Representing Programs in Machine Language

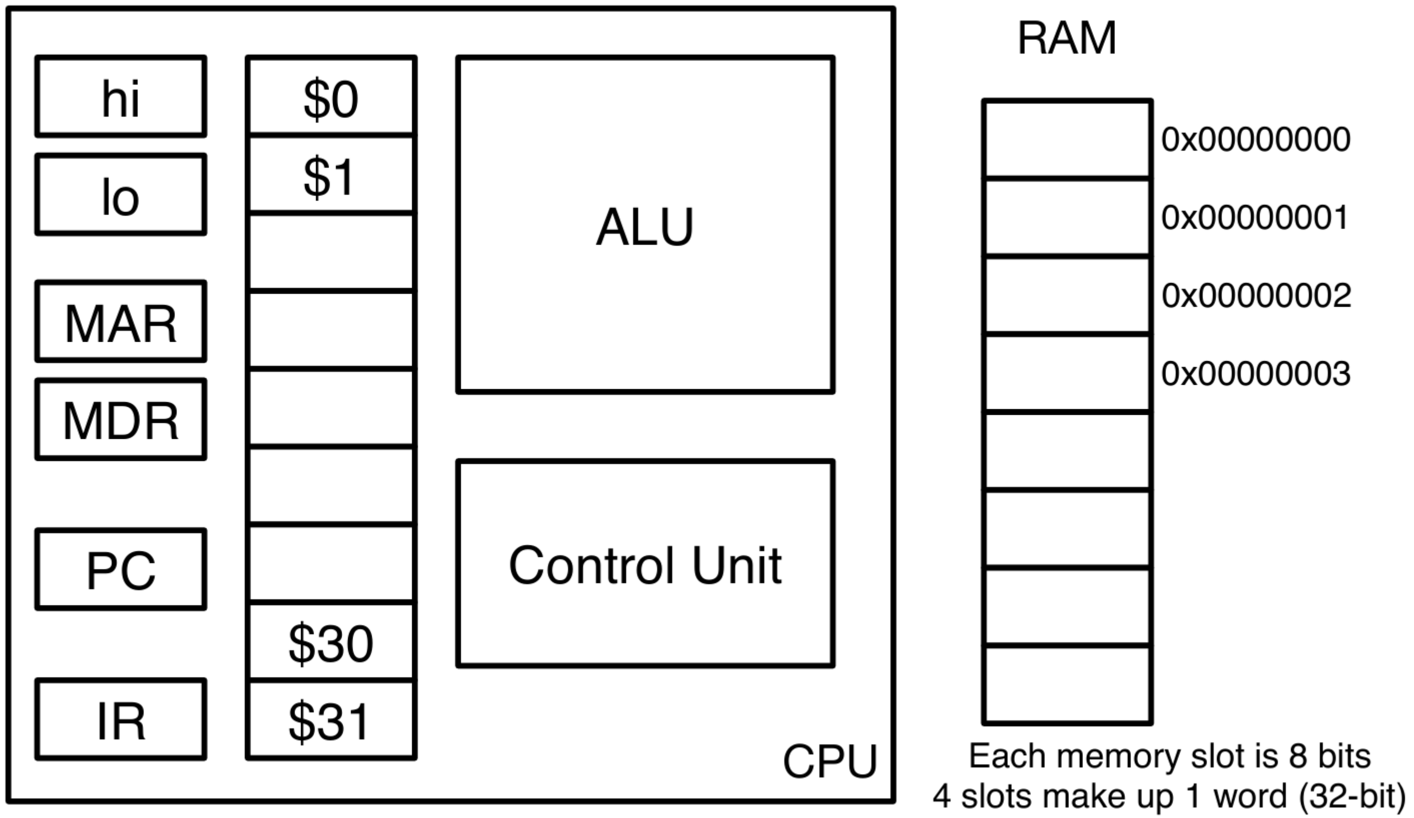
Machine Language

- Previously, we saw methods of representing *numbers* and *text* in binary, as certain patterns of 0s and 1s.
- How do we represent *programs* in binary?
- The idea is to define patterns of bits that represent **instructions** for the computer to perform.
- The computer then runs a simple loop (implemented in hardware) that fetches the instructions and executes them.
- The patterns of bits that a computer understands as valid instructions are the computer's **machine language**.

MIPS Machine Language

- In CS 241, we use a machine language called MIPS.
  - It stands for “Microprocessor without Interlocked Pipeline Stages”, in reference to the hardware design.
- MIPS is not as widely used today as other machine languages like ARM (used in CS 251).
- However, the basic concepts transfer to other machine languages, so understanding MIPS is still helpful.
- There is some interest in switching CS 241 to use ARM eventually (for parity with CS 251) but it’s a lot of work.

MIPS Hardware



MIPS Hardware: Key Points

- The **control unit** determines the sequence of instructions to execute.
- The **arithmetic logic unit** (ALU) performs mathematical operations.
- There are a small number of **registers**, which are physically located on the processor, and each store a 32-bit word.
- We have **random access memory** (RAM), which is separate from the processor. We think of it as a large array of bytes (8-bit chunks).
- Registers are significantly faster to access than RAM, but we only have a handful of registers, and we have a lot of RAM.
- In real-world computers there are more levels of “memory hierarchy”.

