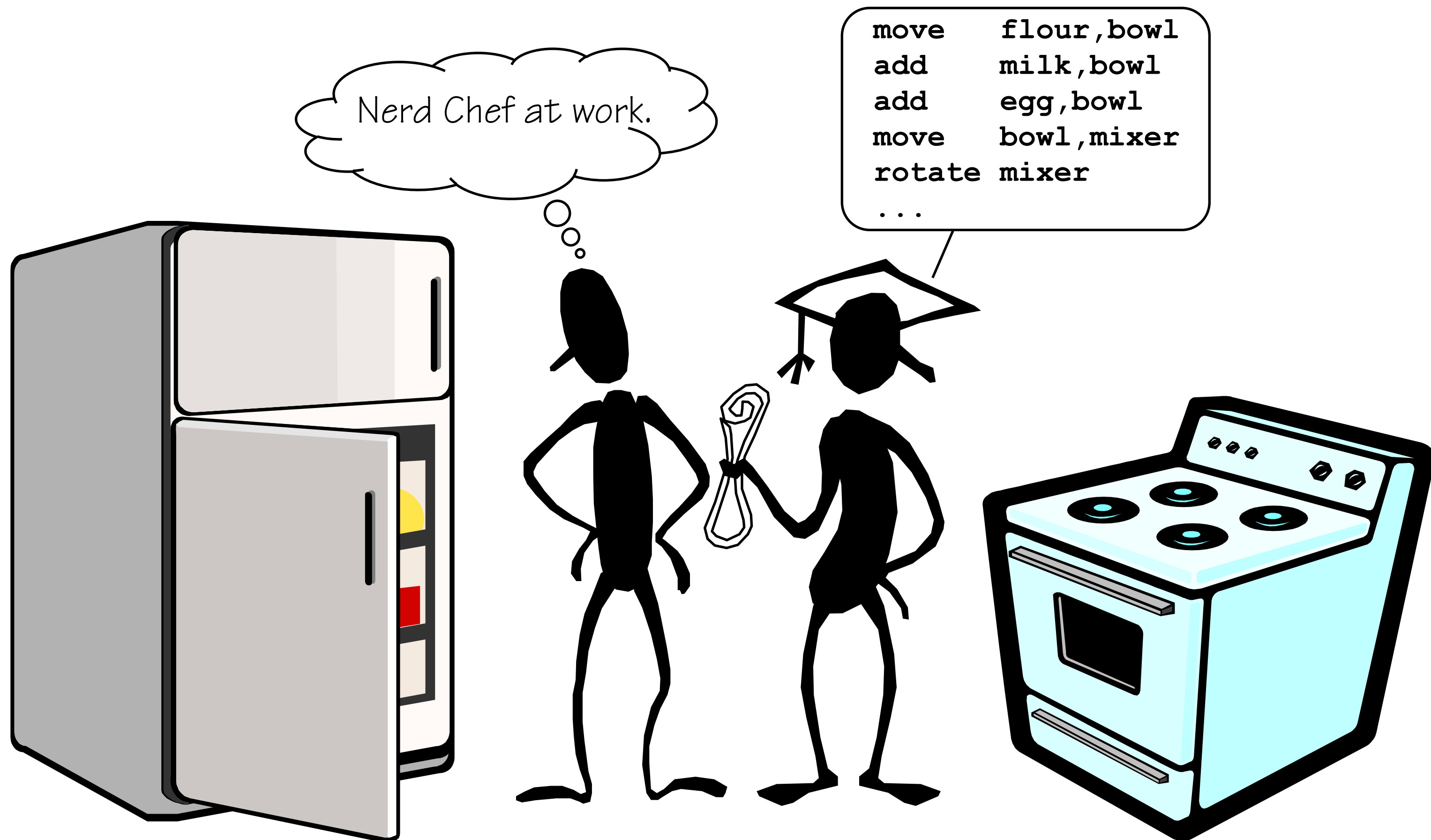
Concocting an Instruction Set



Read: Chapter 2.1-2.7

A General-Purpose Computer

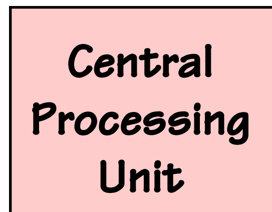
The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one on which nearly all modern, practical computers is based was proposed by John von Neumann in the late 1940s. Its major components are:

A General-Purpose Computer

The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one on which nearly all modern, practical computers is based was proposed by John von Neumann in the late 1940s. Its major components are:

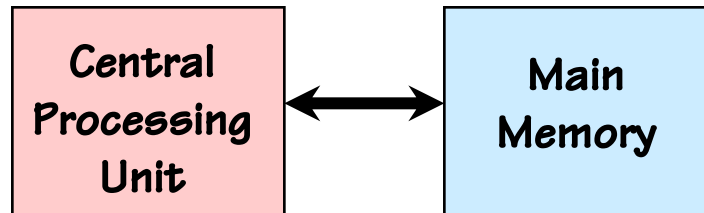


Central Processing Unit (CPU): A device which fetches, interprets, and executes a specified set of operations called Instructions.

A General-Purpose Computer

The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one on which nearly all modern, practical computers is based was proposed by John von Neumann in the late 1940s. Its major components are:

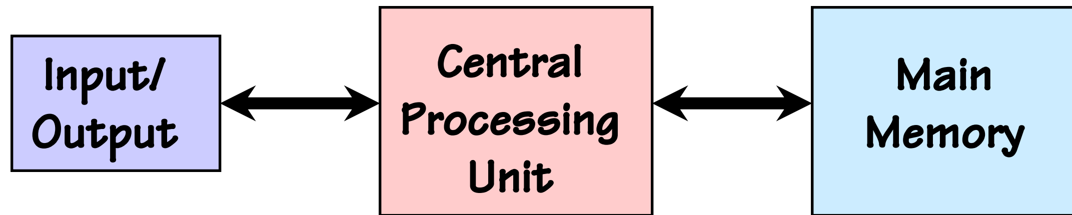


Central Processing Unit (CPU): A device which fetches, interprets, and executes a specified set of operations called **Instructions**.

Memory: storage of N *words* of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.

A General-Purpose Computer
The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one on which nearly all modern, practical computers is based was proposed by John von Neumann in the late 1940s. Its major components are:



Central Processing Unit (CPU): A device which fetches, interprets, and executes a specified set of operations called **Instructions**.

Memory: storage of N *words* of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.

I/O: Devices for communicating with the outside world.

Anatomy of an Instruction

Anatomy of an Instruction

```
add  $t0, $t1, $t2
```

Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination

```
add  $t0, $t1, $t2
```

Anatomy of an Instruction

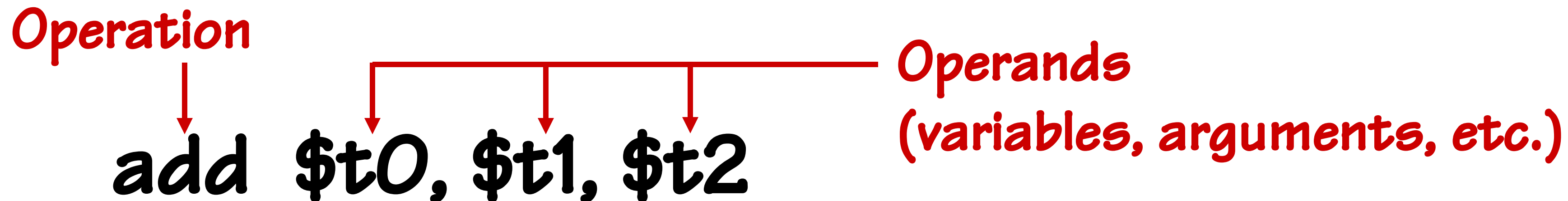
- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination

Operation

```
add  $t0, $t1, $t2
```

Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination



Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination

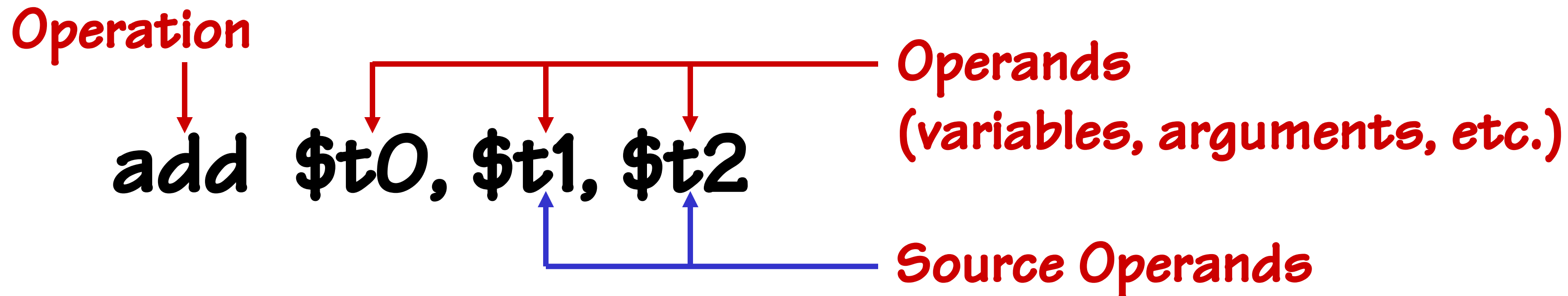
Operation

```
add  $t0, $t1, $t2
```

Operands (variables, arguments, etc.)

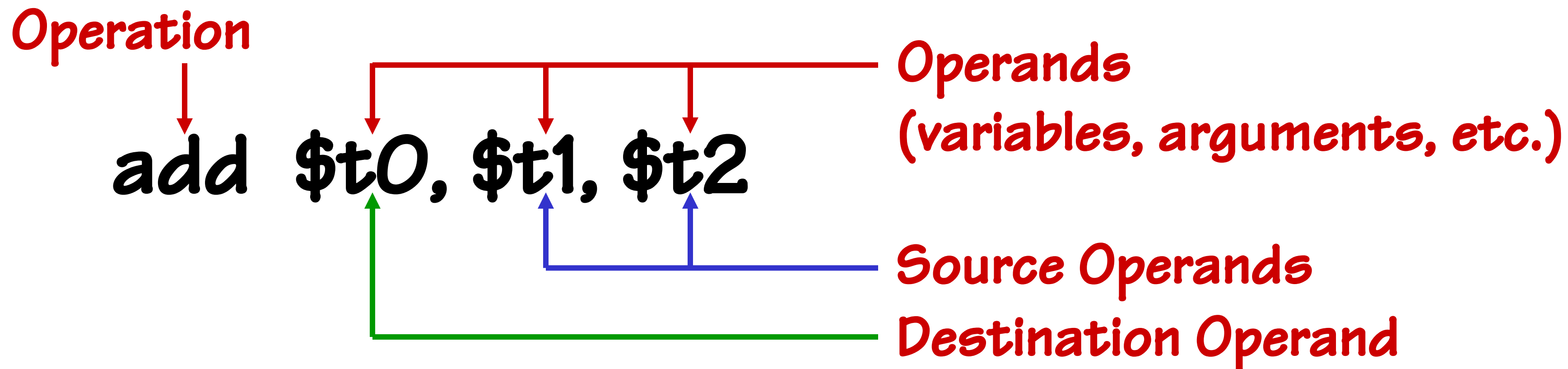
Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination



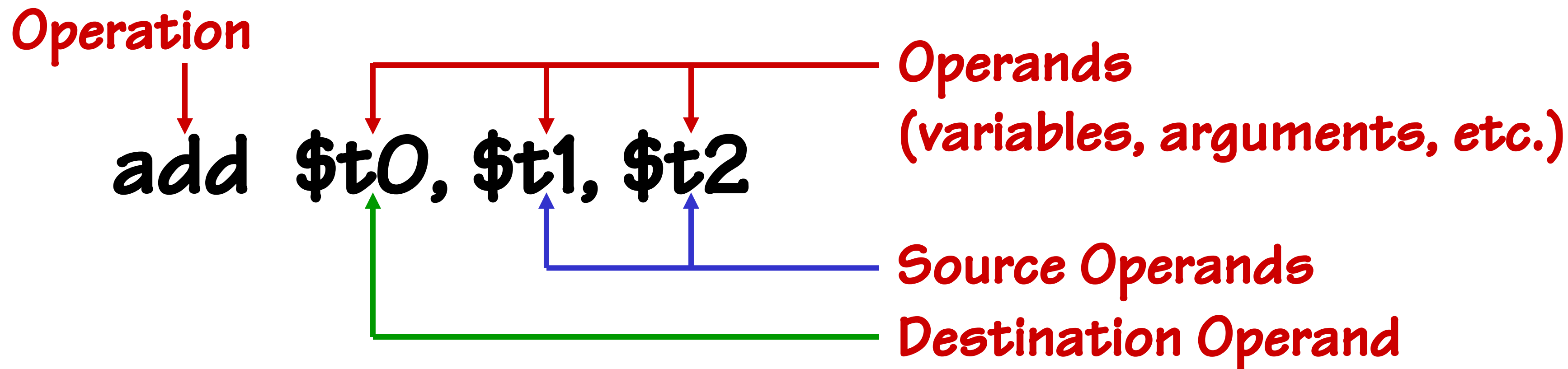
Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination



Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination



addi $t0, $t1, 1

Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination

Operation

Operands (variables, arguments, etc.)

```
add  $t0, $t1, $t2
```

Source Operands

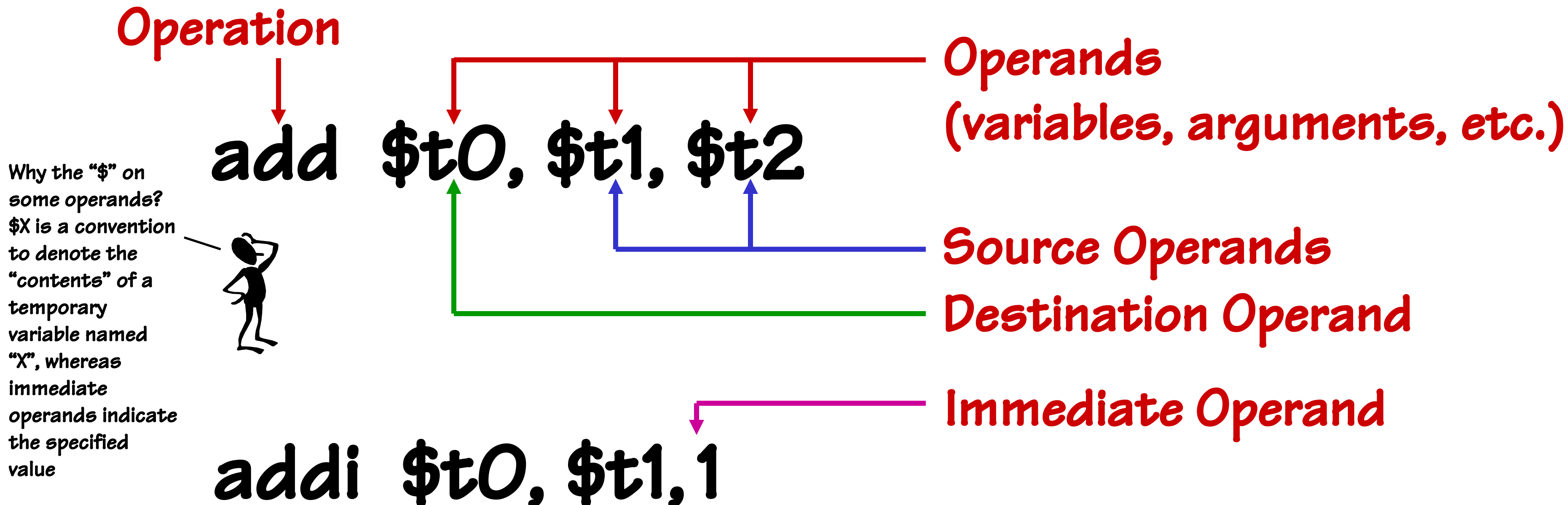
Destination Operand

Immediate Operand

```
addi  $t0, $t1, 1
```

Anatomy of an Instruction

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination



Meaning of an Instruction

- Operations are abbreviated into opcodes (1-4 letters)
- Instructions are specified with a very regular syntax
  - First an opcode followed by arguments
  - Usually the destination is next, then source arguments (This is not strictly the case, but it is generally true)
  - Why this order?
- Analogy to high-level language like Java or C

Meaning of an Instruction

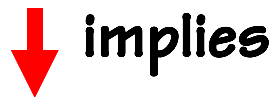
- Operations are abbreviated into opcodes (1-4 letters)
- Instructions are specified with a very regular syntax
  - First an opcode followed by arguments
  - Usually the destination is next, then source arguments (This is not strictly the case, but it is generally true)
  - Why this order?
- Analogy to high-level language like Java or C

```
add  $t0, $t1, $t2
```

Meaning of an Instruction

- Operations are abbreviated into opcodes (1-4 letters)
- Instructions are specified with a very regular syntax
  - First an opcode followed by arguments
  - Usually the destination is next, then source arguments (This is not strictly the case, but it is generally true)
  - Why this order?
- Analogy to high-level language like Java or C

```
add  $t0, $t1, $t2
```




```
int t0, t1, t2
t0 = t1 + t2
```

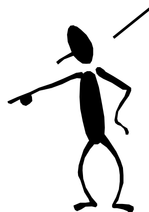
Meaning of an Instruction

- Operations are abbreviated into opcodes (1-4 letters)
- Instructions are specified with a very regular syntax
  - First an opcode followed by arguments
  - Usually the destination is next, then source arguments (This is not strictly the case, but it is generally true)
  - Why this order?
- Analogy to high-level language like Java or C

```
add  $t0, $t1, $t2
```

implies

```
int t0, t1, t2
t0 = t1 + t2
```



The instruction syntax provides operands in the same order as you would expect in a statement from a high level language.

