2.1 Programmable processor concept

Programmable processor overview

A ***programmable processor*** carries out desired functionality by executing instructions from an instruction memory. Unlike a typical circuit that carries out the same functionality repeatedly, a programmable processor can be configured to carry out nearly any functionality by putting different instructions in the instruction memory. Ex: A single programmable processor can be programmed to carry out the functionality of a calculator, a web browser, and a word processor. Storing instructions into an instruction memory is known as ***programming*** the memory, and those instructions are known as a ***program***.

The processor executes one instruction at a time, proceeding to the next instruction when done.

PARTICIPATION ACTIVITY 2.1.1: Simple programmable processor.

Animation captions:

1. The first instruction loads t0 with the value located at 5001 from the data memory.
2. The second instruction loads t1 with the value located at 5002 from the data memory.
3. The next instruction adds t0 + t1, and puts the sum in t5.
4. The next instruction stores t5's value into data memory location 5005.

A programmable processor includes:

- ***CPU***: A ***central processing unit*** executes instructions by controlling an ALU, register file, and other hardware components.
  - ***ALU***: A component that performs arithmetic and logic operations, like addition or subtraction, on data in the register file. Short for ***arithmetic logic unit***.
  - ***Register file***: A set of registers that holds temporary data accessible by the ALU. A ***register*** is a digital circuit that can store multiple bits, such as 32 bits.
- ***Instruction memory***: A memory that holds instructions.
- ***Data memory***: A memory that holds data used by the instructions.

A ***memory*** is a digital circuit that holds relatively large amounts of data, often organized as bytes with each having a unique ***address***, where each byte can either be read ("load") or written ("store"). Ex: A memory may hold 1024 bytes, with addresses 0, 1, 2, ..., 1023. Common memory sizes range from 1 Kbyte to 4 Gbyte, while typical register files are smaller with perhaps 128 or fewer registers.

PARTICIPATION ACTIVITY 2.1.2: Processor basics.

Refer to the above animation.

1) What is the value in data memory location 5002?

Check Show answer

2) What is the value in t1 after the instruction Load t1 DM[5002] executes?

Check Show answer

3) The first Add instruction adds the values in t0 and t1, and writes the result in what register?

Check Show answer

4) How many executed instructions copied data from the data memory to the register file during the animation?

Check Show answer

5) How many executed instructions used the ALU during the animation?

Check Show answer

6) How many total instructions were executed during the animation?

Check Show answer

7) The CPU contains a register file and what other component?

Check Show answer

8) A typical register file has 1G or more registers. Type true or false.

Check Show answer

Note that when the processor reads data from the data memory or register file, the data is copied, not removed.

Instructions

A processor executes instructions in sequence, one at a time. The instruction order thus matters.

PARTICIPATION ACTIVITY | 2.1.3: Instruction order.

1) Complete the following instruction sequence to add the values in registers t0 and t1 and store the result in DM[5007].

_____

Store t6 to DM[5007]

○ Add t2 = t0 + t1

○ Add t6 = t0 + t1

○ Add t6 = t0 + t2

2) Complete the following instruction sequence to add the values in registers t2 and t6 and store the result in DM[5007].

Add t3 = t2 + t6

_____

○ Store t6 to DM[5007]

○ Store t3 to DM[5000]

○ Store t3 to DM[5007]

3) Select the instruction sequence that calculates the sum of register t1 and DM[5000], and writes the sum in register t5.

○ Load t4 with DM[5000]
Add t5 = t1 + t4

○ Add t5 = t1 + t4
Load t4 with DM[5000]

4) Select the instruction sequence that calculates the sum of t0 and t1, and stores the results to DM[5001].

○ Store t2 to DM[5001]

A processor may support hundreds of possible instruction types. Those instruction types can usually be classified into three categories:

- A ***data transfer instruction*** copies data among the data memory and register file.
- An ***ALU instruction*** operates on data.
- A ***branch instruction*** specifies the location of the next instruction to execute, being different from the next instruction in instruction memory.

PARTICIPATION ACTIVITY 2.1.4: Instruction type categories.

Indicate the category for the instruction.

1) Load t0 with DM[5255]

○ Data transfer

○ ALU

○ Branch

2) Jump to instruction 90

○ Data transfer

○ ALU

○ Branch

3) Subtract t6 = t1 - t4

○ Data transfer

○ ALU

○ Branch

Each instruction typically is encoded into a limited number of bits, such as 32 bits. Using a small limited number of bits per instruction ensures more instructions can fit into the memory, and keeps the processor's circuit simple and fast. Some bits may represent the instruction type (like Load or Add), other bits may indicate the registers involved (like t0 or t1), and others a data memory address (like 5005). As such, the number of instruction types is limited. Ex: If the instruction type is represented in 8 bits, then only 2^8 = 256 instruction types are possible.