More About Registers

- We have 32 “general purpose registers”, labelled \$0 to \$31.
  - \$0 *always holds 0.* If you try to assign it a non-zero value, nothing will happen.
  - The other general purpose registers can be assigned values freely, but \$30 and \$31 have special meanings in MIPS machine language programming.
  - \$29 will also have a special meaning later in this course.
- PC (Program Counter) and IR (Instruction Register) are used to fetch and execute instructions. More on this shortly!
- MDR (Memory Data Register) and MAR (Memory Address Register) are used internally by memory access instructions.
- “hi” and “lo” are special registers used for multiplication and division.

More about RAM

- We will often just refer to RAM as “memory”. Technically registers are a type of “memory”, so “main memory” would be more accurate.
- We think of RAM as an array of bytes, i.e., each “slot” in RAM contains a single byte. We will call this array “MEM” (for “memory”).
- We will use array-style notation: MEM[0] refers to the first byte in memory, MEM[1] refers to the second byte, etc.
- The index of a byte in the MEM array is called the **memory address** of the byte.
- We usually write memory addresses in *hexadecimal*.

RAM and Words

- Each slot in RAM is a single byte, but our simplified MIPS instruction set only allows accessing one *word* of memory at a time.
- Recall: A word in our MIPS architecture is 32 bits (4 bytes).
- Additionally, our MIPS instruction set only allows **word-aligned** accesses, that is, the address we access must be a *multiple of 4*.
  - Real MIPS allows unaligned memory access, but it is potentially slower.
- Memory accesses (loading from memory or storing to memory) will load or store the *entire word* that begins at the provided address (4 consecutive memory slots) rather than just a single byte.

Code is Data

- Recall: The computer executes code by running a simple loop that fetches instructions and executes them.
- We call this the **fetch-execute cycle**.
- The instructions themselves are stored in RAM.
- But other non-instruction data, like the input or files the program is operating on, might also be stored in RAM.
- **The fetch-execute cycle makes no distinction between code and other types of data!** In other words, program code **is** just data.
- We'll see some consequences of this later.

The Fetch-Execute Cycle

- At the start of each cycle, the program counter (PC) register contains the *memory address* of the next instruction to execute.

1. The instruction at the memory address in PC is loaded into IR.
2. PC is incremented by 4 bytes (one word), which is the size of a single instruction.
3. The instruction in IR is executed.

- The order is very important – some instructions use or modify PC!
- When the instruction is executed in step 3, PC is already pointing to the *next instruction*.

Running a Program

- A program is a sequence of machine language instructions.
- Each instruction is represented by a 32-bit (4 byte) word.
- To run a program, we need to *place the sequence of instruction words in memory*, then *set PC to the memory address of the first word.*
- Then the fetch-execute cycle takes care of the rest.
- A program called a **loader** is used to place programs in memory and set PC, but we won't discuss loaders until the middle of the course.
- For now we will make the simplifying (but unrealistic) assumption that programs are always loaded at memory address 0x00.

MIPS Machine Language in Detail

- We use a simplified version of the real-world MIPS machine language.
- This lecture, we'll cover instructions for the following:
  - Addition and subtraction
  - Loading a constant into a register
  - Jumps (like "go to", transfers control to a certain location in memory)
  - Memory access (load from memory, store in memory)
- Later in the course, we'll cover:
  - Multiplication, division, and modulo
  - Numeric "less than" comparison
  - Conditional branching (like "go to", but conditional)
  - Calls (jumps with the ability to "return" to the call location)

Addition

- **Machine language encoding:** Add [$d = $s + $t]

```
000000 sssss ttttt ddddd 00000 100000
```

- The “sssss”, “ttttt” and “ddddd” parts are replaced with the numbers of registers $s, $t and $d, encoded as 5-bit unsigned values.
- For example, for $3 = $2 + $1 we write:

```
000000 00010 00001 00011 00000 100000
```

- This instruction treats $s and $t as integers, adds $s + $t, and stores the result in $d.

Subtraction

- **Machine language encoding:** Subtract [$d = $s – $t]

```
000000 sssss ttttt ddddd 00000 100010
```

- The only difference from addition is in the last 6 bits.
- These are called the **function bits**.
- If the first 6 bits, called the **opcode bits**, are all 0, the function bits are used to decide which instruction to perform.
- This instruction treats $s and $t as integers, subtracts $s – $t, and stores the result in $d.

Loading Constant Values

- Constant values are often called **immediates** in the context of machine language.
- Real MIPS has an "add immediate" instruction that lets you do addition with a register and a constant.
- You can do something like $r = $0 + [constant] to *set* $r to a constant.
- Our MIPS variant doesn't have this, so instead we have a special instruction for loading constant values into registers.
- This special instruction takes advantage of the fact that **code is data** and is not treated differently from other data in memory.