Being the Machine!

- **Generally…**
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules

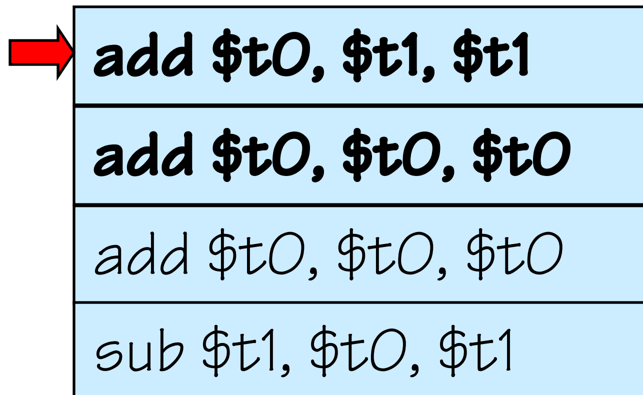
Instructions
add $t0, $t1, $t1
add $t0, $t0, $t0
add $t0, $t0, $t0
sub $t1, $t0, $t1

Variables	
$t0:	0
$t1:	6
$t2:	8
$t3:	10

Being the Machine!

- **Generally…**
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules

Instructions



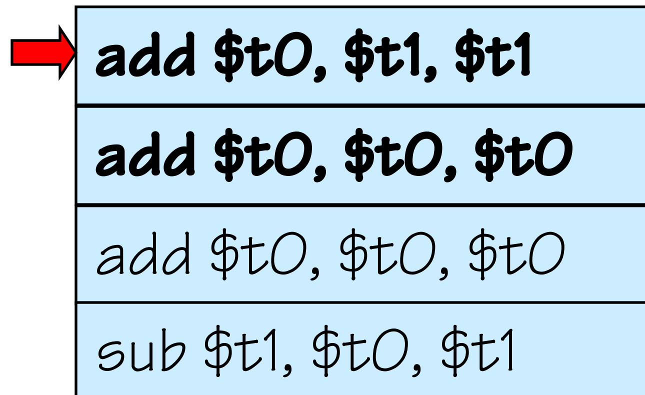
Variables

$t0:	0
$t1:	6
$t2:	8
$t3:	10

Being the Machine!

- **Generally…**
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules

Instructions



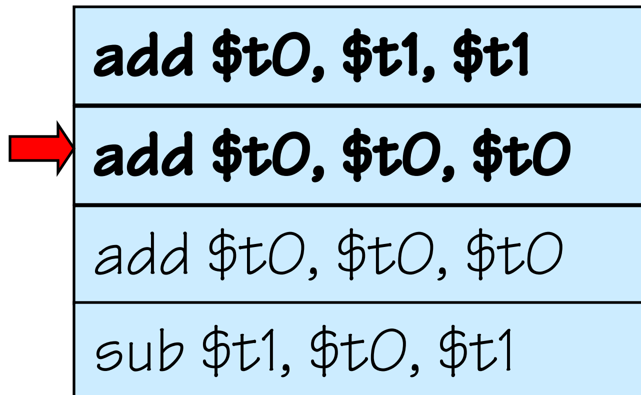
Variables

$t0:	~~0~~ 12
$t1:	6
$t2:	8
$t3:	10

Being the Machine!

- Generally…
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules

Instructions

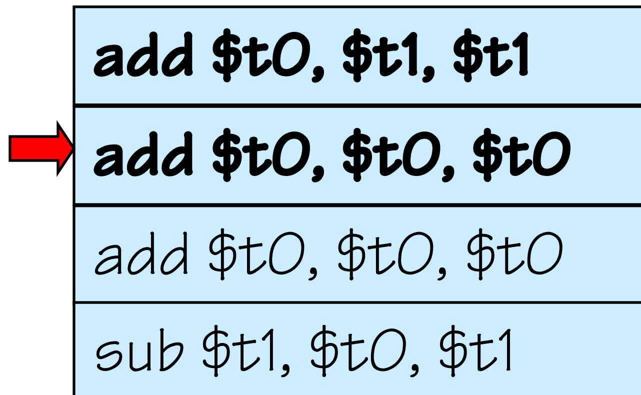


Variables

$t0:	~~0~~ 12
$t1:	6
$t2:	8
$t3:	10

Being the Machine!

- Generally…
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules



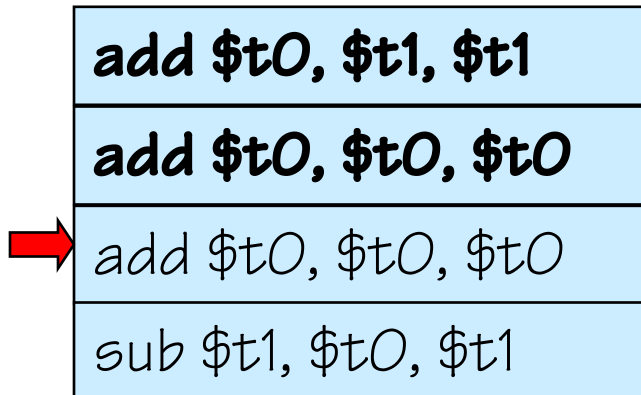
Variables

$t0:	~~0~~ ~~12~~ 24
$t1:	6
$t2:	8
$t3:	10

Being the Machine!

- Generally…
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules

Instructions

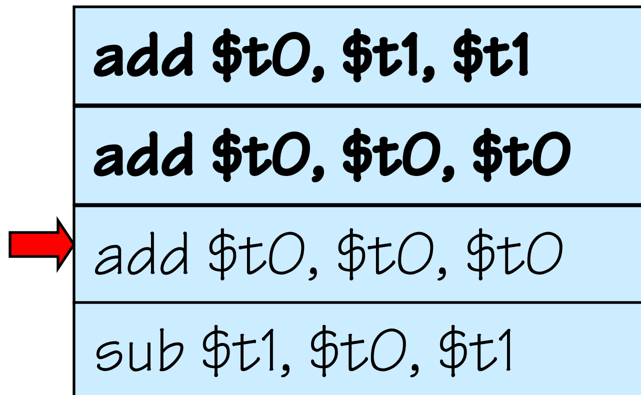


Variables

$t0:	~~0~~ ~~12~~ 24
$t1:	6
$t2:	8
$t3:	10

Being the Machine!

- Generally…
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules



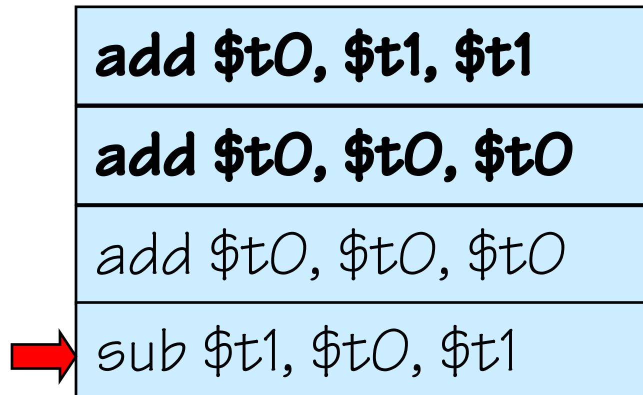
Variables

$t0:	~~0~~ ~~12~~ ~~24~~ 48
$t1:	6
$t2:	8
$t3:	10

Being the Machine!

- Generally…
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules

Instructions



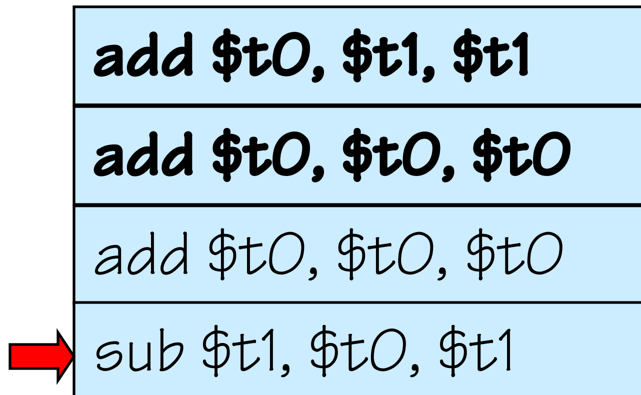
add $t0, $t1, $t1
add $t0, $t0, $t0
add $t0, $t0, $t0
sub $t1, $t0, $t1

Variables

$t0:	~~0~~ ~~12~~ ~~24~~ 48
$t1:	6
$t2:	8
$t3:	10

Being the Machine!

- Generally…
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules

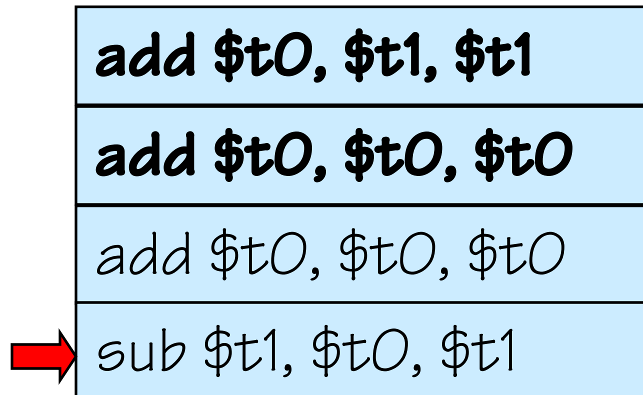


Variables

$t0:	~~0~~ ~~12~~ ~~24~~ 48
$t1:	~~6~~ 42
$t2:	8
$t3:	10

Being the Machine!

- **Generally…**
  - Instructions are executed sequentially from a list
  - Instructions execute after all previous instructions have completed, therefore their results are available to the next instruction
  - But, you may see exceptions to these rules



What is this program doing?

Variables

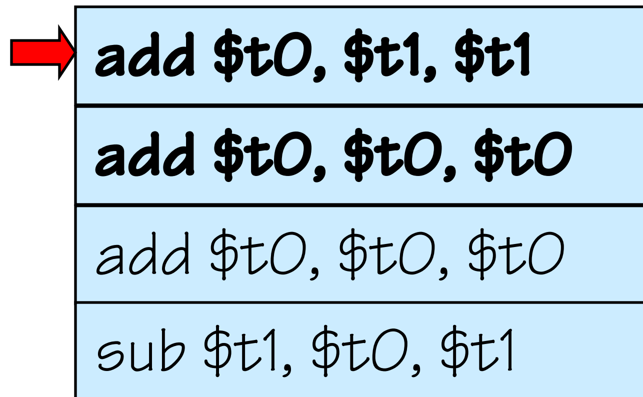
$t0:	~~0~~ ~~12~~ ~~24~~ 48
$t1:	~~6~~ 42
$t2:	8
$t3:	10

Analyzing the Machine!

Instructions
add $t0, $t1, $t1
add $t0, $t0, $t0
add $t0, $t0, $t0
sub $t1, $t0, $t1

Variables	
$t0:	**w**
$t1:	**x**
$t2:	**y**
$t3:	**z**

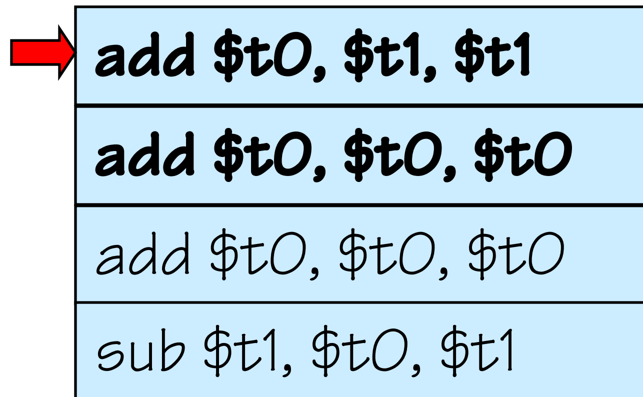
Analyzing the Machine!



Variables

$t0:	**w**
$t1:	**x**
$t2:	**y**
$t3:	**z**

Analyzing the Machine!

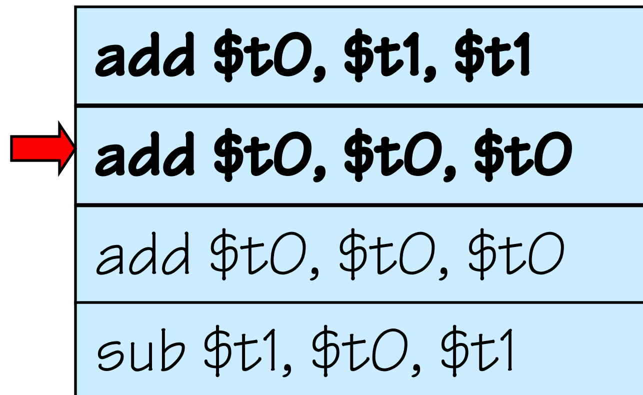


Variables

$t0:	~~w~~ 2x
$t1:	x
$t2:	y
$t3:	z

Analyzing the Machine!

Instructions

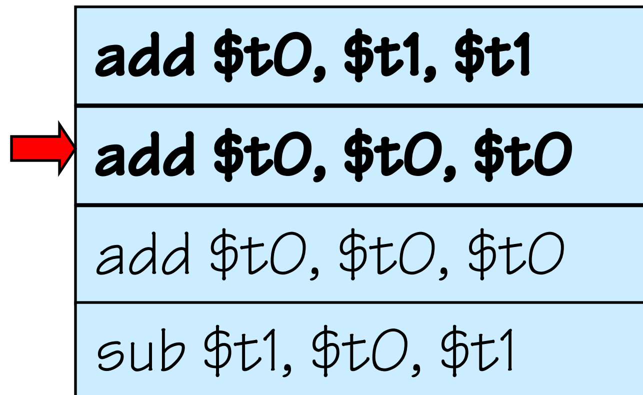


Variables

$t0:	~~w~~ 2x
$t1:	x
$t2:	y
$t3:	z

Analyzing the Machine!

Instructions

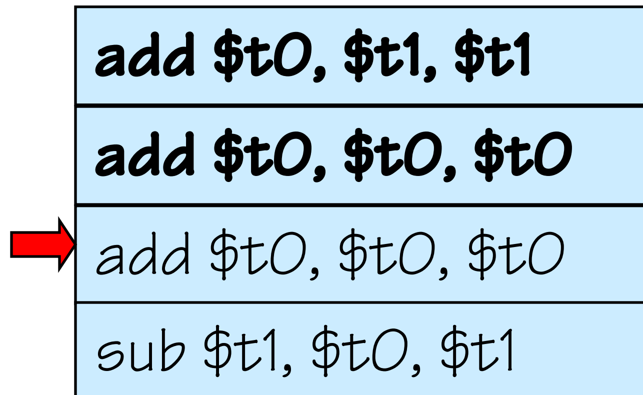


Variables

$t0:	~~w~~ ~~2x~~ 4x
$t1:	x
$t2:	y
$t3:	z

Analyzing the Machine!

Instructions

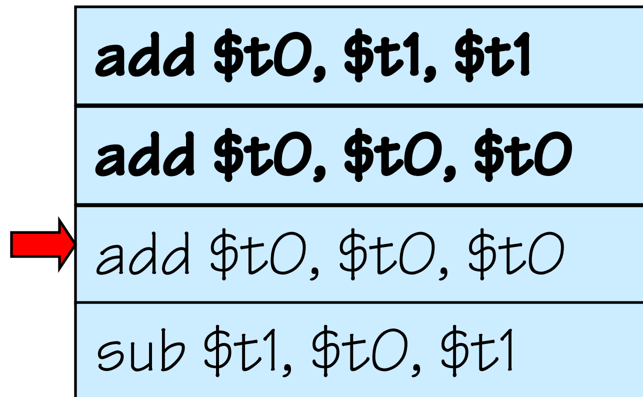


Variables

$t0:	~~w~~ ~~2x~~ 4x
$t1:	x
$t2:	y
$t3:	z

Analyzing the Machine!