Thus, a processor's instruction types are limited and kept basic, like the basic Add, Store, and Load instructions seen above. A programmer must achieve desired functionality using just those relatively-few instruction types.

The set of instruction types supported by a particular processor is called the processor's ***instruction set***. A program written using a processor's instructions is called an ***assembly language program***, in contrast to programs written in higher-level languages like C, C++, Java, or Python.

PARTICIPATION ACTIVITY 2.1.5: Using limited instruction types.

Assume a processor's only instruction type available for adding is:
Add regA = regB + regC
where regA, regB, and regC each is any register.

1) Which computes t5 = t1 + t3?

○ Add t5 = t1 + t3

○ Add t1 = t5 + t3

2) Assume the following initial values: t1 = 7, t2 = 6, t3 = 8. What is in t5 after the following:

Add t0 = t1 + t2
Add t5 = t0 + t3

○ 13

○ 3

○ 21

3) Assume the following initial values: t1 = 7, t2 = 6, t3 = 8. What is in t5 after the following:

Add t3 = t1 + t2
Add t5 = t3 + t3

○ 26

○ 21

4) Which computes t0 = t1 + t2 + t3?

○ Add t3 = t1 + t2
Add t0 = t3 + t3

○ Add t4 = t1 + t2
Add t0 = t4 + t3

5) Which computes t0 = t1 + t2 + t3 + t4?

○ Add t4 = t1 + t2
Add t0 = t4 + t3

○ Add t5 = t1 + t2
Add t6 = t3 + t4
Add t0 = t5 + t6

6) Can t4 = t3 + t2 + t1 + t0 be computed in two instructions?

○ Yes

○ No

Register file

Each register in the register file has a name. The ***zero register*** is a read-only register that always holds the value 0. In the animation above, t0 ... t6 refer to the register file's next seven registers, which can be read and written by instructions.

An ALU instruction may read a register's value and write the operation's result into that very same register.

PARTICIPATION ACTIVITY | 2.1.6: Register file: Reading and writing.

Animation captions:

1. The instruction adds the values located at t2 and t3.
2. The result overwrites the value in t2 with 70.

PARTICIPATION ACTIVITY | 2.1.7: Registers.

1) Assume the following initial values: t0 = 7, t1 = 5, and t2 = 3.
What is in t2 after the following:

Add t2 = t2 + t1

○ 3
○ 8
○ 12

2) Assume the following initial values: t0 = 2, t1 = 5, and t2 = 4. What is in t0 after the following:

Add t0 = t0 + t1
Add t0 = t0 + t2

○ 2
○ 7
○ 11

3) Assume the following initial values: t2 = 7, t3 = 1, and t4 = 9.
What is in t4 after the following:

Add t3 = t3 + t2
Add t4 = t4 + t3

○ 9
○ 10
○ 17

4) Complete the following instruction sequence to add the values in registers t3, t4, and t5, and store the result in DM[1007].

Add t6 = t4 + t5

____

Store t6 to DM[1007]

○ Add t6 = t6 + t3

○ Add t6 = t6 + t4

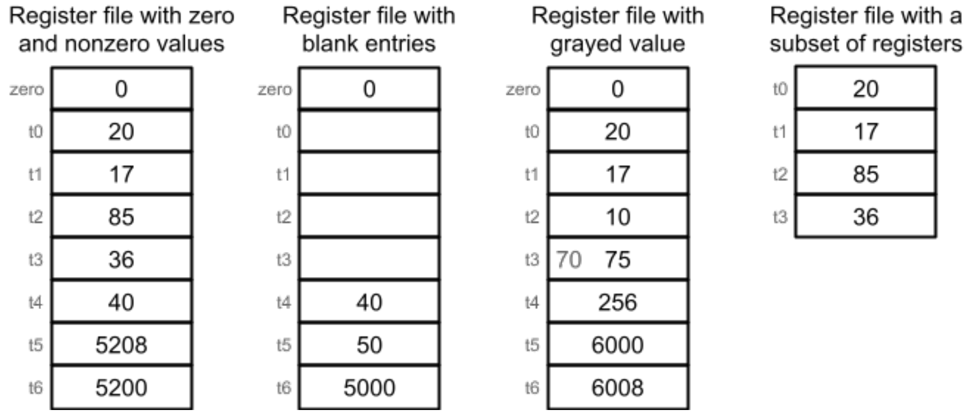
○ Add t6 = t4 + t3

Reset

*A **reset** is an input that when asserted causes a circuit to enter a known state. A processor's reset causes 0's to be written to all registers, including the register file and program counter. So, after the reset, the processor executes the instruction at address 0. A **power-on-reset** circuit resets the processor when power is first applied.*

Register file and data memory

When displaying data values in registers or memory, this material may show: 1) a 0 for register if the register is known to be 0, such as on reset, 2) a blank location if the value is unknown, such as memory location that has not yet been written, or 3) a grayed value representing a previous value for a register or memory location that has been written a new value. Additionally, all registers within the register file may not be shown, instead showing only those registers that are relevant for each example.