Load Immediate and Skip

- **Machine language encoding:** Load Immediate And Skip [$d]

```
000000 00000 00000 ddddd 00000 010100
```

- We'll call this "LIS" for short.
- This instruction has two steps:
  - Copy the value in memory at PC into $d. (For short: $d = MEM[PC])
  - Increment PC by 4.
- This is the first instruction we have seen that uses PC.
- Think carefully about how this interacts with the fetch-execute cycle.

How LIS Works

- Recall the order of operations in the fetch-execute cycle:
    1. Fetch the instruction at MEM[PC], storing it in IR.
    2. Increment PC by 4 bytes (1 word).
    3. Execute the instruction in IR.
- When LIS runs, at step 3, PC is pointing to the word in memory *after* the LIS instruction.
- We use this word in memory to store the constant we want to load.
- The LIS instruction does $d = MEM[PC], storing the constant in $d.
- Then it increments PC by 4. Why is this important?
- Otherwise the fetch-execute cycle would try to execute our constant!

MIPS in Action

```
   Instructions       Machine Language (Binary)        (Hex)
PC LIS into $8        00000000000000000100000000010100 0x00004014
   (constant)         00000000000000000000000000001011 0x0000000B
   LIS into $9        00000000000000000100100000010100 0x00004814
   (constant)         00000000000000000000000000001101 0x0000000D
   Add $3 = $8 + $9   00000001000010010001100000100000 0x01091820
```

```
$3 = ?              PC = 0x00000000
$8 = ?              IR = ?
$9 = ?              Next Step: Fetch
```

MIPS in Action

	Instructions	Machine Language (Binary)	(Hex)
PC	LIS into $8	00000000000000000100000000010100	0x00004014
	(constant)	00000000000000000000000000001011	0x0000000B
	LIS into $9	00000000000000000100100000010100	0x00004814
	(constant)	00000000000000000000000000001101	0x0000000D
	Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

$3 = ?　　　PC = 0x00000000

$8 = ?　　　IR = <u>0x00004014</u> (LIS into $8)

$9 = ?　　　Next Step: Increment PC by 4

MIPS in Action

	Instructions	Machine Language (Binary)	(Hex)
	LIS into $8	00000000000000000100000000010100	0x00004014
PC	(constant)	00000000000000000000000000001011	0x0000000B
	LIS into $9	00000000000000000100100000010100	0x00004814
	(constant)	00000000000000000000000000001101	0x0000000D
	Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

$3 = ?

$8 = ?

$9 = ?

PC = 0x00000004

IR = 0x00004014 (LIS into $8)

Next Step: Execute

MIPS in Action

```
   Instructions       Machine Language (Binary)       (Hex)
   LIS into $8        00000000000000000100000000010100 0x00004014
PC (constant)         00000000000000000000000000001011 0x0000000B
   LIS into $9        00000000000000000100100000010100 0x00004814
   (constant)         00000000000000000000000000001101 0x0000000D
   Add $3 = $8 + $9   00000001000010010001100000100000 0x01091820

$3 = ?               PC = 0x00000004
$8 = ?               IR = 0x00004014 (LIS into $8)
$9 = ?               Next Step (LIS): Set $d = MEM[PC]
```

MIPS in Action

Instructions	Machine Language (Binary)	(Hex)
LIS into $8	00000000000000000100000000010100	0x00004014
PC (constant)	00000000000000000000000000001011	0x0000000B
LIS into $9	00000000000000000100100000010100	0x00004814
(constant)	00000000000000000000000000001101	0x0000000D
Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

$3 = ? PC = 0x00000004

$8 = 0x0000000B IR = 0x00004014 (LIS into $8)

$9 = ? Next Step (LIS): Increment PC by 4

MIPS in Action

```
    Instructions       Machine Language (Binary)          (Hex)
    LIS into $8        00000000000000000100000000010100 0x00004014
    (constant)         00000000000000000000000000001011 0x0000000B
PC  LIS into $9        00000000000000000100100000010100 0x00004814
    (constant)         00000000000000000000000000001101 0x0000000D
    Add $3 = $8 + $9   00000001000010010001100000100000 0x01091820

$3 = ?              PC = 0x00000008
$8 = 0x0000000B     IR = 0x00004014 (LIS into $8)
$9 = ?              Next Step: Fetch
```

MIPS in Action

	Instructions	Machine Language (Binary)	(Hex)
	LIS into $8	00000000000000000100000000010100	0x00004014
	(constant)	00000000000000000000000000001011	0x0000000B
PC	LIS into $9	00000000000000000100100000010100	0x00004814
	(constant)	00000000000000000000000000001101	0x0000000D
	Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

$3 = ? PC = 0x00000008

$8 = 0x0000000B IR = 0x00004814 (LIS into $9)

$9 = ? Next Step: Increment PC by 4

MIPS in Action

	Instructions	Machine Language (Binary)	(Hex)
	LIS into $8	00000000000000000100000000010100	0x00004014
	(constant)	00000000000000000000000000001011	0x0000000B
	LIS into $9	00000000000000000100100000010100	0x00004814
PC	(constant)	00000000000000000000000000001101	0x0000000D
	Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

```
$3 = ?            PC = 0x0000000C
$8 = 0x0000000B   IR = 0x00004814 (LIS into $9)
$9 = ?            Next Step: Execute
```