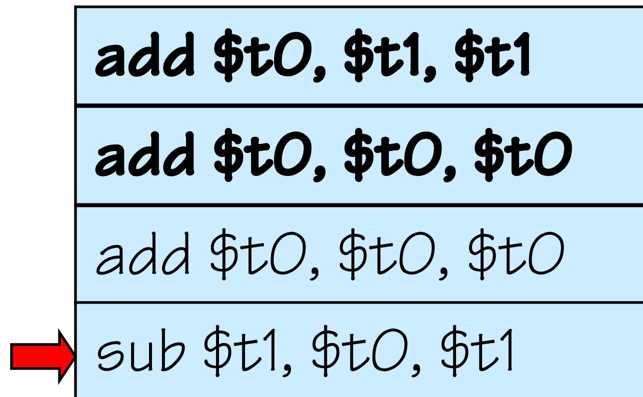
Instructions



Variables

$t0:	~~x~~ ~~2x~~ ~~4x~~ 8x
$t1:	x
$t2:	y
$t3:	z

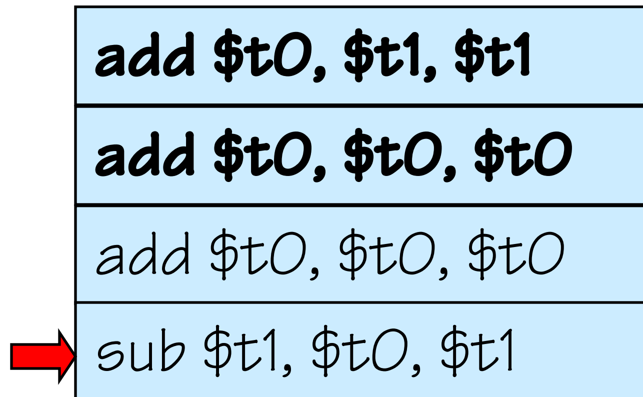
Analyzing the Machine!



Variables

$t0:	~~x~~ ~~2x~~ ~~4x~~ 8x
$t1:	x
$t2:	y
$t3:	z

Analyzing the Machine!

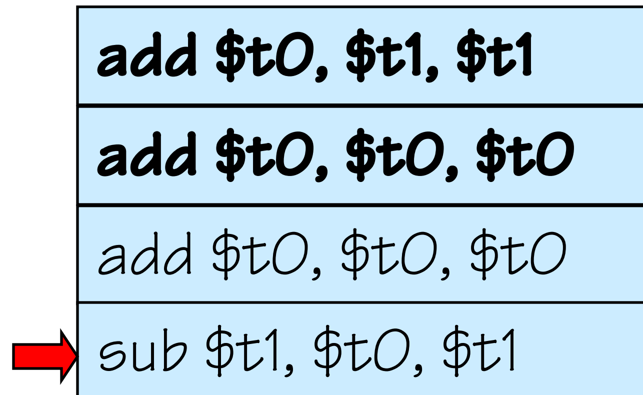


Instructions
add $t0, $t1, $t1
add $t0, $t0, $t0
add $t0, $t0, $t0
sub $t1, $t0, $t1

Variables	
$t0:	~~x~~ ~~2x~~ ~~4x~~ 8x
$t1:	~~x~~ 7x
$t2:	y
$t3:	z

Analyzing the Machine!

- Repeat the process treating the variables as unknowns
- Knowing what the program does allows us to write down its specification, and give it a meaningful name
- The instruction sequence is now a general purpose tool

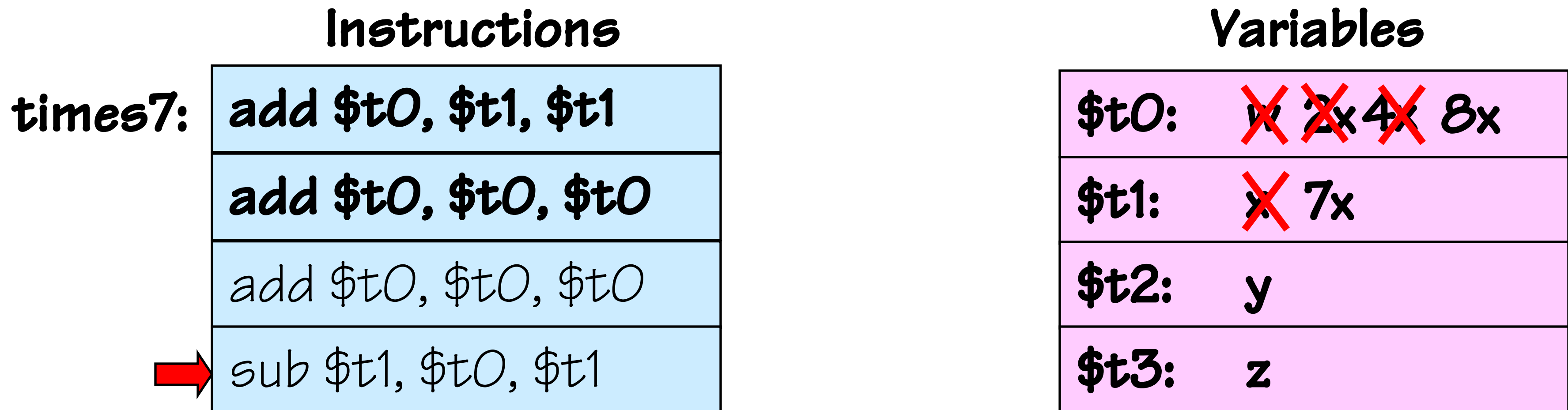


Variables

$t0:	~~x~~ ~~2x~~ ~~4x~~ 8x
$t1:	~~x~~ 7x
$t2:	y
$t3:	z

Analyzing the Machine!

- Repeat the process treating the variables as unknowns
- Knowing what the program does allows us to write down its specification, and give it a meaningful name
- The instruction sequence is now a general purpose tool



Looping the Flow

- Operations to change the flow of sequential execution
- A jump instruction with opcode 'j'
- The operand refers to a label of some other instruction

Instructions

Label	Instruction
times7:	**add $t0, $t1, $t1**
	add $t0, $t0, $t0
	add $t0, $t0, $t0
	sub $t1, $t0, $t1

Variables

Register	Variable
$t0:	w
$t1:	x
$t2:	y
$t3:	z

Looping the Flow

- Operations to change the flow of sequential execution
- A jump instruction with opcode 'j'
- The operand refers to a label of some other instruction

Instructions

Label	Instruction
times7:	**add $t0, $t1, $t1**
	add $t0, $t0, $t0
	add $t0, $t0, $t0
	sub $t1, $t0, $t1
	j times7

Variables

Register	Variable
$t0:	**w**
$t1:	**x**
$t2:	**y**
$t3:	**z**

Looping the Flow

- Operations to change the flow of sequential execution
- A jump instruction with opcode 'j'
- The operand refers to a label of some other instruction

	Instructions
times7:	**add $t0, $t1, $t1**
	add $t0, $t0, $t0
	add $t0, $t0, $t0
	sub $t1, $t0, $t1
	j times7

Variables	
$t0:	~~w~~ 8x
$t1:	~~x~~ 7x
$t2:	y
$t3:	z

Looping the Flow

- Operations to change the flow of sequential execution
- A jump instruction with opcode 'j'
- The operand refers to a label of some other instruction

Instructions

Label	Instruction
times7:	**add $t0, $t1, $t1**
	add $t0, $t0, $t0
	add $t0, $t0, $t0
	sub $t1, $t0, $t1
	j times7

Variables

Register	Value
$t0:	~~x~~ ~~8x~~ 56x
$t1:	~~x~~ ~~7x~~ 49x
$t2:	y
$t3:	z

Looping the Flow

- Operations to change the flow of sequential execution
- A jump instruction with opcode 'j'
- The operand refers to a label of some other instruction

	Instructions
times7:	**add $t0, $t1, $t1**
	add $t0, $t0, $t0
	add $t0, $t0, $t0
	sub $t1, $t0, $t1
	j times7

Variables	
$t0:	~~w~~ ~~8x~~ ~~56x~~ 392x
$t1:	~~x~~ ~~7x~~ ~~49x~~ 343x
$t2:	y
$t3:	z

Looping the Flow

- Operations to change the flow of sequential execution
- A jump instruction with opcode 'j'
- The operand refers to a label of some other instruction

Instructions

times7:	**add $t0, $t1, $t1**
	add $t0, $t0, $t0
	add $t0, $t0, $t0
	sub $t1, $t0, $t1
	j times7

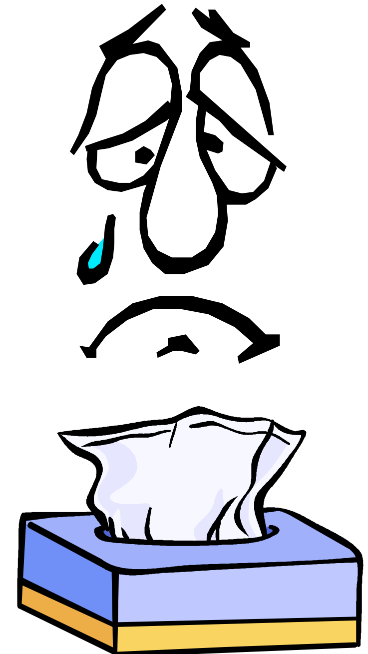
An infinite loop

Variables

$t0:	~~w~~ ~~8x~~ ~~56x~~ 392x
$t1:	~~x~~ ~~7x~~ ~~49x~~ 343x
$t2:	y
$t3:	z

Open Issues in our Simple Model

- WHERE are INSTRUCTIONS stored?
- HOW are instructions represented?
- WHERE are VARIABLES stored?
- How are labels associated with particular instructions?
- How do you access more complicated variable types like
  - Arrays?
  - Structures?
  - Objects?
- Where does a program start executing?
- How does it stop?



The Stored-Program Computer

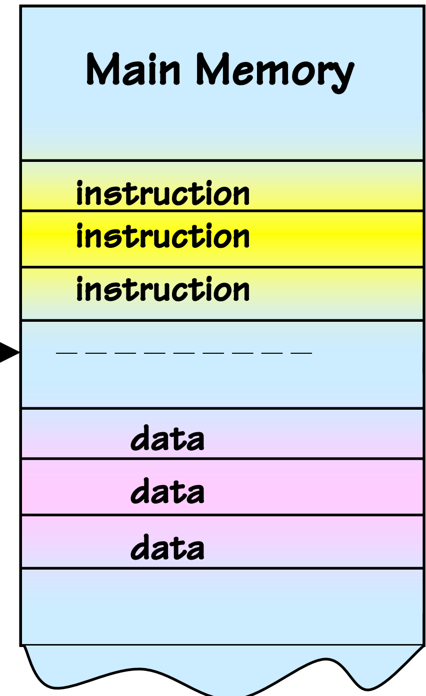
- The von Neumann architecture addresses these issues of our simple programmable machine example:
- Instructions and Data are stored in a common memory
- Sequential semantics: To the programmer all instructions appear to be executed sequentially

The Stored-Program Computer

- The von Neumann architecture addresses these issues of our simple programmable machine example:
- Instructions and Data are stored in a common memory
- Sequential semantics: To the programmer all instructions appear to be executed sequentially

Key idea: Memory holds not only data, but *coded instructions* that make up a *program*.

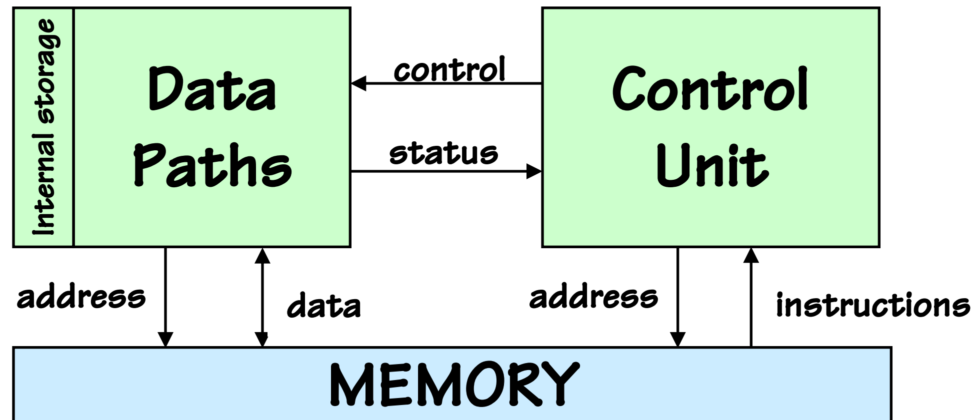
Central Processing Unit



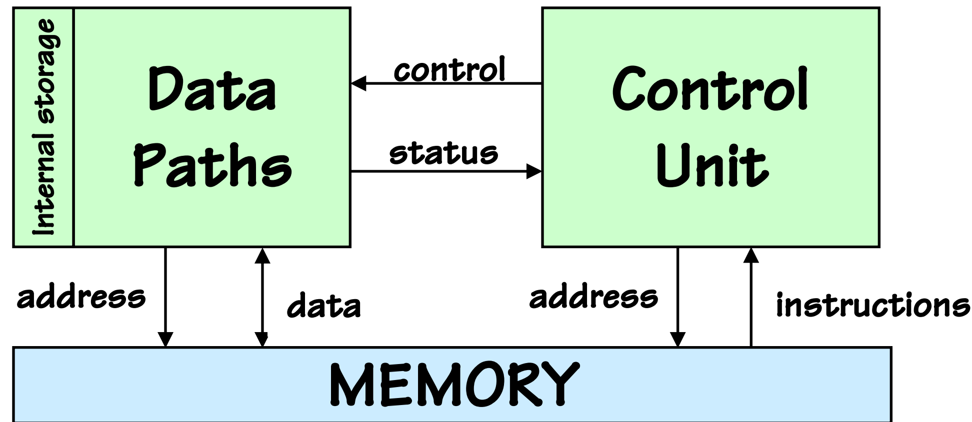
CPU fetches and executes instructions from memory ...

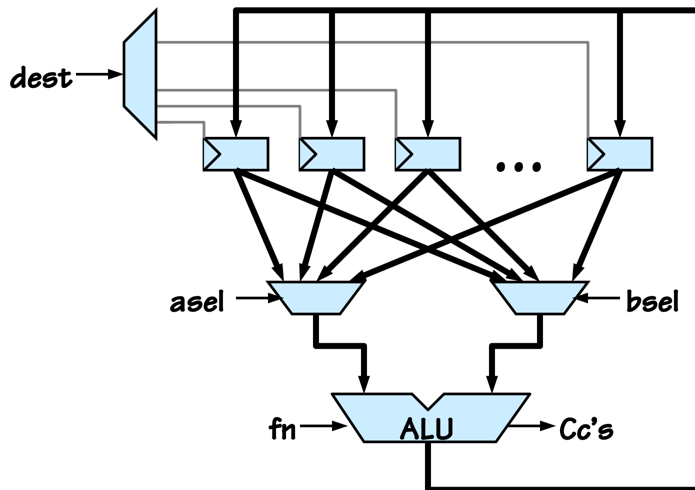
- The CPU is a H/W interpreter
- Program IS simply data for this interpreter
- Main memory: Single expandable resource pool
  - constrains both data and program size
  - don't need to make separate decisions of how large of a program or data memory to buy

