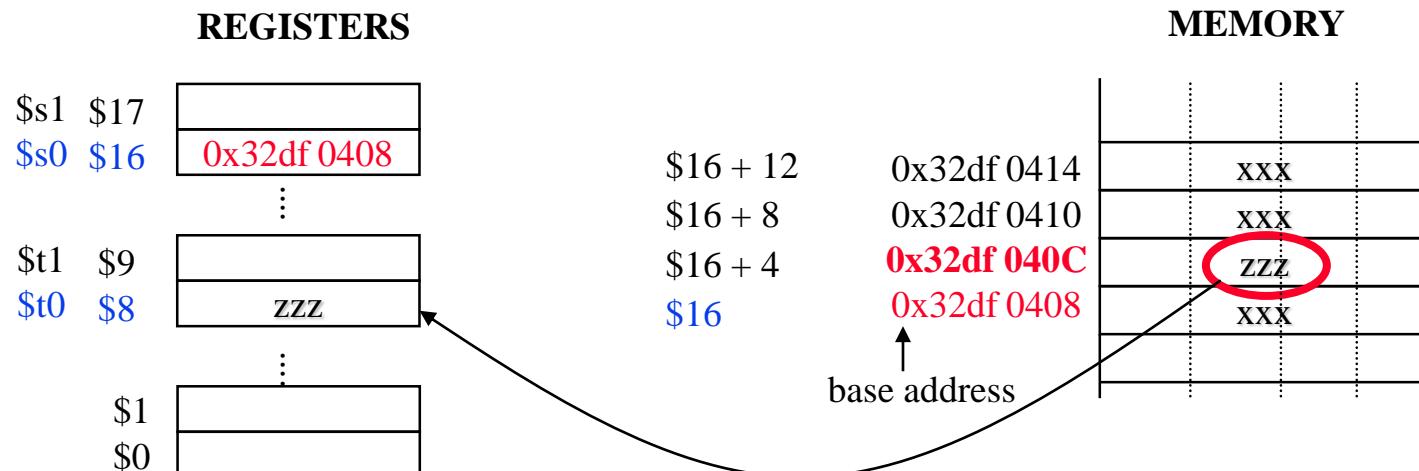
\$t0: register that will receive value

4: numerical offset **in bytes** called “offset”

\$s0: register containing pointer to memory called “base register”

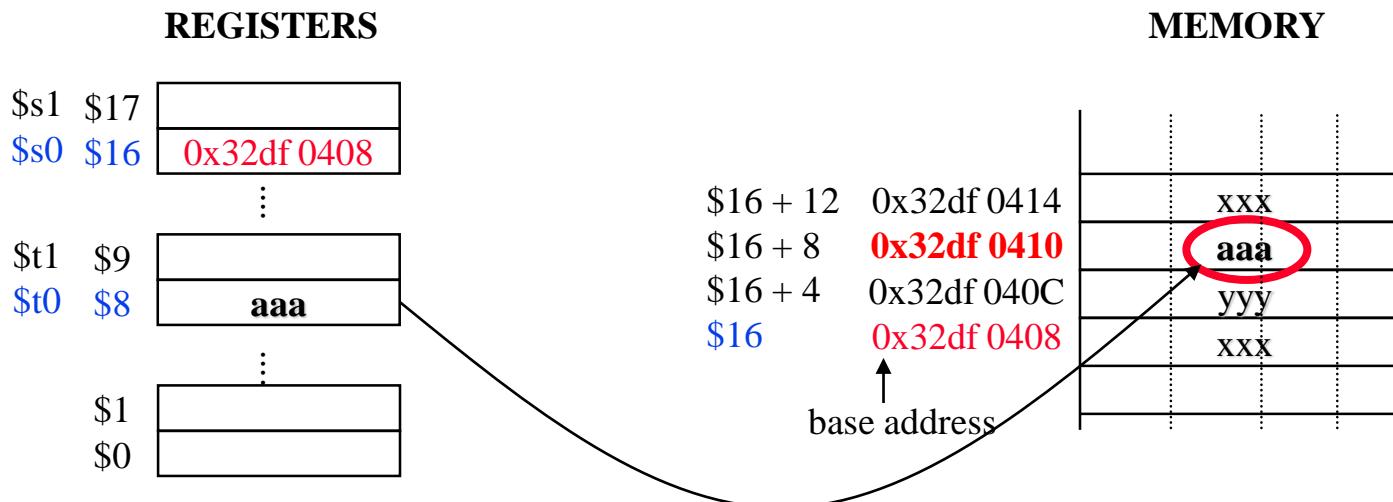
- Example: **lw \$t0, 4(\$s0)**

This instruction will take the pointer in **\$s0** (0x32df 0408), add 4 to it (0x32df040C), and then load the value from the memory pointed to by this calculated sum into register **\$t0**