MIPS in Action

Instructions	Machine Language (Binary)	(Hex)
LIS into $8	00000000000000000100000000010100	0x00004014
(constant)	00000000000000000000000000001011	0x0000000B
LIS into $9	00000000000000000100100000010100	0x00004814
PC (constant)	00000000000000000000000000001101	0x0000000D
Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

```
$3 = ?            PC = 0x0000000C
$8 = 0x0000000B   IR = 0x00004814 (LIS into $9)
$9 = ?            Next Step (LIS): Set $d = MEM[PC]
```

MIPS in Action

```
    Instructions       Machine Language (Binary)         (Hex)
    LIS into $8        00000000000000000100000000010100 0x00004014
    (constant)         00000000000000000000000000001011 0x0000000B
    LIS into $9        00000000000000000100100000010100 0x00004814
PC  (constant)         00000000000000000000000000001101 0x0000000D
    Add $3 = $8 + $9   00000001000010010001100000100000 0x01091820

$3 = ?           PC = 0x0000000C
$8 = 0x0000000B  IR = 0x00004814 (LIS into $9)
$9 = 0x0000000D  Next Step (LIS): Increment PC by 4
```

MIPS in Action

```
    Instructions       Machine Language (Binary)        (Hex)
    LIS into $8        00000000000000000100000000010100 0x00004014
    (constant)         00000000000000000000000000001011 0x0000000B
    LIS into $9        00000000000000000100100000010100 0x00004814
    (constant)         00000000000000000000000000001101 0x0000000D
PC  Add $3 = $8 + $9   00000001000010010001100000100000 0x01091820

$3 = ?           PC = 0x00000010
$8 = 0x0000000B  IR = 0x00004814 (LIS into $9)
$9 = 0x0000000D  Next Step: Fetch
```

MIPS in Action

Instructions	Machine Language (Binary)	(Hex)
LIS into $8	00000000000000000100000000010100	0x00004014
(constant)	00000000000000000000000000001011	0x0000000B
LIS into $9	00000000000000000100100000010100	0x00004814
(constant)	00000000000000000000000000001101	0x0000000D
PC Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

```
$3 = ?            PC = 0x00000010
$8 = 0x0000000B   IR = 0x01091820 (Add $3 = $8 + $9)
$9 = 0x0000000D   Next Step: Increment PC by 4
```

MIPS in Action

Instructions	Machine Language (Binary)	(Hex)
LIS into $8	00000000000000000100000000010100	0x00004014
(constant)	00000000000000000000000000001011	0x0000000B
LIS into $9	00000000000000000100100000010100	0x00004814
(constant)	00000000000000000000000000001101	0x0000000D
Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

PC

$3 = ?   PC = 0x00000014

$8 = 0x0000000B   IR = 0x01091820 (Add $3 = $8 + $9)

$9 = 0x0000000D   Next Step: Execute

MIPS in Action

Instructions	Machine Language (Binary)	(Hex)
LIS into $8	00000000000000000100000000010100	0x00004014
(constant)	00000000000000000000000000001011	0x0000000B
LIS into $9	00000000000000000100100000010100	0x00004814
(constant)	00000000000000000000000000001101	0x0000000D
Add $3 = $8 + $9	00000001000010010001100000100000	0x01091820

PC

$3 = 0x00000018   PC = 0x00000014

$8 = 0x0000000B   IR = 0x01091820 (Add $3 = $8 + $9)

$9 = 0x0000000D   Next Step: Fetch

What Happens After The End

- In the previous example, what happens after the last instruction?
- The fetch-execute cycle is just going to go on fetching and executing whatever is in memory after our program.
- What we actually want is for our program to stop and *return to the loader* (the program that placed our program in memory).
- There is an instruction that lets us jump (i.e., set PC to) an arbitrary memory address, so that we can transfer control to some other code.
- Before running our program, the loader stores the correct return address in **$31** so that we can return there when we're done.

Jump Register

- **Machine language encoding:** Jump Register [$s]

    `000000 sssss 00000 00000 00000 001000`

- This instruction sets PC to the value in $s.
- After the jump, the MIPS machine will resume the fetch-execute cycle from the new location.
- Most commonly used with $31 to return to the loader when a program is finished (thus ending the program).
- Later when we discuss how to implement procedures, it will also be used for returning from a procedure call.

Memory Access Instructions

- **Machine language encodings:**

```
Load Word  100011 sssss ttttt iiii iiii iiii iiii
Store Word 101011 sssss ttttt iiii iiii iiii iiii
```

- These instructions operate on **words**, not single bytes.
  - Load Word loads the 4 consecutive bytes starting at address \$s + i into \$t.
  - Store Word stores \$t in the 4 consecutive bytes starting at address \$s + i.
- We'll use this notation:
  - Load Word [\$t ← \$s + i] means load the word from address \$s + i into \$t.
  - Store Word [\$t → \$s + i] means store the word in \$t into address \$s + i.