Anatomy of a von Neumann Computer

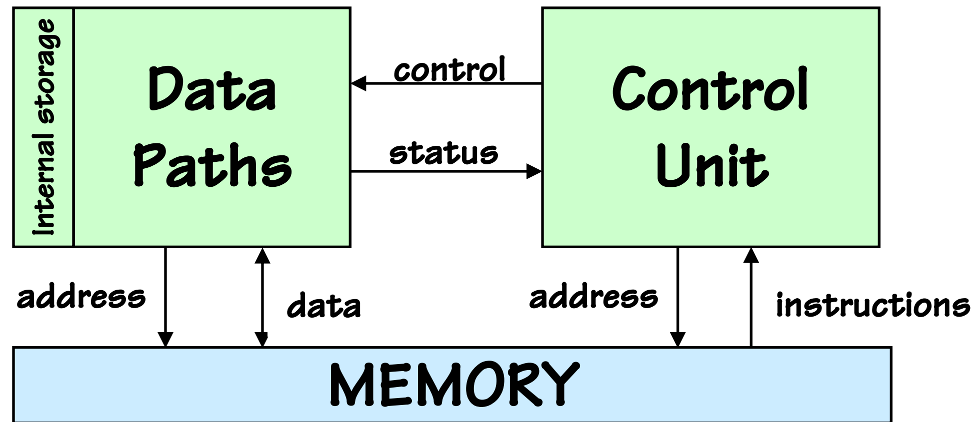


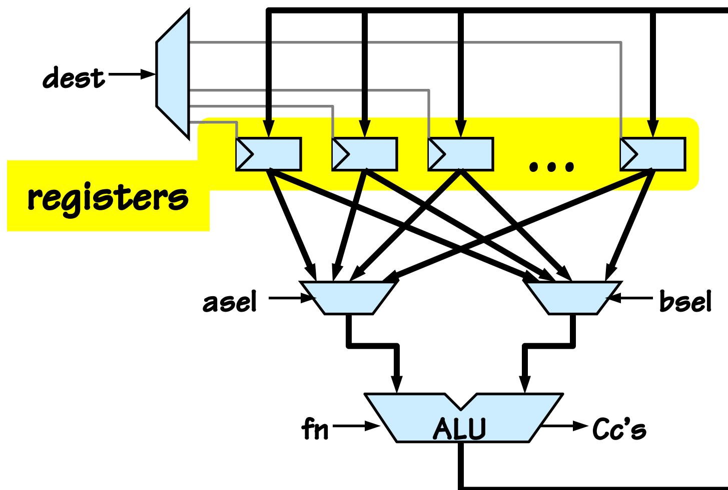
Anatomy of a von Neumann Computer



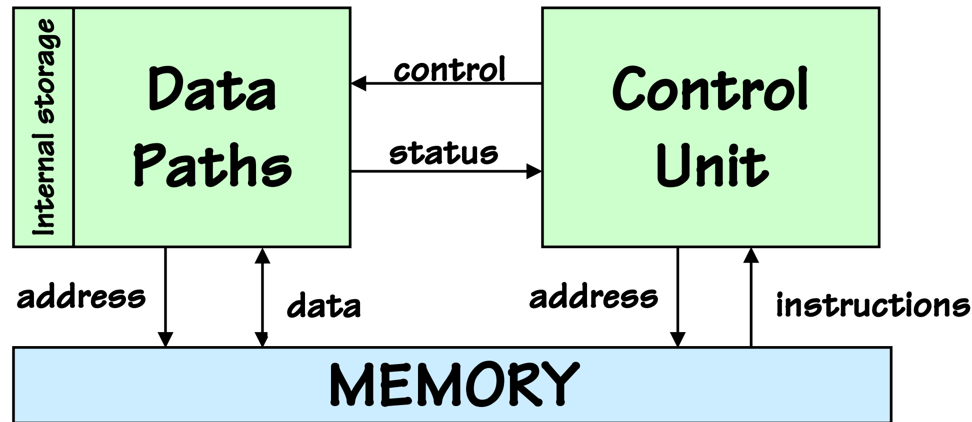


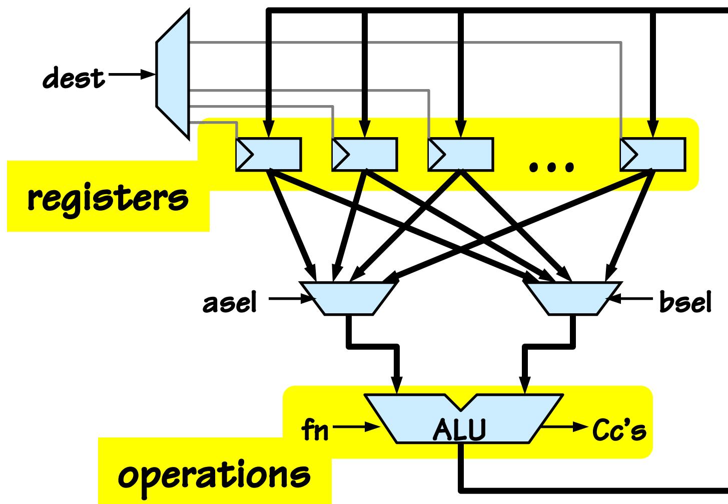
Anatomy of a von Neumann Computer



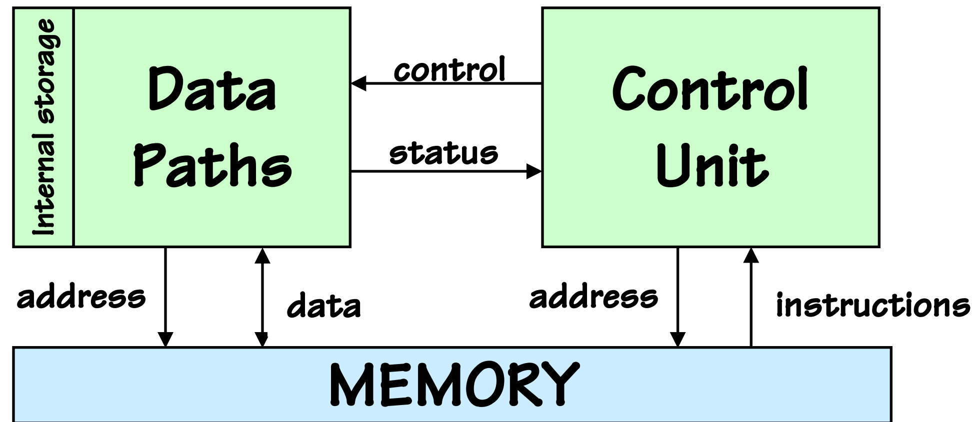


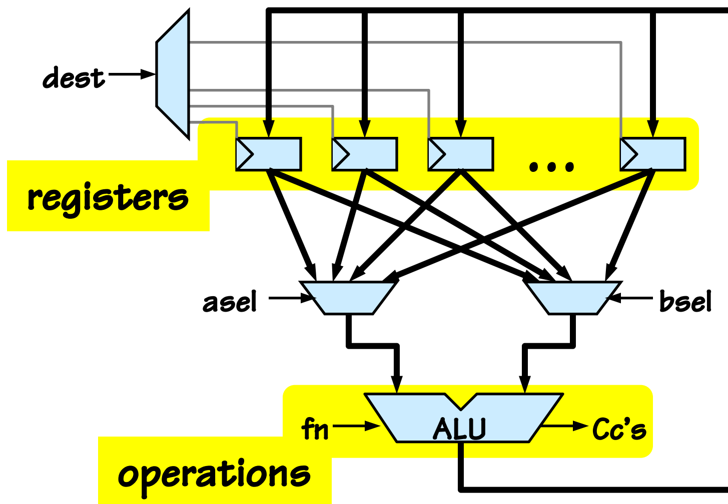
Anatomy of a von Neumann Computer





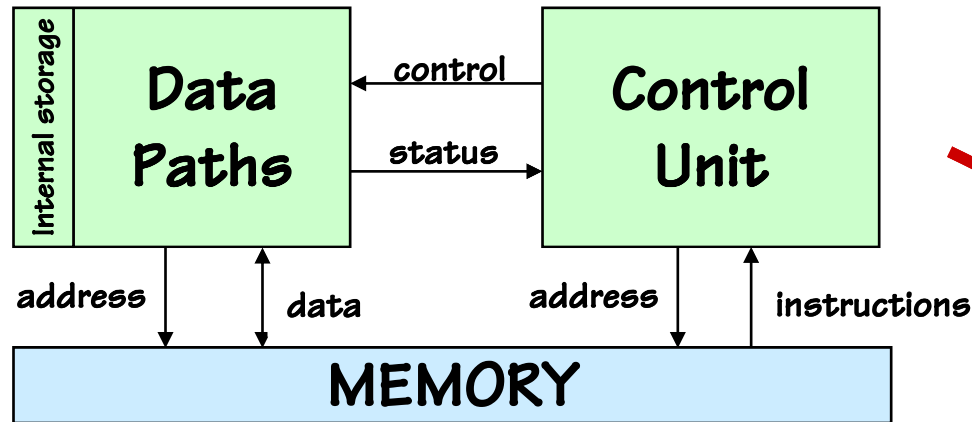
Anatomy of a von Neumann Computer

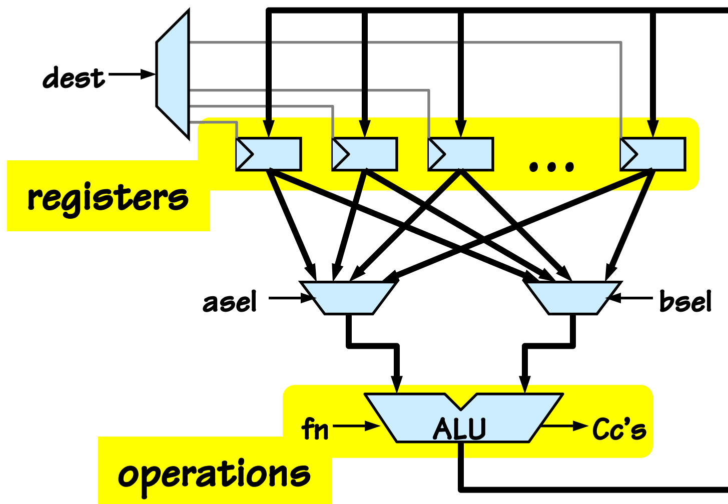


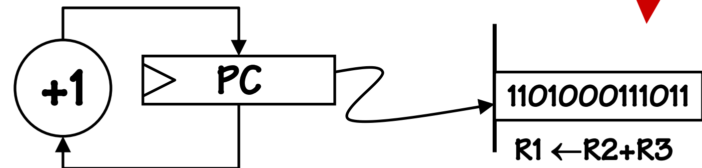


More about this stuff later!

Anatomy of a von Neumann Computer







More about this stuff later!

- INSTRUCTIONS coded as binary data
- PROGRAM COUNTER or PC: Address of next instruction to be executed
- logic to translate instructions into control signals for data path

Instruction Set Architecture (ISA)

Encoding of instructions raises some interesting choices...

- Tradeoffs: performance, compactness, programmability
- Uniformity. Should different instructions
  - Be the same size?
  - Take the same amount of time to execute?
    - Trend: Uniformity. Affords simplicity, speed, pipelining.
- Complexity. How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, “polynomial evaluate”, etc
    - “Reduced Instruction Set Computer” (RISC) philosophy: simple instructions, optimized for speed

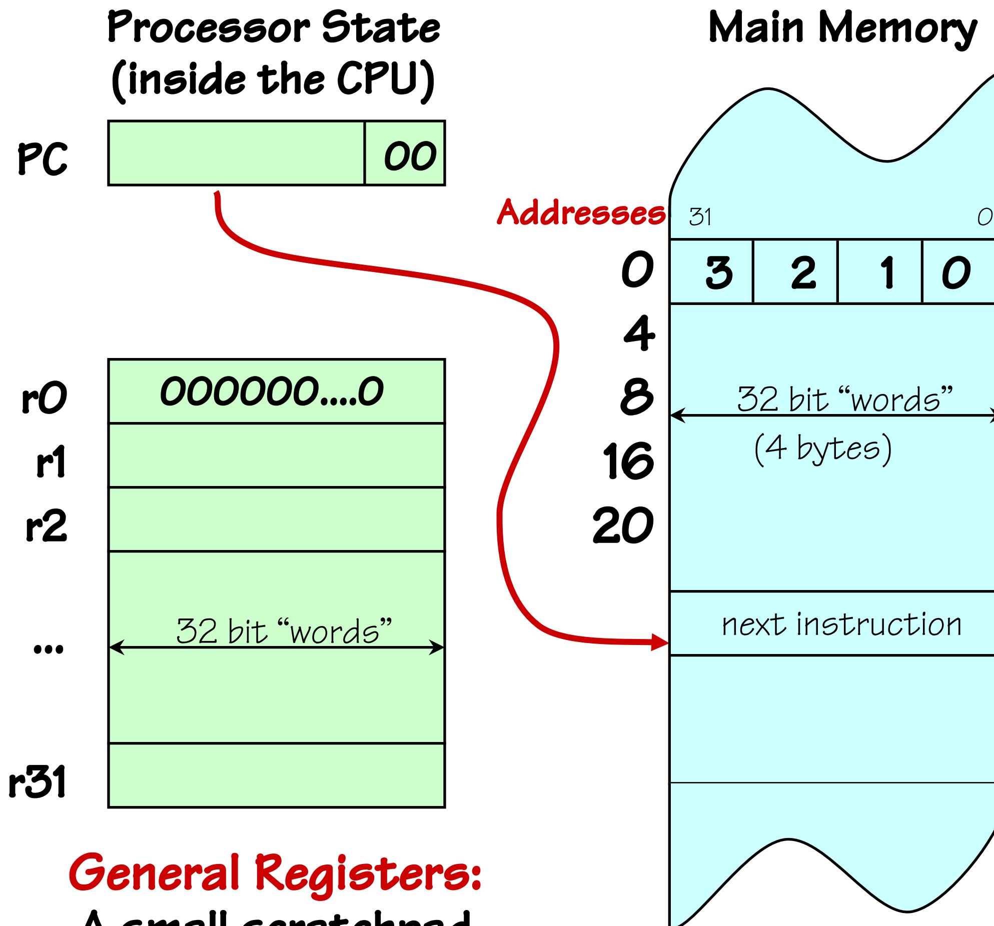
Mix of Engineering & Art...

Trial (by simulation) is our best technique for making choices!

Our representative example: the MIPS architecture!

MIPS Programming Model

a representative simple RISC machine



In Comp 411 we'll use a clean and sufficient subset of the MIPS-32 core Instruction set.

Fetch/Execute loop:

- fetch Mem[PC]
- PC = PC + 4†
- execute fetched instruction (may change PC!)
- repeat!

*†MIPS uses byte memory addresses. However, each instruction is 32-bits wide, and *must* be aligned on a multiple of 4 (word) address. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.*

Some MIPs Memory Nits

- Memory locations are 32 bits wide
  - BUT, they are *addressable* in different-sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/double)
- We also frequently need access to individual bits! (Instructions help to do this)
- Every BYTE has a unique address (MIPS is a byte-addressable machine)
- Every instruction is one word

Addr				
0:	3	2	1	0
4:	7	6	5	4
8:	12	10	9	8
12:	15	14	13	12

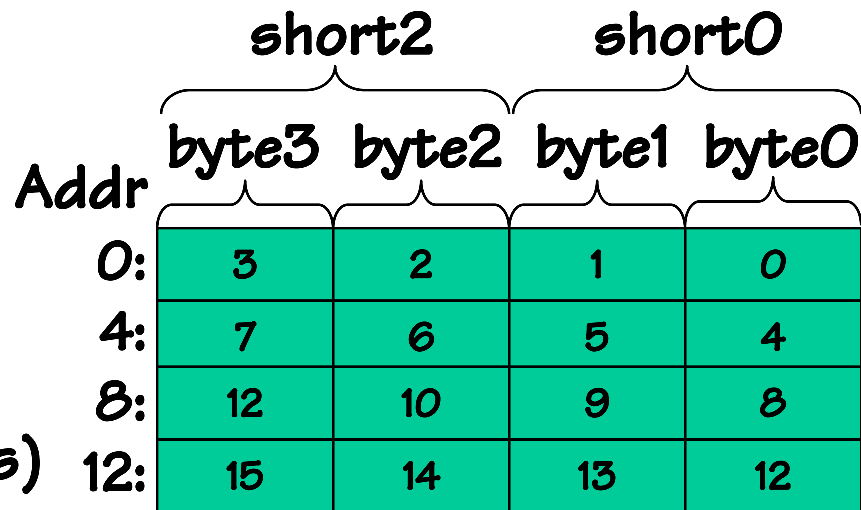
Some MIPs Memory Nits

- Memory locations are 32 bits wide
  - BUT, they are *addressable* in different-sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/double)
- We also frequently need access to individual bits! (Instructions help to do this)
- Every BYTE has a unique address (MIPS is a byte-addressable machine)
- Every instruction is one word