Data Transfer: Reg to Memory

- Also want to store from register into memory
 - Store instruction syntax is identical to Load's
- MIPS Instruction Name:
sw (meaning Store Word, so 32 bits or one word are loaded at a time)
- Example: **sw \$t0, 8(\$s0)**
This instruction will take the pointer in **\$s0**, add 8 to it, and then store the value from register **\$t0** into that memory address



Memory Operand Example 1

- C code:

```
int A[100];           // array A[100] in memory
```

```
g = h + A[8];
```

- g in \$s1, h in \$s2, base address of A in \$s3

- Compiled MIPS code:

- Index 8 requires offset of 32

- 4 bytes per word

```
lw $t0, 32($s3)      # reg $t0 gets A[8]
```

```
add $s1, $s2, $t0
```

offset



base register

REGISTERS

\$t0

\$s3 addr

\$s2 h

\$s1 g

addr + 32

...

A[8]

...

A[1]

A[0]

MEMORY

Memory Operand Example 1

- C code:

g = h + A[8];

- g in \$s1, h in \$s2, base address of A in \$s3

- Compiled MIPS code:

- Index 8 requires offset of 32

- 4 bytes per word

lw \$t0, 32(\$s3) # reg \$t0 gets A[8]
add \$s1, \$s2, \$t0

offset

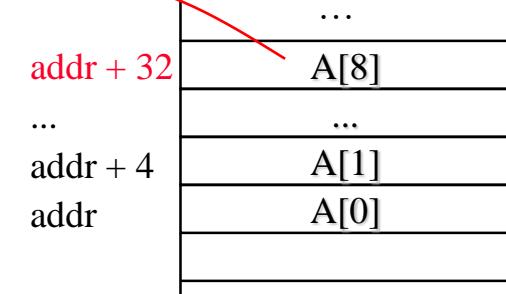


base register

REGISTERS

\$t0	A[8]
\$s3	addr
\$s2	h
\$s1	g

MEMORY



Memory Operand Example 1

- **C code:**

`g = h + A[8];`

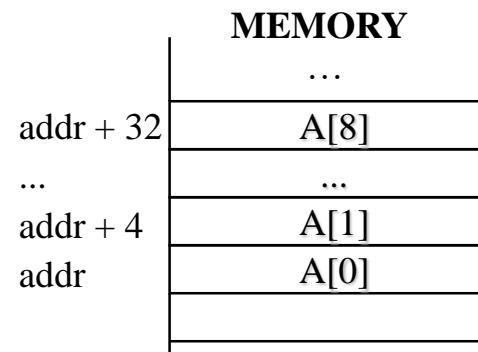
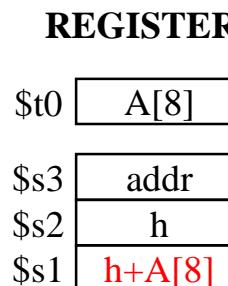
- **g** in `$s1`, **h** in `$s2`, base address of **A** in `$s3`

- **Compiled MIPS code:**

- **Index 8 requires offset of 32**

- **4 bytes per word**

```
lw $t0, 32($s3)      # reg $t0 gets A[8]
add $s1, $s2, $t0      # reg $s1 gets h+A[8]
```



Memory Operand Example 2

- C code:

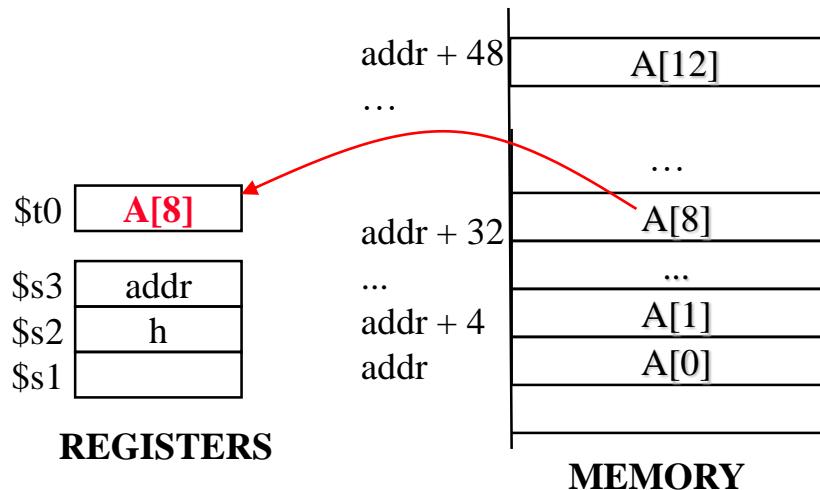
$A[12] = h + A[8];$

- h in $\$s2$, base address of A in $\$s3$

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw $t0, 32($s3)      # reg $t0 gets A[8]
add $t0, $s2, $t0      # reg $t0 gets h+A[8]
sw $t0, 48($s3)      # stores h+A[8] into A[12]
```



Memory Operand Example 2

- C code:

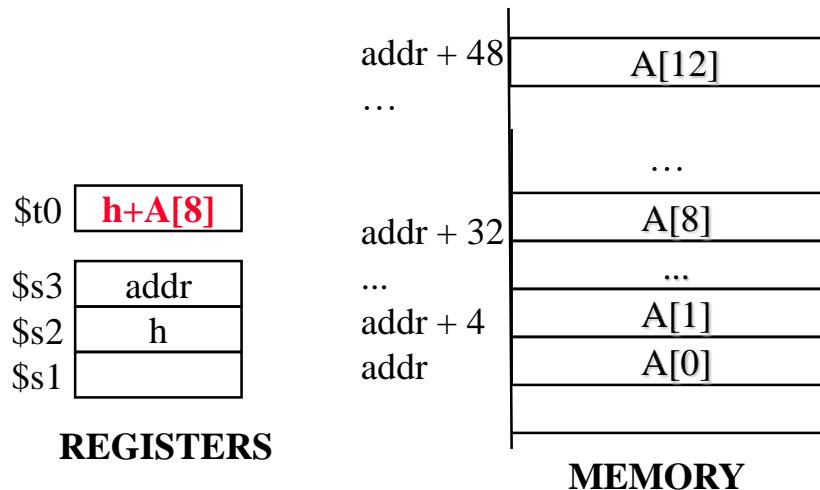
$A[12] = h + A[8];$

- h in $\$s2$, base address of A in $\$s3$

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw $t0, 32($s3)      # reg $t0 gets A[8]
add $t0, $s2, $t0      # reg $t0 gets h+A[8]
sw $t0, 48($s3)      # stores h+A[8] into A[12]
```



Memory Operand Example 2

- C code:

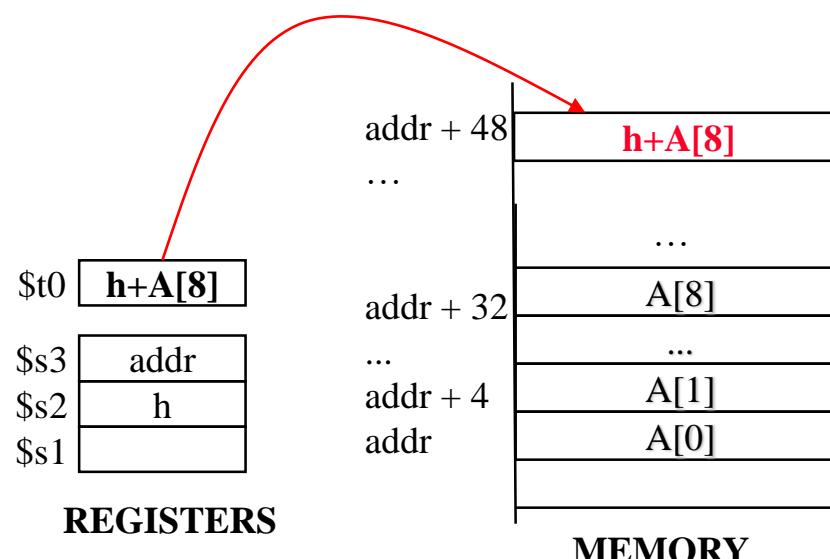
$A[12] = h + A[8];$

- h in $\$s2$, base address of A in $\$s3$

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw $t0, 32($s3)      # reg $t0 gets A[8]
add $t0, $s2, $t0      # reg $t0 gets h+A[8]
sw $t0, 48($s3)      # stores h+A[8] into A[12]
```



Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

addi \$s3, \$s3, 4

- No subtract immediate instruction

- Just use a negative constant

addi \$s2, \$s1, -1

- *Design Principle 3: Make the common case fast*

- Small constants are common
 - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., move between registers
`add $t2, $s1, $zero # $t2 gets $s1 (=s1+0)`

Unsigned Binary Integers

- ° Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$
- Example
 - 0000 0000 0000 0000 0000 0000 0000 1011₂
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
 - 0 to $+4,294,967,295$

2's-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+(2^{n-1} - 1)$
- Example
 - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 32 bits
 - $-2,147,483,648$ to $+2,147,483,647$

2's-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2's-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$\begin{aligned}x + \bar{x} &= 1111\dots111_2 = -1 \\ \bar{x} + 1 &= -x\end{aligned}$$

- Example: negate +2
 - $+2 = 0000\ 0000\dots0010_2$
 - $-2 = 1111\ 1111\dots1101_2 + 1$
 $= 1111\ 1111\dots1110_2$

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- In MIPS instruction set
 - addi: extend immediate value
 - 1b, 1h: extend loaded byte/halfword
 - beq, bne: extend the displacement