Memory Access Instructions

- **Machine language encodings:**

```
Load Word  100011 sssss ttttt iiii iiii iiii iiii
Store Word 101011 sssss ttttt iiii iiii iiii iiii
```

- This is the first instruction we have seen that has an **immediate** (constant) operand.
- The i field is interpreted as a 16-bit two’s complement integer.
- You can think of the register operand $s as a “base address”, and the immediate operand i can be used to easily specify an “offset” from this base address.

Example: Working with Arrays

- When you write "int A[3] = {1,2,3};" in C/C++, how is this stored?
- Assuming "int" is 32 bits, it's like this: →
- When you use A[i] to access an element, C/C++ implicitly multiplies i by sizeof(int) to ensure you access the right data.
- **This does not happen automatically in machine code.** You must do it yourself!
- If $1 contains the address of A:
  - Load Word [$3 ← $1 + 0] loads A[0] into $3
  - Load Word [$3 ← $1 + **4**] loads A[1] into $3

A[0]	MEM[address of A]	00000000
	MEM[address of A + 1]	00000000
	MEM[address of A + 2]	00000000
	MEM[address of A + 3]	00000001
A[1]	MEM[address of A + 4]	00000000
	MEM[address of A + 5]	00000000
	MEM[address of A + 6]	00000000
	MEM[address of A + 7]	00000010
A[2]	MEM[address of A + 8]	00000000
	MEM[address of A + 9]	00000000
	MEM[address of A + 10]	00000000
	MEM[address of A + 11]	00000011

Example: Working with Arrays

- Because LW loads one **word** at a time, use multiples of 4 as offsets.
  - Element A[i] starts at (address of A) + (4 · i)
- This works fine for “hardcoded” accesses, e.g. for $3 = A[2], use offset i = 8 from the starting address of the array.
- What if we want to access A[i] where i is the value in $2? We can’t encode register numbers in the i field of the instruction.
- We need to multiply $2 by 4 in code.

A[0]	MEM[address of A]	00000000
	MEM[address of A + 1]	00000000
	MEM[address of A + 2]	00000000
	MEM[address of A + 3]	00000001
A[1]	MEM[address of A + 4]	00000000
	MEM[address of A + 5]	00000000
	MEM[address of A + 6]	00000000
	MEM[address of A + 7]	00000010
A[2]	MEM[address of A + 8]	00000000
	MEM[address of A + 9]	00000000
	MEM[address of A + 10]	00000000
	MEM[address of A + 11]	00000011

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.
- Goal: A[size – 1] = 0x241c
- What’s the memory address of A[size – 1]?
  - $1 = address of A[0]
  - $1 + 4 = address of A[1]
  - $1 + 8 = address of A[2] …
  - $1 + (size – 1) * 4 = address of A[size – 1]
- The size is stored in $2, so we can compute this address.

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.
- Goal: A[size – 1] = 0x241c
- Using **Store Word**, set address **$1 + (size – 1) * 4** to value **0x241c**.
- We can rewrite this as $1 + (size * 4) – 4.
  - Store Word [$t → $s + i] becomes Store Word [0x241c → $1 + (size * 4) + (-4)]
- But Store Word [$t → $s + i] is encoded in machine language as:
  `101011 sssss ttttt iiii iiii iiii iiii`
- The s and t bits encode *register numbers*.

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.
- Goal: A[size – 1] = 0x241c
- Using **Store Word**, set address **$1 + (size – 1) * 4** to value **0x241c**.
- We could do this with something like:

  Store Word [0x241c → $1 + (size * 4) + (-4)]
- But we need to load 0x241c into a register.
- We also need to compute $1 + (size * 4) and place it in a register.

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.
- Goal: A[size – 1] = 0x241c
- The Plan:

```
LIS [$3 ← 0x241c] load the hex constant
Add [$2 = $2 + $2] $2 = size * 2
Add [$2 = $2 + $2] $2 = size * 4
Add [$1 = $1 + $2] $1 = $1 + (size * 4)
SW  [$3 → $1 – 4] use store word to set A[size-1] to 0x241c
$3 contains 0x241c
$1 contains address of A[size], subtracting 4 gives address of A[size-1]
```

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
LIS [$3 ← 0x241c] load the hex constant
Add [$2 = $2 + $2] $2 = size * 2
Add [$2 = $2 + $2] $2 = size * 4
Add [$1 = $1 + $2] $1 = $1 + (size * 4)
SW  [$3 → $1 – 4] use store word to set A[size-1] to 0x241c
```

- We now have to translate our “plan” into actual machine language instructions that the MIPS processor can understand.
- This is a simple but tedious matter of looking up the encodings and filling in the necessary parameters.

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

`LIS [$3 ← 0x241c]` *load the hex constant*

- This line will become two machine code words: the LIS instruction and the constant 0x241c.
- LIS [$d] encoding: `000000 00000 00000 ddddd 00000 010100`
  - For $3, d = 00011 → `000000 00000 00000 00011 00000 010100`
- 0x241c as a 32-bit word is: `0000 0000 0000 0000 0010 0100 0001 1100`
  - Just translate each hex digit and pad on the left with 0s.

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
00000000000000000001100000010100  LIS [$3]
00000000000000000010010000011100  0x241c
Add [$2 = $2 + $2] $2 = size * 2
Add [$2 = $2 + $2] $2 = size * 4
Add [$1 = $1 + $2] $1 = $1 + (size * 4)
SW  [$3 → $1 – 4] use store word to set A[size-1] to 0x241c
```