Addr	byte3	byte2	byte1	byte0
0:	3	2	1	0
4:	7	6	5	4
8:	12	10	9	8
12:	15	14	13	12

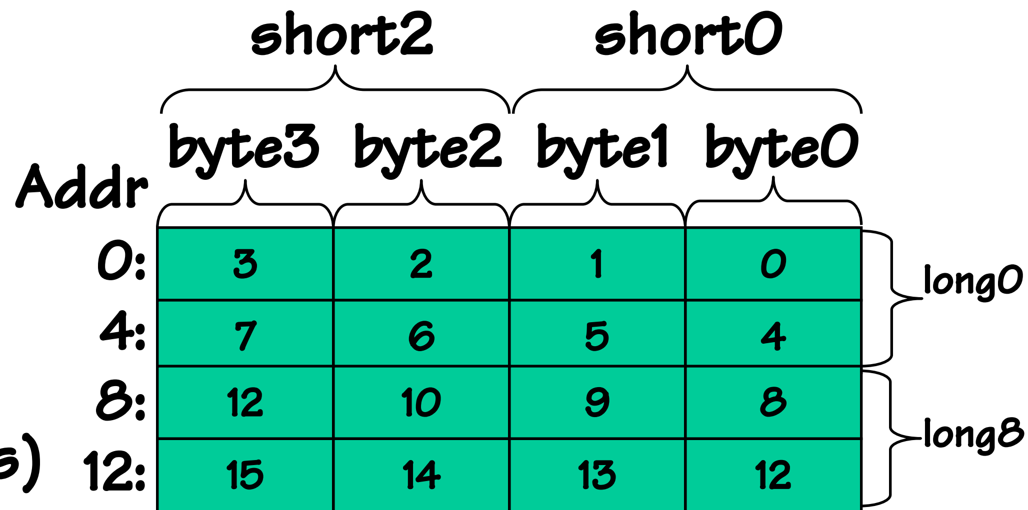
Some MIPs Memory Nits

- Memory locations are 32 bits wide
  - BUT, they are *addressable* in different-sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/double)
- We also frequently need access to individual bits! (Instructions help to do this)
- Every BYTE has a unique address (MIPS is a byte-addressable machine)
- Every instruction is one word



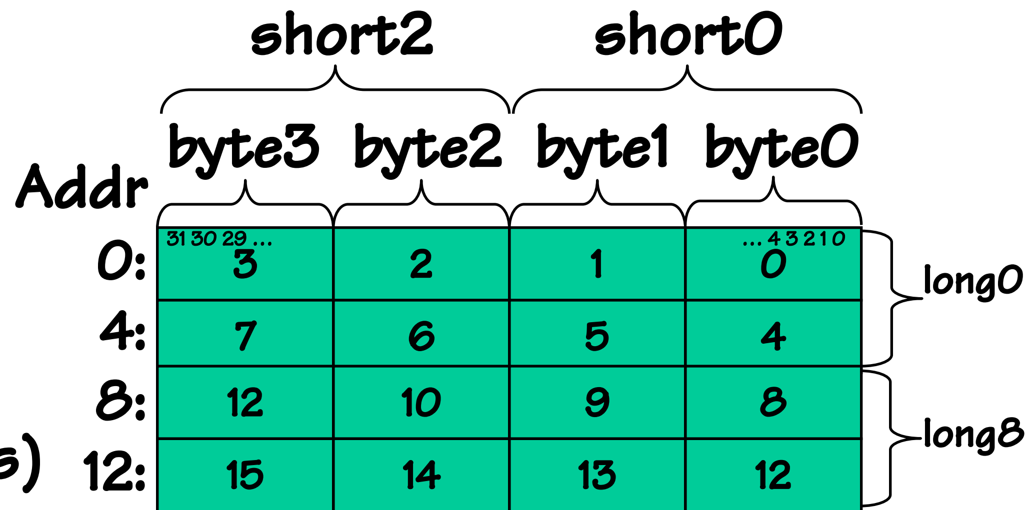
Some MIPs Memory Nits

- Memory locations are 32 bits wide
  - BUT, they are *addressable* in different-sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/double)
- We also frequently need access to individual bits! (Instructions help to do this)
- Every BYTE has a unique address (MIPS is a byte-addressable machine)
- Every instruction is one word



Some MIPs Memory Nits

- Memory locations are 32 bits wide
  - BUT, they are *addressable* in different-sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/double)
- We also frequently need access to individual bits! (Instructions help to do this)
- Every BYTE has a unique address (MIPS is a byte-addressable machine)
- Every instruction is one word



MIPS Register Nits

- There are 32 named registers [$0, $1, …. $31]
- The operands of *all* ALU instructions are registers
  - This means to operate on a variables in memory you must:
    - Load the value/values from memory into a register
    - Perform the instruction
    - Store the result back into memory
  - Going to and from memory can be expensive
    (4x to 20x slower than operating on a register)
  - Net effect: Keep variables in registers as much as possible!
- 2 registers have H/W specific "side-effects"
  (ex: $0 always contains the value '0'… more later)
- 4 registers are dedicated to specific tasks by convention
- 26 are available for general use
- Further conventions delegate tasks to other registers

MIPS Instruction Formats

All MIPs instructions fit in a single 32-bit word. Every instruction includes various "fields" that encode combinations of

- a 6-bit operation or "OPCODE"
  - specifying one of < 64 basic operations
  - escape codes to enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants ("immediate" values) of various sizes, 16-bits, 5-bits, and 26-bits. Sometimes treated as signed values, sometimes not.

There are three basic instruction formats:

MIPS Instruction Formats

All MIPs instructions fit in a single 32-bit word. Every instruction includes various "fields" that encode combinations of

- a 6-bit operation or "OPCODE"
  - specifying one of < 64 basic operations
  - escape codes to enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants ("immediate" values) of various sizes, 16-bits, 5-bits, and 26-bits. Sometimes treated as signed values, sometimes not.

There are three basic instruction formats:

- R-type, 3 register operands (2 sources, destination)

MIPS Instruction Formats

All MIPs instructions fit in a single 32-bit word. Every instruction includes various "fields" that encode combinations of

- a 6-bit operation or "OPCODE"
  - specifying one of < 64 basic operations
  - escape codes to enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants ("immediate" values) of various sizes, 16-bits, 5-bits, and 26-bits. Sometimes treated as signed values, sometimes not.

There are three basic instruction formats:

- R-type, 3 register operands (2 sources, destination)

MIPS Instruction Formats

All MIPs instructions fit in a single 32-bit word. Every instruction includes various "fields" that encode combinations of

- a 6-bit operation or "OPCODE"
  - specifying one of < 64 basic operations
  - escape codes to enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants ("immediate" values) of various sizes, 16-bits, 5-bits, and 26-bits. Sometimes treated as signed values, sometimes not.

There are three basic instruction formats:

- R-type, 3 register operands (2 sources, destination)

MIPS Instruction Formats

All MIPs instructions fit in a single 32-bit word. Every instruction includes various "fields" that encode combinations of

- a 6-bit operation or "OPCODE"
  - specifying one of < 64 basic operations
  - escape codes to enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants ("immediate" values) of various sizes, 16-bits, 5-bits, and 26-bits. Sometimes treated as signed values, sometimes not.

There are three basic instruction formats:

- **R-type**, 3 register operands (2 sources, destination)

- **I-type**, 2 register operands, 16-bit literal constant

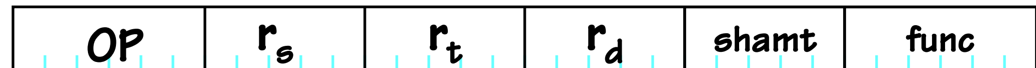
MIPS Instruction Formats

All MIPs instructions fit in a single 32-bit word. Every instruction includes various "fields" that encode combinations of

- a 6-bit operation or "OPCODE"
  - specifying one of < 64 basic operations
  - escape codes to enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants ("immediate" values) of various sizes, 16-bits, 5-bits, and 26-bits. Sometimes treated as signed values, sometimes not.

There are three basic instruction formats:

- **R-type**, 3 register operands (2 sources, destination)



- **I-type**, 2 register operands, 16-bit literal constant

- **J-type**, no register operands, 26-bit literal constant

MIPS ALU Operations

Sample coded operation: ADD instruction

R-type:					
0 0 0 0 0 0	0 1 0 1 1	0 1 0 0 1	0 1 0 1 0	0 0 0 0 0	1 0 0 0 0 0

MIPS ALU Operations

Sample coded operation: ADD instruction

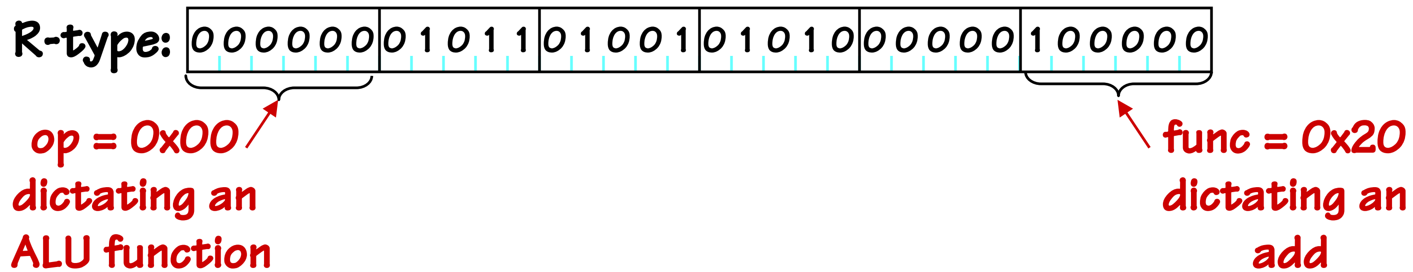
R-type:

000000	01011	01001	01010	00000	100000

op = 0x00
dictating an
ALU function

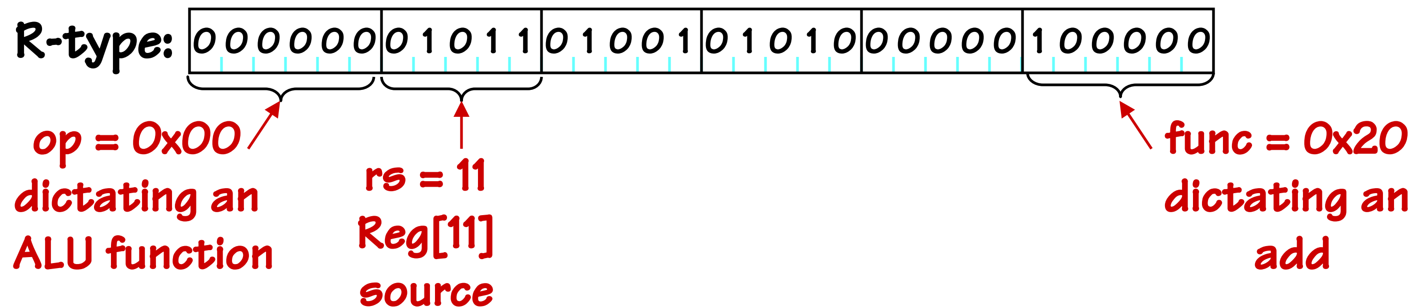
MIPS ALU Operations

Sample coded operation: ADD instruction



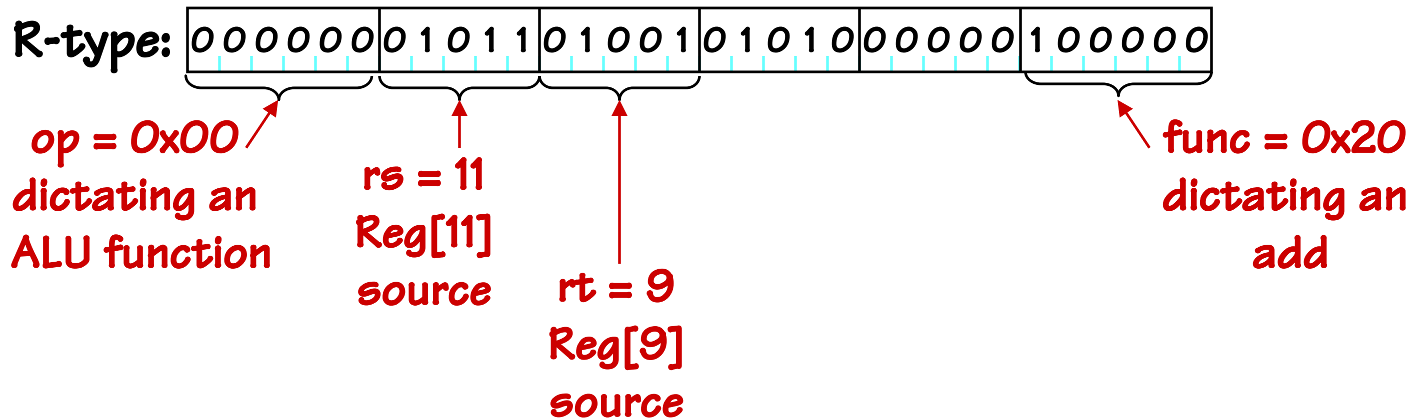
MIPS ALU Operations

Sample coded operation: ADD instruction



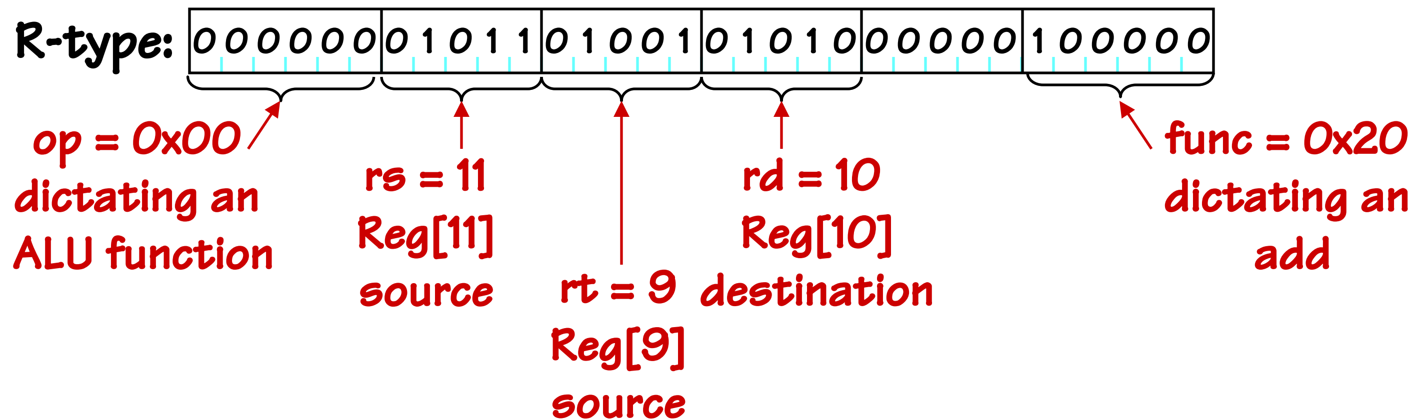
MIPS ALU Operations

Sample coded operation: ADD instruction



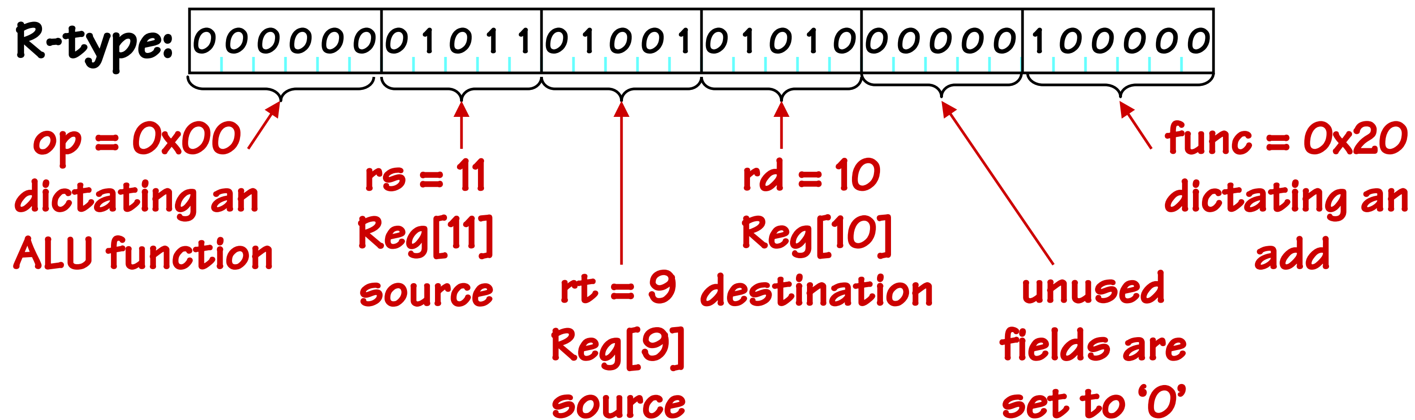
MIPS ALU Operations

Sample coded operation: ADD instruction



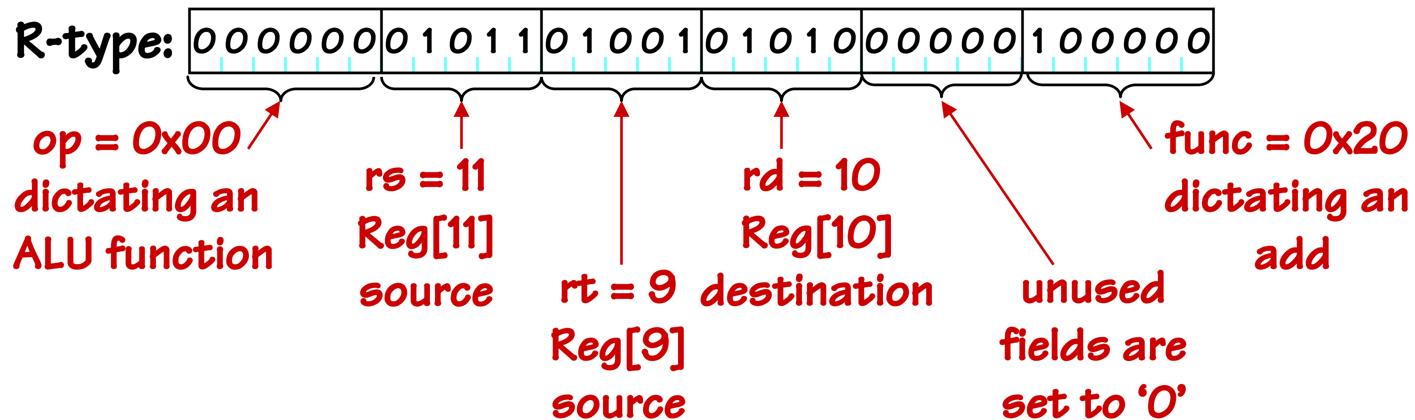
MIPS ALU Operations

Sample coded operation: ADD instruction



MIPS ALU Operations

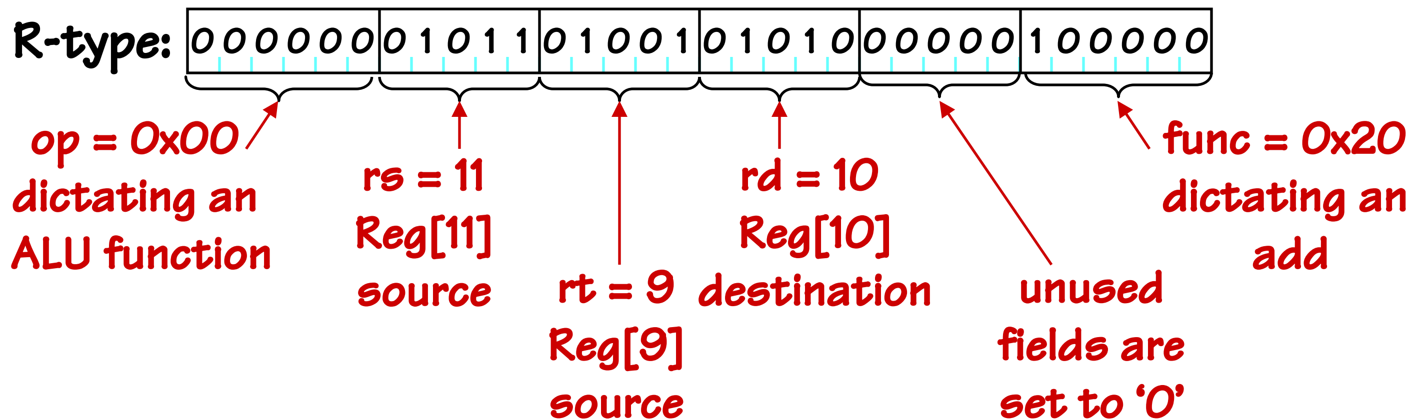
Sample coded operation: ADD instruction



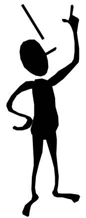
What we prefer to write: `add $10, $11, $9` ("assembly language")

MIPS ALU Operations

Sample coded operation: ADD instruction



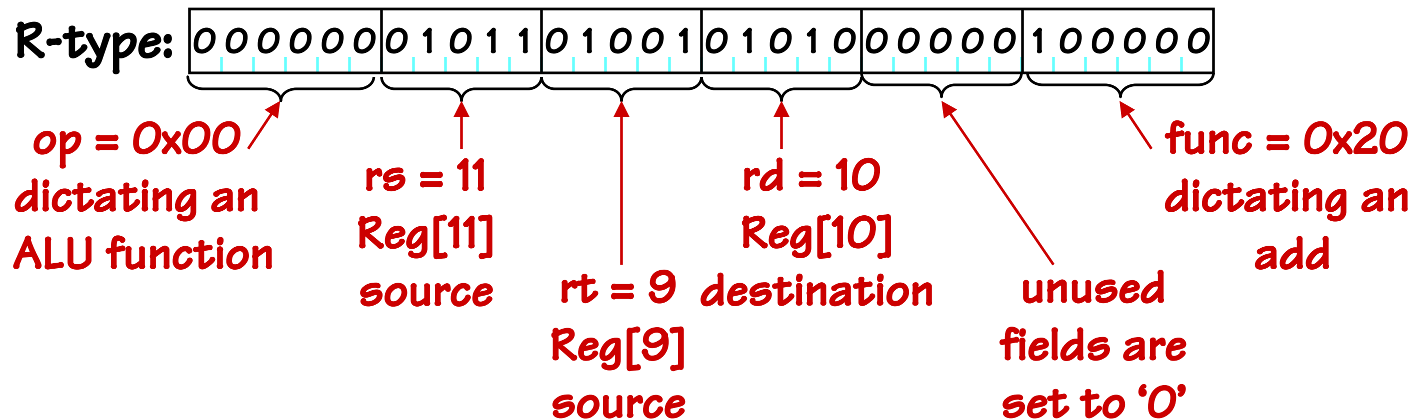
References to register contents are prefixed by a "$" to distinguish them from constants or memory addresses



What we prefer to write: `add $10, $11, $9` ("assembly language")

MIPS ALU Operations

Sample coded operation: ADD instruction



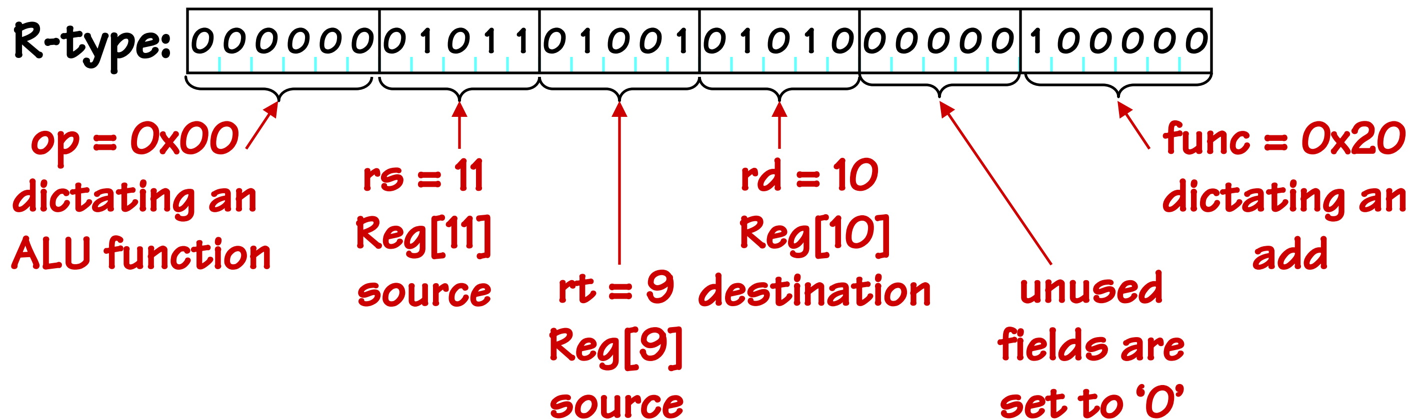
References to register contents are prefixed by a "$" to distinguish them from constants or memory addresses

What we prefer to write: `add $10, $11, $9` ("assembly language")

The convention with MIPS assembly language is to specify the destination operand first, followed by source operands.

MIPS ALU Operations

Sample coded operation: ADD instruction



References to register contents are prefixed by a "$" to distinguish them from constants or memory addresses

What we prefer to write: `add $10, $11, $9` ("assembly language")

The convention with MIPS assembly language is to specify the destination operand first, followed by source operands.

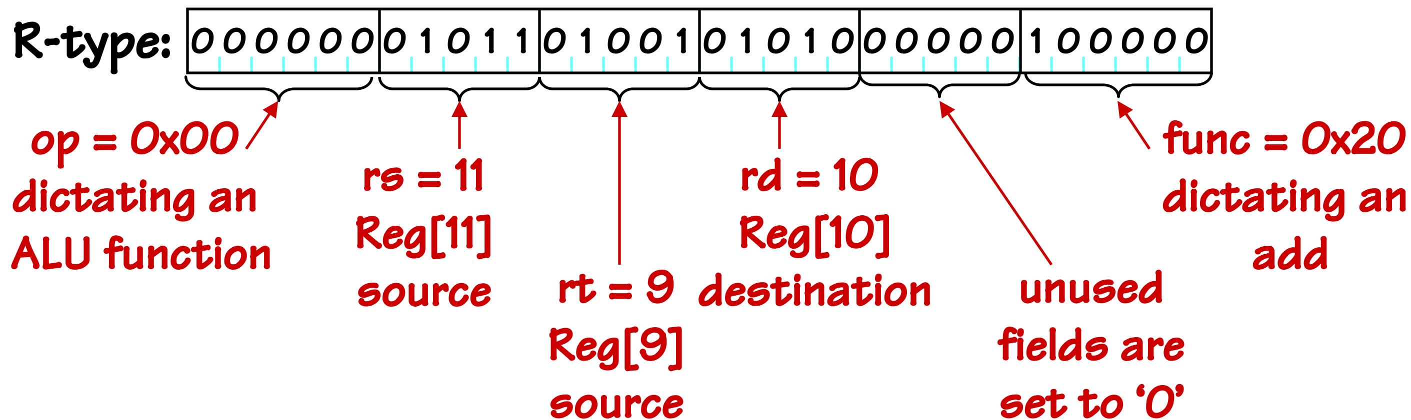
`add rd, rs, rt:`

Reg[rd] = Reg[rs] + Reg[rt]

"Add the contents of rs to the contents of rt; store the result in rd"

MIPS ALU Operations

Sample coded operation: ADD instruction



References to register contents are prefixed by a "$" to distinguish them from constants or memory addresses

What we prefer to write: `add $10, $11, $9` ("assembly language")

The convention with MIPS assembly language is to specify the destination operand first, followed by source operands.

`add rd, rs, rt:`

Reg[rd] = Reg[rs] + Reg[rt]

"Add the contents of rs to the contents of rt; store the result in rd"

Similar instructions for other ALU operations:
- arithmetic: add, sub, addu, subu, mult, multu, div, divu
- compare: slt, sltu
- logical: and, or, xor, nor
- shift: sll, srl, sra, sllv, srav, srlv

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction

R-type:

000000	00000	00010	00010	00100	000000

Assembly: `sll $2, $2, 4`

`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt*
to the left by *shamt*;
store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction

R-type: | 000000 | 00000 | 00010 | 00010 | 00100 | 000000 |

op = 0x00 dictating an ALU function

Assembly: `sll $2, $2, 4`

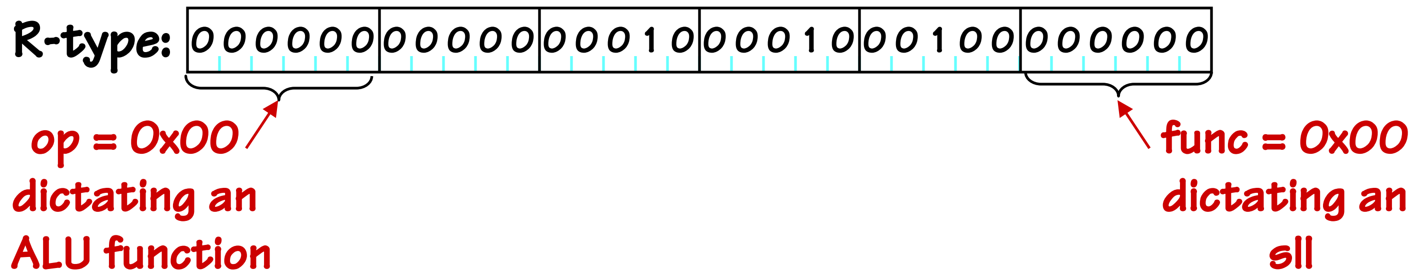
`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

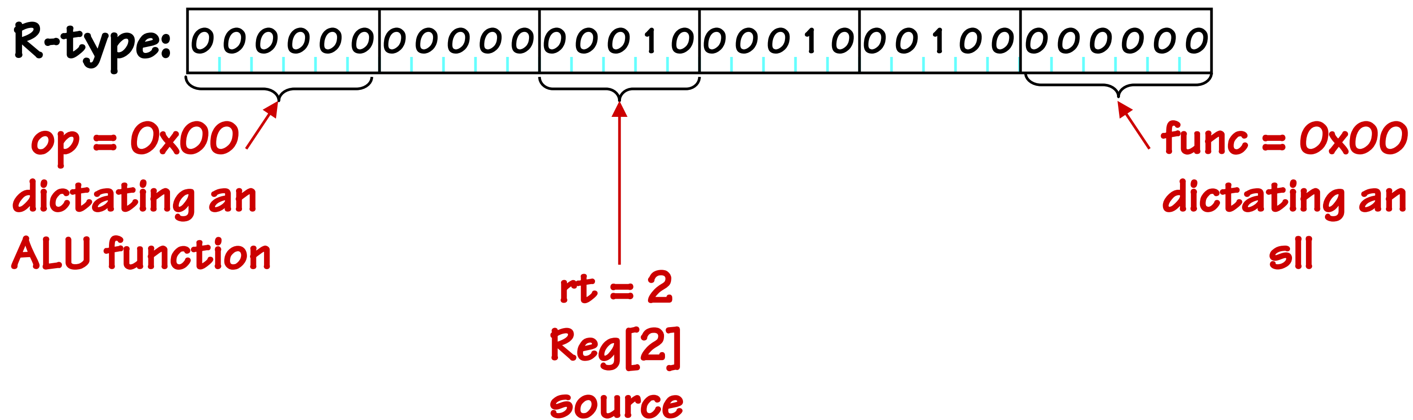
`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

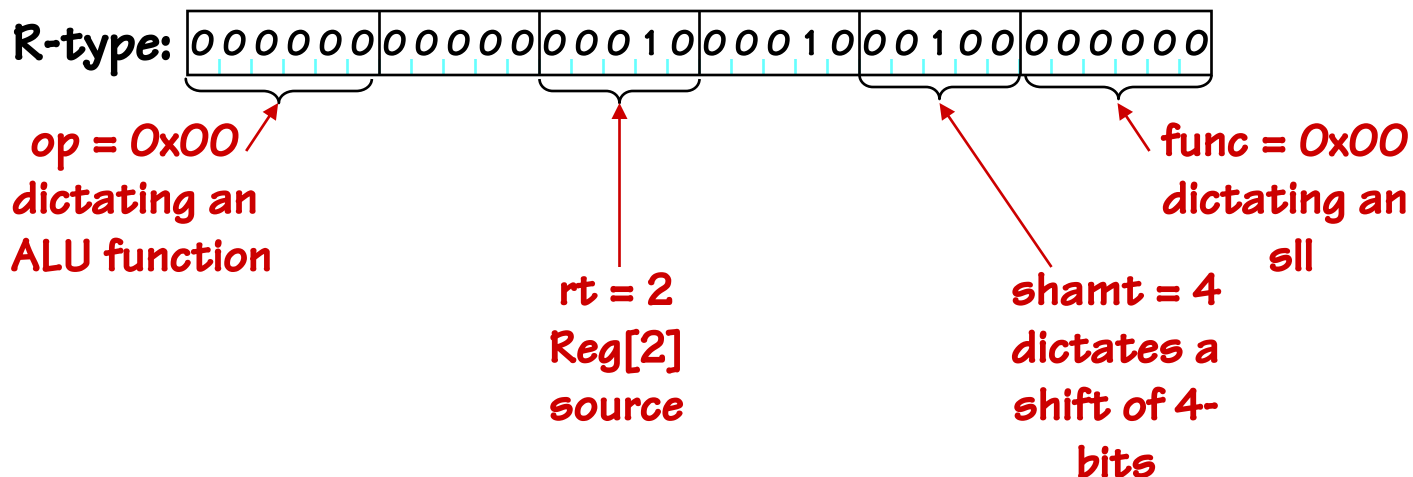
`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt*
to the left by *shamt*;
store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