- Add [$d = $s + $t]: `000000 sssss ttttt ddddd 00000 100000`
  - Replacing s, t, d with 00010 for $2:
    `000000 00010 00010 00010 00000 100000`

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
00000000000000000001100000010100  LIS [$3]
00000000000000000010010000011100  0x241c
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000010000100001000000100000  Add [$2 = $2 + $2]
Add [$1 = $1 + $2] $1 = $1 + (size * 4)
SW  [$3 → $1 – 4] use store word to set A[size-1] to 0x241c
```

- Add [$d = $s + $t]: `000000 sssss ttttt ddddd 00000 100000`
  - Replacing d and s with 00001 for $1, and t with 00010 for $2:
    `000000 00001 00010 00001 00000 100000`

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
00000000000000000001100000010100  LIS [$3]
00000000000000000010010000011100  0x241c
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000001000100000100000100000  Add [$1 = $1 + $2]
SW  [$3 → $1 – 4] use store word to set A[size-1] to 0x241c
```

- SW [$t → $s + i]: `101011 sssss ttttt iiii iiii iiii iiii`
  - Replace s with 00001 for $1, and t with 00011 for $3.
  - For i, encode -4 in two’s complement! 00…0100 → 11…1011 + 1 → 11…1100

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
00000000000000000001100000010100  LIS [$3]
00000000000000000010010000011100  0x241c
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000001000100000100000100000  Add [$1 = $1 + $2]
SW  [$3 → $1 – 4] use store word to set A[size-1] to 0x241c
```

- SW [$t → $s + i]: `101011 sssss ttttt iiii iiii iiii iiii`

  `101011 00001 00011 1111 1111 1111 1100`

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
00000000000000000001100000010100   LIS [$3]
00000000000000000010010000011100   0x241c
00000000010000100001000000100000   Add [$2 = $2 + $2]
00000000010000100001000000100000   Add [$2 = $2 + $2]
00000000001000100000100000100000   Add [$1 = $1 + $2]
10101100001000111111111111111100   SW  [$3 → $1 – 4]
```

- Technically we are done if the goal is just to write a “code snippet”, but if we want a proper *program* that terminates when it is finished, we need a Jump Register [$31] instruction at the end!

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
00000000000000000001100000010100  LIS [$3]
00000000000000000010010000011100  0x241c
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000001000100000100000100000  Add [$1 = $1 + $2]
10101100001000111111111111111100  SW  [$3 → $1 – 4]
```

- JR [$s]: `000000 sssss 00000 00000 00000 001000`
  - Replacing s with 11111 for $31:
    `00000 11111 00000 00000 00000 001000`

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
00000000000000000001100000010100  LIS [$3]
00000000000000000010010000011100  0x241c
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000001000100000100000100000  Add [$1 = $1 + $2]
10101100001000111111111111111100  SW  [$3 → $1 – 4]
00000011111000000000000000001000  JR  [$31]
```

- This is a complete MIPS machine language program that overwrites the last element of an array (location and size specified by $1 and $2) with the constant value 0x241c and then returns.

Example: Working with Arrays

- Suppose $1 contains the address of A and $2 contains the size of A.
- Write MIPS machine code that sets the last element of A to 0x241c.

```
00000000000000000001100000010100
00000000000000000010010000011100
00000000010000100001000000100000
00000000010000100001000000100000
00000000001000100000100000100000
10101100001000111111111111111100
00000011111000000000000000001000
```

Example: A Strange Input

- Let’s suppose we load our array-modifying program into memory at address zero.

```
Address   Data in Memory                    Meaning
0x00 (0)  00000000000000000001100000010100  LIS [$3]
0x04 (4)  00000000000000000010010000011100  0x241c
0x08 (8)  00000000010000100001000000100000  Add [$2 = $2 + $2]
0x0c (12) 00000000010000100001000000100000  Add [$2 = $2 + $2]
0x10 (16) 00000000001000100000100000100000  Add [$1 = $1 + $2]
0x14 (20) 10101100001000111111111111111100  SW  [$3 → $1 - 4]
0x18 (24) 00000011111000000000000000001000  JR  [$31]
```

- Something wonderful will happen if we specify the *array address* $1 as zero, and the *array size* $2 as 7.

Example: A Strange Input

- Suppose $1 = 0 and $2 = 7 initially when we run this program. What is going to happen?

```
Address    Data in Memory                    Meaning
0x00 (0)   00000000000000000001100000010100  LIS [$3]
0x04 (4)   00000000000000000010010000011100  0x241c
0x08 (8)   00000000010000100001000000100000  Add [$2 = $2 + $2]
0x0c (12)  00000000010000100001000000100000  Add [$2 = $2 + $2]
0x10 (16)  00000000001000100000100000100000  Add [$1 = $1 + $2]
0x14 (20)  10101100001000111111111111111100  SW  [$3 → $1 – 4]
0x18 (24)  00000011111000000000000000001000  JR  [$31]
```

Example: A Strange Input

- Suppose $1 = 0 and $2 = 7 initially when we run this program. What is going to happen?

```
Address   Data in Memory                      Meaning
0x00 (0)  00000000000000000001100000010100    LIS [$3]
0x04 (4)  00000000000000000010010000011100    0x241c
0x08 (8)  00000000010000100001000000100000    Add [$2 = $2 + $2]
0x0c (12) 00000000010000100001000000100000    Add [$2 = $2 + $2]
0x10 (16) 00000000001000100000100000100000    Add [$1 = $1 + $2]
0x14 (20) 10101100001000111111111111111100    SW  [$3 → $1 - 4]
0x18 (24) 00000011111000000000000000001000    JR  [$31]
```

- Let's focus on when PC is at the red line (right before the Store Word instruction is executed).

Example: A Strange Input

- Suppose \$1 = 0 and \$2 = 7 initially when we run this program. What is going to happen?

```
Address    Data in Memory                      Meaning
0x00 (0)   00000000000000000001100000010100   LIS [$3]
0x04 (4)   00000000000000000010010000011100   0x241c
0x08 (8)   00000000010000100001000000100000   Add [$2 = $2 + $2]
0x0c (12)  00000000010000100001000000100000   Add [$2 = $2 + $2]
0x10 (16)  00000000001000100000100000100000   Add [$1 = $1 + $2]
0x14 (20)  10101100001000111111111111111100   SW  [$3 → $1 - 4]
0x18 (24)  00000011111000000000000000001000   JR  [$31]
```

- We fetch the instruction and PC moves to the next line. When we execute the instruction, it will overwrite MEM[\$1 – 4] with \$3.

Example: A Strange Input

- Suppose $1 = 0 and $2 = 7 initially when we run this program. What is going to happen?

Address	Data in Memory	Meaning
0x00 (0)	00000000000000000001100000010100	*LIS [$3]*
0x04 (4)	00000000000000000010010000011100	*0x241c*
0x08 (8)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x0c (12)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x10 (16)	**00000000001000100000100000100000**	*Add [$1 = $1 + $2]*
0x14 (20)	10101100001000111111111111111100	*SW [$3 → $1 - 4]*
0x18 (24)	00000011111000000000000000001000	*JR [$31]*

- What is $1? **Earlier on** we added the value of $2 to it. Since $1 was initially 0, now **$1 = $2** when the SW instruction executes.

Example: A Strange Input

- Suppose $1 = 0 and $2 = 7 initially when we run this program. What is going to happen?

Address	Data in Memory	Meaning
0x00 (0)	00000000000000000001100000010100	*LIS [$3]*
0x04 (4)	00000000000000000010010000011100	*0x241c*
0x08 (8)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x0c (12)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x10 (16)	00000000001000100000100000100000	*Add [$1 = $1 + $2]*
0x14 (20)	10101100001000111111111111111100	*SW [$3 → $1 - 4]*
0x18 (24)	00000011111000000000000000001000	*JR [$31]*

- What is $2? **Earlier on** we added $2 to itself twice. Since the initial value was 7, the final value is **28** or 0x1c in hexadecimal.

Example: A Strange Input

- Suppose \$1 = 0 and \$2 = 7 initially when we run this program. What is going to happen?

Address	Data in Memory	Meaning
0x00 (0)	00000000000000000001100000010100	*LIS [$3]*
0x04 (4)	00000000000000000010010000011100	*0x241c*
0x08 (8)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x0c (12)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x10 (16)	00000000001000100000100000100000	*Add [$1 = $1 + $2]*
0x14 (20)	10101100001000111111111111111100	*SW [$3 → $1 - 4]*
0x18 (24)	00000011111000000000000000001000	*JR [$31]*

- So \$1 = \$2 = 28 (0x1c) when the SW executes, and therefore it will modify memory address 28 – 4 = 24 (0x18).

Example: A Strange Input

- Suppose \$1 = 0 and \$2 = 7 initially when we run this program. What is going to happen?

Address	Data in Memory	Meaning
0x00 (0)	00000000000000000001100000010100	*LIS [\$3]*
0x04 (4)	00000000000000000010010000011100	*0x241c*
0x08 (8)	00000000010000100001000000100000	*Add [\$2 = \$2 + \$2]*
0x0c (12)	00000000010000100001000000100000	*Add [\$2 = \$2 + \$2]*
0x10 (16)	00000000001000100000100000100000	*Add [\$1 = \$1 + \$2]*
0x14 (20)	10101100001000111111111111111100	*SW [\$3 → \$1 - 4]*
0x18 (24)	<u>00000011111000000000000000001000</u>	*JR [\$31]*

- Let's now see what happens when we execute the Store Word instruction that places the value in \$3 at address 0x18.