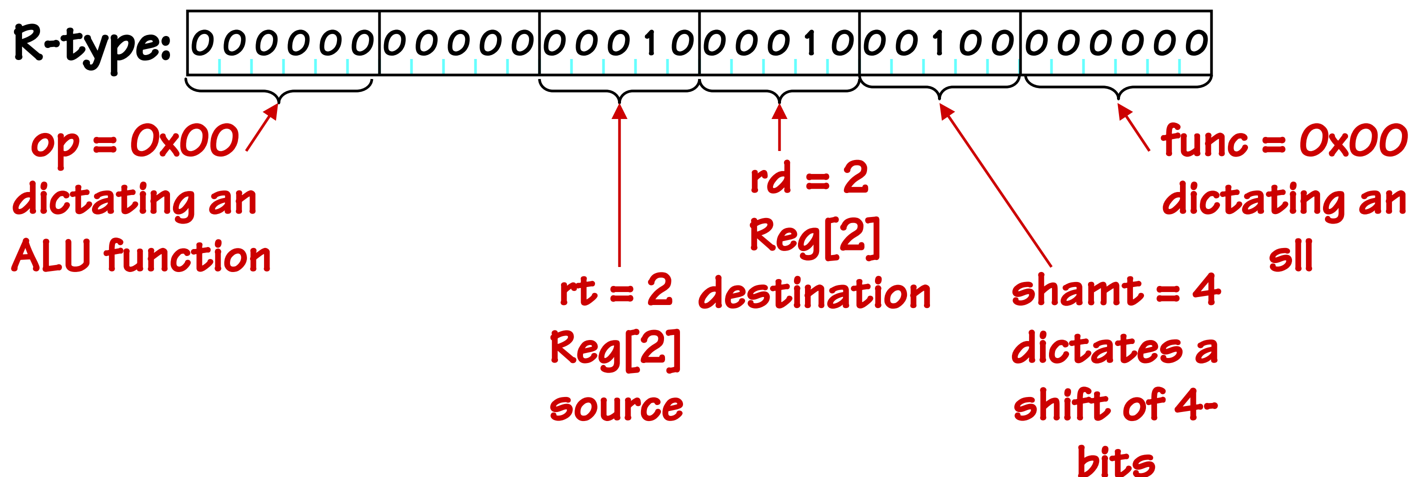
`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt*
to the left by *shamt*;
store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

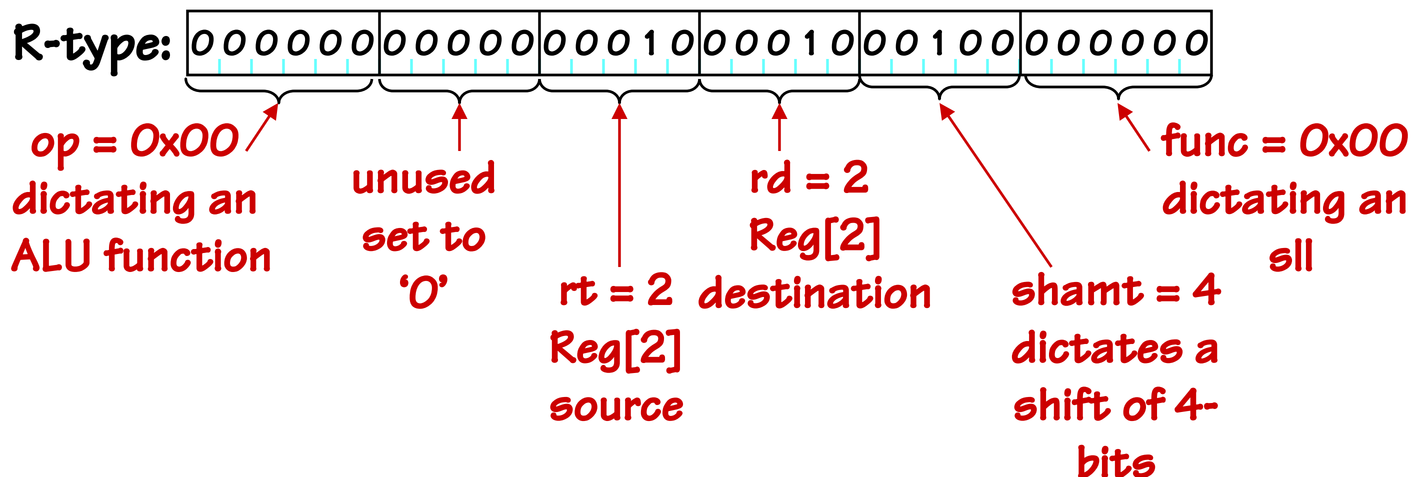
`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

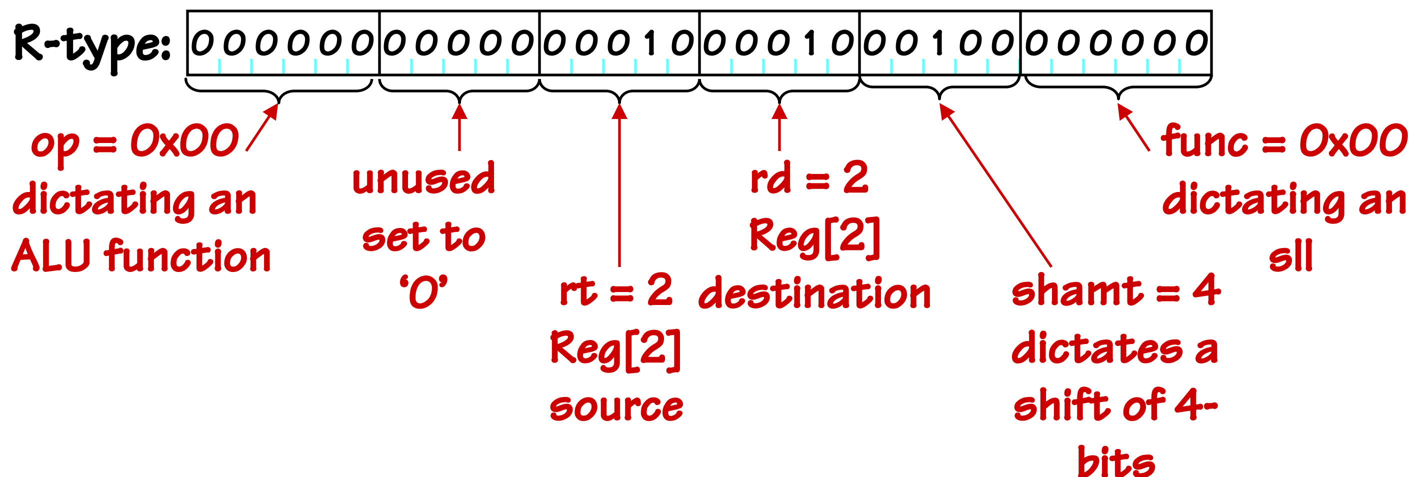
`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

Assembly: `sllv $2, $2, $8`

`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

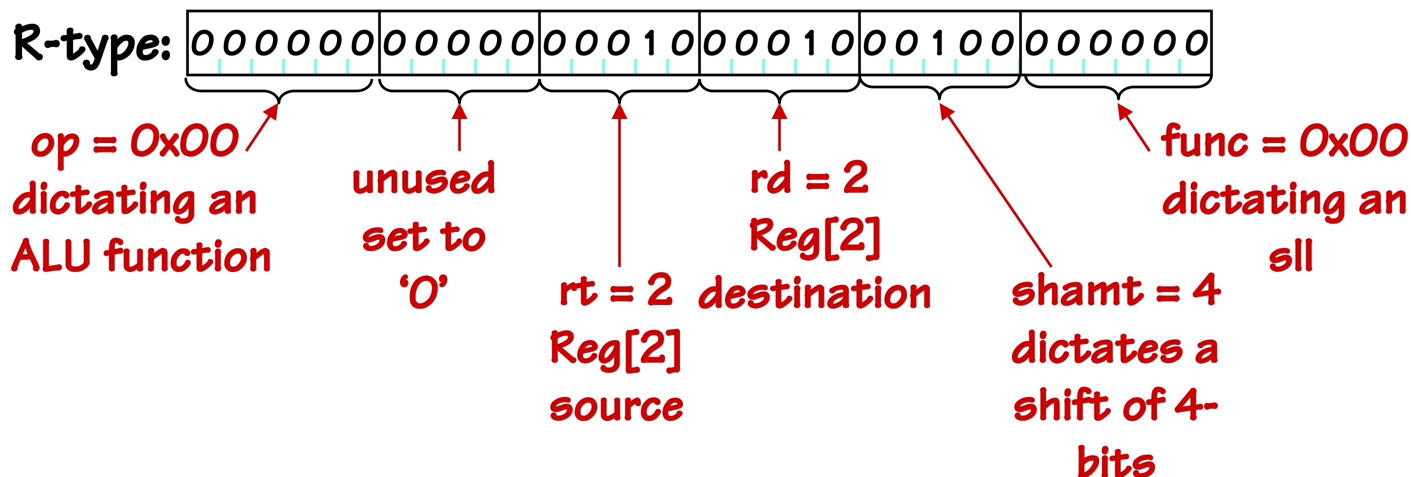
`sllv rd, rt, rs:`

Reg[rd] = Reg[rt] << Reg[rs]

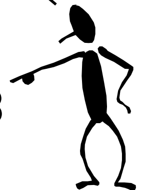
"Shift the contents of *rt* left by the contents of *rs*; store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



This is peculiar syntax for MIPS, in this ALU instruction the rt operand precedes the rs operand. Usually, it's the other way around



Assembly: `sll $2, $2, 4`

`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

Assembly: `sllv $2, $2, $8`

`sllv rd, rt, rs:`

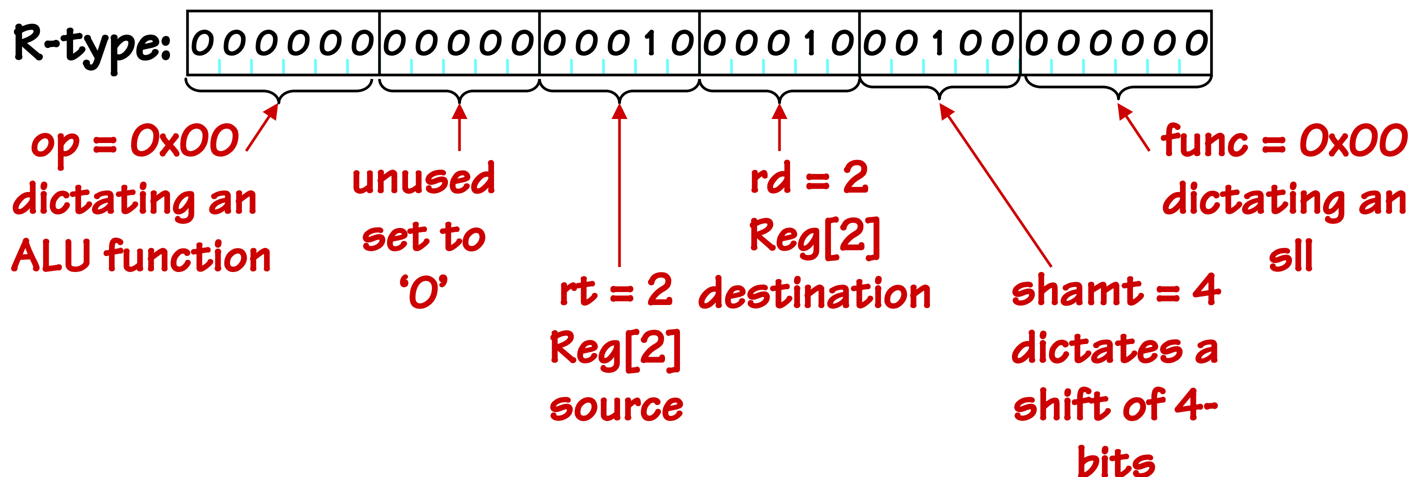
Reg[rd] = Reg[rt] << Reg[rs]

"Shift the contents of *rt* left by the contents of *rs*; store the result in *rd*"

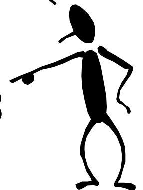
MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction

How are shifts useful?



This is peculiar syntax for MIPS, in this ALU instruction the rt operand precedes the rs operand. Usually, it's the other way around



Assembly: `sll $2, $2, 4`

`sll rd, rt, shamt:`

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

Assembly: `sllv $2, $2, $8`

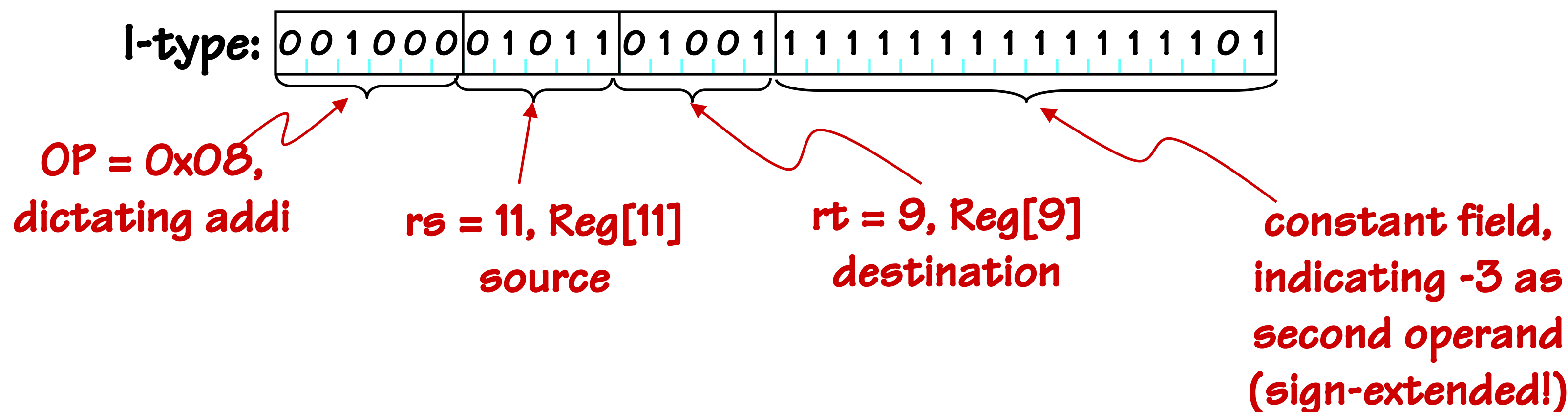
`sllv rd, rt, rs:`

Reg[rd] = Reg[rt] << Reg[rs]

"Shift the contents of *rt* left by the contents of *rs*; store the result in *rd*"

MIPS ALU Operations with Immediate

addi instruction: adds register contents, signed-constant:



Symbolic version: `addi $9, $11, -3`

`addi rt, rs, imm`:

Reg[rt] = Reg[rs] + sxt(imm)

"Add the contents of rs to const; store result in rt"

Similar instructions for other ALU operations:

- arithmetic: addi, addiu
- compare: slti, sltiu
- logical: andi, ori, xori, lui

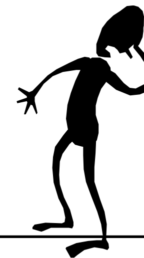
Immediate values are sign-extended for arithmetic and compare operations, but not for logical operations.

Why Built-in Constants? (Immediate)

- Why not…
  - put constants in memory (was common in older instruction sets)?
  - create more hard-wired registers for constants (like $0)?
- SMALL constants are used frequently (50% of operands)
  - In a C compiler (gcc) 52% of ALU operations involve a constant
  - In a circuit simulator (spice) 69% involve constants
  - e.g., B = B + 1; C = W & 0x00ff; A = B + 0;
- ISA Design Principle: *Make the common cases fast*
- MIPS Instructions:

```
addi    $29, $29, 4
slti    $8, $18, 10
andi    $29, $29, 6
ori     $29, $29, 4
```

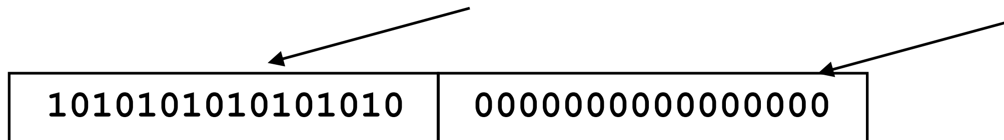
How large of constants should we allow for? If they are too big, we won't have enough bits leftover for the instructions.



Why are there so many different sized constants in the MIPS ISA? Couldn't the shift amount have been encoded using the I-format?

One way to answer architectural questions is to evaluate the consequences of different choices using carefully chosen representative benchmarks (programs and/or code sequences). Make choices that are "best" according to some metric (cost, performance, …).

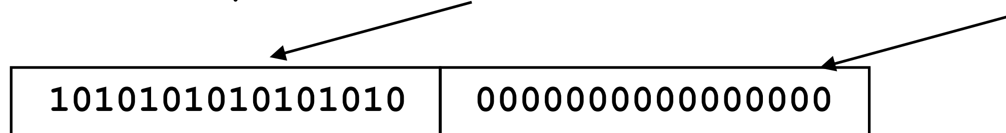
How About Larger Constants?