Self-Modifying Code!

- Suppose $1 = 0 and $2 = 7 initially when we run this program. What is going to happen?

Address	Data in Memory	Meaning
0x00 (0)	00000000000000000001100000010100	*LIS [$3]*
0x04 (4)	00000000000000000010010000011100	*0x241c*
0x08 (8)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x0c (12)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x10 (16)	00000000001000100000100000100000	*Add [$1 = $1 + $2]*
0x14 (20)	10101100001000111111111111111100	*SW [$3 → $1 – 4]*
0x18 (24)	00000000000000000010010000011100	*0x241c*

- **The program code itself was modified!** The Jump Register instruction has been overwritten with the value 0x241c.

Self-Modifying Code!

- Suppose $1 = 0 and $2 = 7 initially when we run this program. What is going to happen?

Address	Data in Memory	Meaning
0x00 (0)	00000000000000000001100000010100	*LIS [$3]*
0x04 (4)	00000000000000000010010000011100	*0x241c*
0x08 (8)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x0c (12)	00000000010000100001000000100000	*Add [$2 = $2 + $2]*
0x10 (16)	00000000001000100000100000100000	*Add [$1 = $1 + $2]*
0x14 (20)	10101100001000111111111111111100	*SW [$3 → $1 - 4]*
0x18 (24)	**00000000000000000010010000011100**	*0x241c*

- Note that PC is **still at this line** waiting to fetch and execute the next instruction.

Self-Modifying Code!

- Suppose $1 = 0 and $2 = 7 initially when we run this program. What is going to happen?

```
Address    Data in Memory                     Meaning
0x00 (0)   00000000000000000001100000010100   LIS [$3]
0x04 (4)   00000000000000000010010000011100   0x241c
0x08 (8)   00000000010000100001000000100000   Add [$2 = $2 + $2]
0x0c (12)  00000000010000100001000000100000   Add [$2 = $2 + $2]
0x10 (16)  00000000001000100000100000100000   Add [$1 = $1 + $2]
0x14 (20)  10101100001000111111111111111100   SW  [$3 → $1 - 4]
0x18 (24)  00000000000000000010010000011100   0x241c
```

- The next thing the MIPS machine will do is try to fetch and execute 0x241c. This isn't a valid instruction so the program crashes.

Self-Modifying Code!

- Suppose $1 = 0 and $2 = 7 initially when we run this program. What is going to happen?

```
Address    Data in Memory                      Meaning
0x00 (0)   00000000000000000001100000010100  LIS [$3]
0x04 (4)   00000000000000000010010000011100  0x241c
0x08 (8)   00000000010000100001000000100000  Add [$2 = $2 + $2]
0x0c (12)  00000000010000100001000000100000  Add [$2 = $2 + $2]
0x10 (16)  00000000001000100000100000100000  Add [$1 = $1 + $2]
0x14 (20)  10101100001000111111111111111100  SW  [$3 → $1 - 4]
0x18 (24)  00000000000000000010010000011100  0x241c
```

- **Code is just another form of data in memory.** Our program interpreted *itself* as an “array” and modified its own “last element”.

A Better Way?

```
00000000000000000001100000010100  LIS [$3]
00000000000000000010010000011100  0x241c
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000010000100001000000100000  Add [$2 = $2 + $2]
00000000001000100000100000100000  Add [$1 = $1 + $2]
10101100001000111111111111111100  SW  [$3 → $1 – 4]
00000011111000000000000000001000  JR  [$31]
```

- After coming up with our “plan” on the right, translating it into machine language was a straightforward but very tedious process.
- We’ll put our study of machine language on hold for now, and instead try to find a way to **automate** this translation process so that we can write more interesting and complicated programs.

MIPS Assembly Language

- An **assembly language** is a text representation of a machine language.
- MIPS machine language has a corresponding assembly language. It looks quite similar to the “program plan” we came up with, but is a little different syntactically.

MIPS Machine Language	**MIPS Assembly Language**
`00000000000000000001100000010100`	`lis $3`
`00000000000000000010010000011100`	`.word 0x241c`
`00000000010000100001000000100000`	`add $2, $2, $2`
`00000000010000100001000000100000`	`add $2, $2, $2`
`00000000001000100000100000100000`	`add $1, $1, $2`
`10101100001000111111111111111100`	`sw $3, -4($1)`
`00000011111000000000000000001000`	`jr $31`

Moving Forward: Writing an Assembler

- A program that translates assembly language to machine language is called an **assembler**. Having an assembler will make writing programs much more convenient.
- Unfortunately, writing an assembler is non-trivial and we'll need several more lectures to explain how it's done.
- The process of putting together the sequences of bits is not that bad. The difficult part is actually **string processing:**

  `"add $1, $1, $2"` is really `['a', 'd', 'd', ' ', '$', '1', ',', ' ', ...]`

- Our next topic will be **scanning**, a technique for grouping sequences of characters into meaningful chunks of data.