How About Larger Constants?

- In order to load a 32-bit constant into a register a two instruction sequence is used, "load upper immediate"

```
lui $8, 1010101010101010
```



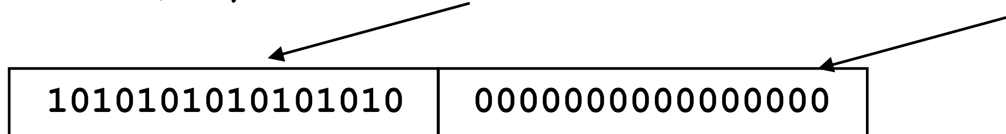
- Then must get the lower order bits right, i.e.,

```
ori $8, $8, 1010101010101010
```

How About Larger Constants?

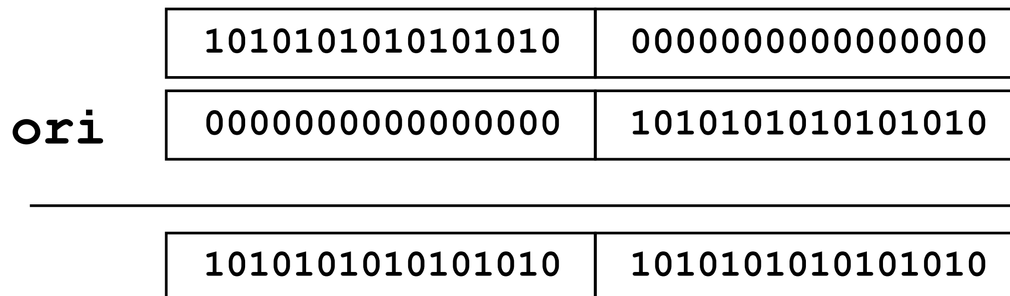
- In order to load a 32-bit constant into a register a two instruction sequence is used, "load upper immediate"

```
lui $8, 1010101010101010
```



- Then must get the lower order bits right, i.e.,

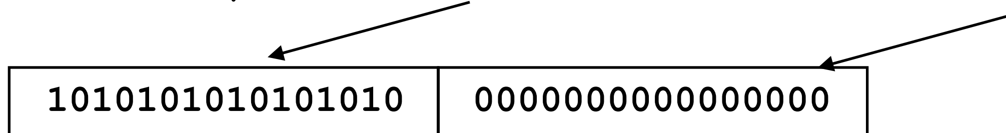
```
ori $8, $8, 1010101010101010
```



How About Larger Constants?

- In order to load a 32-bit constant into a register a two instruction sequence is used, "load upper immediate"

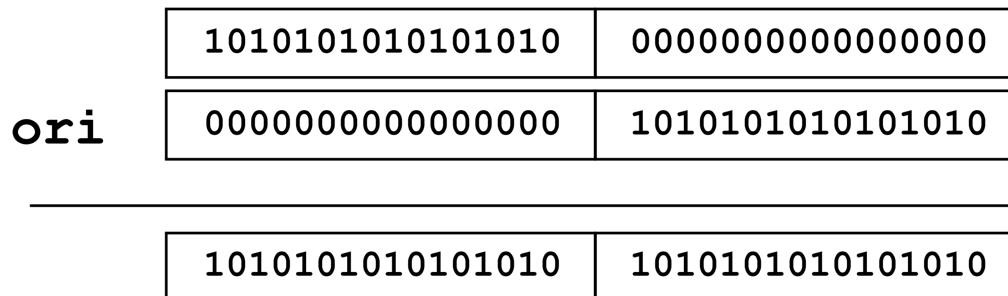
```
lui $8, 1010101010101010
```



- Then must get the lower order bits right, i.e.,

```
ori $8, $8, 1010101010101010
```

Reminder: In MIPS, Logical Immediate instructions (ANDI, ORI, XORI) do not sign-extend their constant operand



First MIPS Program

(fragment)

Suppose you want to compute the following expression:

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

Where the variables f, g, h, i, and j are assigned to registers $16, $17, $18, $19, and $20 respectively. What is the MIPS assembly code?

```
add $8,$17,$18          # (g + h)
add $9,$19,$20          # (i + j)
sub $16,$8,$9           # f = (g + h) - (i + j)
```

These three instructions are like our little ad-hoc machine from the beginning of lecture. Of course, limiting ourselves to registers for storage falls short of our ambitions....

Needed: instruction-set support for reading and writing locations in main memory...

MIPS Load & Store Instructions

MIPS is a LOAD/STORE architecture. This means that *all* data memory accesses are limited to load and store instructions, which transfer register contents to-and-from memory. ALU operations work only on registers.

`lw rt, imm(rs)` Reg[rt] = Mem[Reg[rs] + sxt(const)]

"Fetch into rt the contents of the memory location whose address is *const* plus the contents of rs"

Abbreviation: `lw rt,imm` for `lw rt, imm($0)`

`sw rt, imm(rs)` Mem[Reg[rs] + sxt(const)] = Reg[rt]

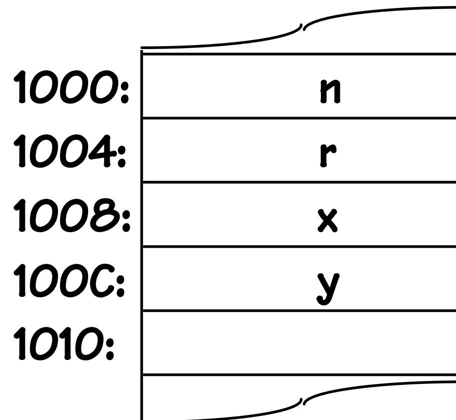
"Store the contents of rt into the memory location whose address is *const* plus the contents of rs"

Abbreviation: `sw rt, imm` for `sw rt, imm($0)`

BYTE ADDRESSES, but lw and sw 32-bit word access word-aligned addresses. The resulting lowest two address bits must be 0!

Storage Conventions

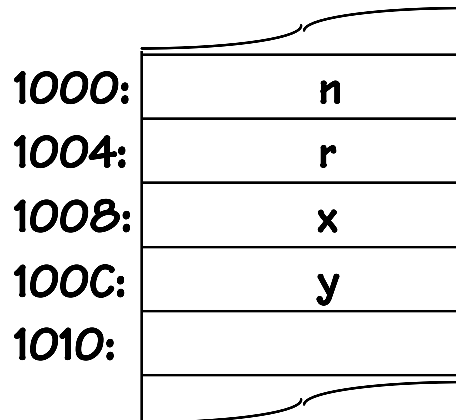
- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results



Storage Conventions

Address assigned at compile time

- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results

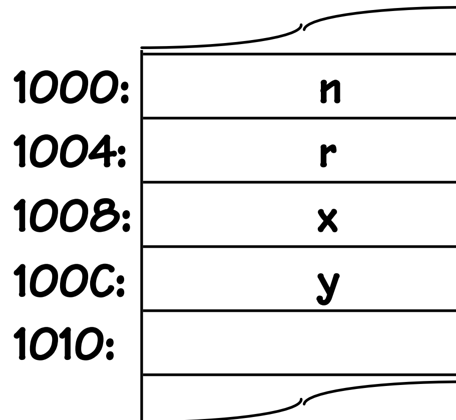


Storage Conventions

Address assigned at compile time

- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results

```
int x, y;
y = x + 37;
```

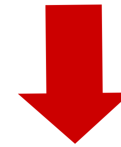


Storage Conventions

Address assigned at compile time

- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results

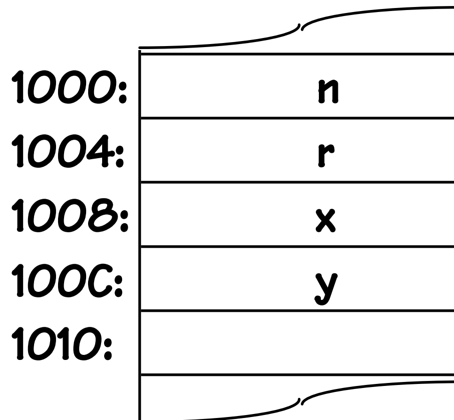
```
int x, y;
y = x + 37;
```



Compilation approach:
LOAD, COMPUTE, STORE

translates to

```
lw      $t0, 0x1008($0)
addi    $t0, $t0, 37
sw      $t0, 0x100C($0)
```

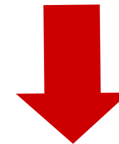


Storage Conventions

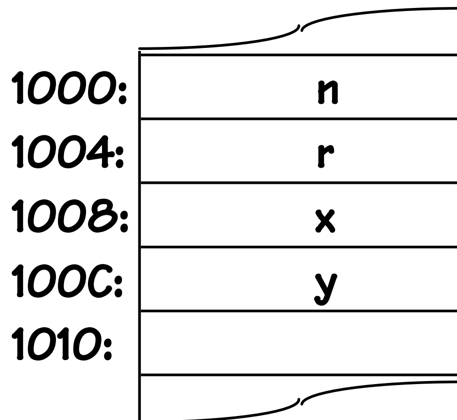
Address assigned at compile time

- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results

```
int x, y;
y = x + 37;
```



Compilation approach:
LOAD, COMPUTE, STORE



translates to

```
lw     $t0, 0x1008($0)
addi   $t0, $t0, 37
sw     $t0, 0x100C($0)
```

or, more humanely, to

```
x=0x1008
y=0x100C
lw     $t0, x
addi   $t0, $t0, 37
sw     $t0, y
```

rs defaults to Reg[0] (0)

MIPS Register Usage Conventions

By convention, the MIPS registers are assigned to specific uses, and names. These are supported by the assembler, and higher-level languages. We'll use these names increasingly.

Name	Register number	Usage
$zero	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Capabilities thus far: Expression Evaluation

Translation of an Expression:

```
int x, y;
y = (x-3)*(y+123456)
```

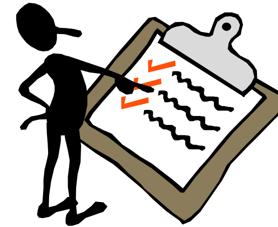
```
x:      .word 0
y:      .word 0
c:      .word 123456
    ...
    lw          $t0, x
    addi        $t0, $t0, -3
    lw          $t1, y
    lw          $t2, c
    add         $t1, $t1, $t2
    mul         $t0, $t0, $t1
    sw          $t0, y
```

- VARIABLES are allocated storage in main memory
- VARIABLE references translate to LD or ST
- OPERATORS translate to ALU instructions
- SMALL CONSTANTS translate to ALU instructions w/ built-in constant
- "LARGE" CONSTANTS translate to initialized variables

NB: Here we assume that variable addresses fit into 16-bit constants!

Can We Run Any Algorithm?

Model <u>thus far</u>:

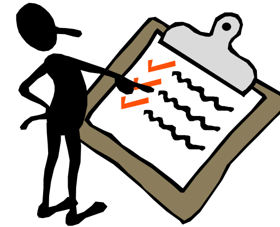


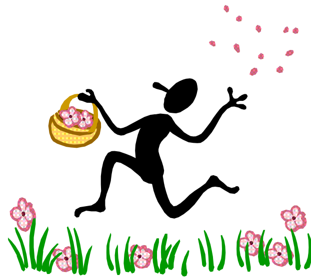
- Executes instructions sequentially –
- Number of operations executed = number of instructions in our program!

Can We Run Any Algorithm?

Model thus far:

- Executes instructions sequentially –
- Number of operations executed = number of instructions in our program!





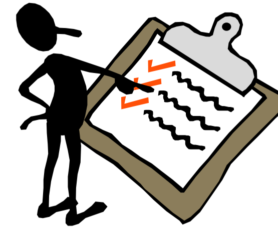
Good news: programs can't "loop forever"!

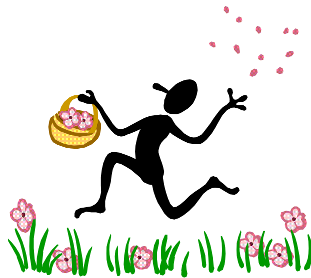
- So far the MIPS subset produces straight-line code only

Can We Run Any Algorithm?

Model <u>thus far</u>:

- Executes instructions sequentially –
- Number of operations executed = number of instructions in our program!





Good news: programs can't "loop forever"!

- So far the MIPS subset produces straight-line code only

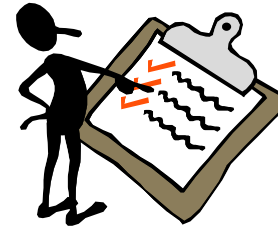
Bad news:

- Straight-line code
- Can't do a loop
- Can't reuse a block of code

Can We Run Any Algorithm?

Model <u>thus far</u>:

- Executes instructions sequentially –
- Number of operations executed = number of instructions in our program!



Good news: programs can't "loop forever"!

- So far the MIPS subset produces straight-line code only

Bad news:

- Straight-line code
- Can't do a loop
- Can't reuse a block of code

Needed: ability to change the PC.

MIPS Branch Instructions

MIPS *branch instructions* provide a way of conditionally changing the PC to some nearby location...

```
beq rs, rt, label  # Branch if equal

    if (REG[RS] == REG[RT])
     {
       PC = PC + 4 + 4*offset;
     }
```

```
bne rs, rt, label  # Branch if not equal

    if (REG[RS] != REG[RT])
     {
       PC = PC + 4 + 4*offset;
     }
```

NB: Branch targets are specified relative to the current instruction (actually relative to the next instruction, which would be fetched by default). The assembler hides the calculation of these offset values from the user, by allowing them to specify a target address (usually a label) and it does the job of computing the offset's value. The size of the constant field (16-bits) limits the range of branches.

MIPS Branch Instructions

MIPS *branch instructions* provide a way of conditionally changing the PC to some nearby location...

```
beq rs, rt, label   # Branch if equal

    if (REG[RS] == REG[RT])
     {
       PC = PC + 4 + 4*offset;
     }
```

```
bne rs, rt, label   # Branch if not equal

    if (REG[RS] != REG[RT])
     {
       PC = PC + 4 + 4*offset;
     }
```

Notice on memory references offsets are multiplied by 4, so that branch targets are restricted to word boundaries.

NB: Branch targets are specified relative to the current instruction (actually relative to the next instruction, which would be fetched by default). The assembler hides the calculation of these offset values from the user, by allowing them to specify a target address (usually a label) and it does the job of computing the offset's value. The size of the constant field (16-bits) limits the range of branches.

MIPS Jumps

- The range of MIPS branch instructions is limited to approximately ± 64K instructions from the branch instruction. In order to branch farther an unconditional jump instruction is used.

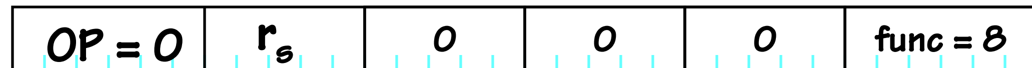
- Instructions:

```
j label           # jump to label (PC = PC[31-28] || CONST[25:0]*4)
jal label         # jump to label and store PC+4 in $31
jr $t0            # jump to address specified by register's contents
jalr $t0, $ra     # jump to address specified by register's contents
```

- Formats:
  - J-type: used for j

  - J-type: used for jal

  - R-type, used for jr



  - R-type, used for jalr

Now we can do a real program: Factorial...

Synopsis (in C):

- Input in n, output in ans
- r1, r2 used for temporaries
- follows algorithm of our earlier data paths.

```
int n, ans;
register int r1, r2;
r1 = 1;
r2 = n;
while (r2 != 0) {
    r1 = r1 * r2;
    r2 = r2 - 1;
}
ans = r1;
```

MIPS code, in assembly language:

```
n:      .word   123
ans:    .word   0
        ...
        addi    $t0, $0, 1          # t0 = 1
        lw      $t1, n              # t1 = n
loop:   beq     $t1, $0, done       # while (t1 != 0)
        mul     $t0, $t0, $t1       # t0 = t0 * t1
        addi    $t1, $t1, -1        # t1 = t1 - 1
        beq     $0, $0, loop        # Always branch
done:   sw      $t0, ans            # ans = r1
```

To summarize:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

MIPS Instruction Decoding Ring

OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mult	multu	div	divu				
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

Summary

- We will use a subset of MIPS instruction set as a prototype
  - Fixed-size 32-bit instructions
  - Mix of three basic instruction formats
    - R-type - Mostly 2 source and 1 destination register
    - I-type - 1-source, a small (16-bit) constant, and a destination register
    - J-type - A large (26-bit) constant used for jumps
  - Load/Store architecture
  - 31 general purpose registers, one hardwired to 0, and, by convention, several are used for specific purposes.
- ISA design requires tradeoffs, usually based on
  - History
  - Art
  - Engineering
  - Benchmark results