Register file with zero and nonzero values

zero	0
t0	20
t1	17
t2	85
t3	36
t4	40
t5	5208
t6	5200

Register file with blank entries

zero	0
t0	
t1	
t2	
t3	
t4	40
t5	50
t6	5000

Register file with grayed value

zero	0
t0	20
t1	17
t2	10
t3	70 75
t4	256
t5	6000
t6	6008

Register file with a subset of registers

t0	20
t1	17
t2	85
t3	36

MIPS

MIPS is a processor that was popular in various computers in the 1990's, and is found in some embedded computing devices today. MIPS is presently one of the most popular processors for learning assembly language programming, and also for learning processor design. MIPS is known for having a simple and elegant instruction set, which in turn enables simple and fast processor designs.

MIPS' instruction set has just over 100 instructions, and each instruction is 32 bits. The MIPS register file has 32 registers, each being 32 bits. Memory addresses are 32 bits. Memory can be accessed by words (4 bytes), half words (2 bytes), or bytes.

For educational purposes, this material teaches a greatly-simplified version of MIPS, known as ***MIPSzy***, using a small subset of the MIPS instruction set, and using a register file with only 8 primary registers. MIPSzy only allows memory to be accessed by words (4 bytes).

PARTICIPATION ACTIVITY 2.1.8: MIPS and MIPSzy.

1) MIPS is the most popular processor in commercial products today.

○ True

○ False

2) The register file in MIPS has 32 registers.

○ True

○ False

3) The register file in MIPSzy has 32 registers.

○ True

○ False

Exploring further:

- MIPS Processor
- Computer Organization and Design (6e) - Interactive Version (MIPS)

2.2 lw, sw: Load and store instructions

Load instruction: lw

A ***load instruction*** copies data from memory into a register. A MIPS load instruction format is shown below. Another section discusses the reason for the 0( ) around the memory-address.

```
lw register 0(memory-address)
```

MIPS register names start with a $. MIPSzy supports 8 registers. Writeable registers are $t0, $t1, ..., $t6. A special $zero register always has the value 0 and can only be read, not written.

The load instruction's memory-address is a register whose value is the memory address from which data is copied.

Load word

lw is short for "load word", in contrast to just loading a byte (a word is four bytes).

PARTICIPATION ACTIVITY 2.2.1: Load instruction: lw.

1) If $t6's value is 2020, what is the memory address being accessed by the following instruction?

```
lw $t0, 0($t6)
```

Check **Show answer**

2) Given the following register file and memory contents, what value is loaded into register $t3 by the following instruction?

```
lw $t3, 0($t6)
```

Register file

$zero	0
$t0	
$t1	
$t2	
$t3	
$t4	40
$t5	5208
$t6	5200

Data memory (DM)

5200	24
5204	400
5208	30
5212	80
5216	-20
5220	17

Check **Show answer**

3) Given the following register file, complete the load instruction to load register $t2 with data at memory address 5012.

Register file

$zero	
$t0	300

$t1	
$t2	
$t3	
$t4	5000
$t5	5008
$t6	5012

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

Check Show answer

4) Assuming $t5 holds 6000, write a load instruction that loads register $t4 with data at memory address 6000.

Check Show answer

Store instruction: sw

A ***store instruction*** copies data from a register to memory. A MIPS store instruction format is shown below. Another section discusses the reason for the 0( ) around the memory-address.

```
sw register 0(memory-address)
```

PARTICIPATION ACTIVITY | 2.2.2: Store instruction: sw.

1) Assuming $t6 holds 600 and $t0 holds 5008, what is the memory address for the following instruction?

```
sw $t6, 0($t0)
```

Check Show answer

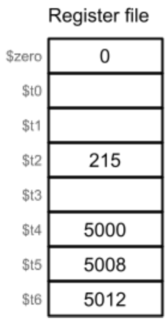
2) Given $t2 holds 6200, $t3 holds 536, and $t4 holds 616, what value is stored into memory?

```
sw $t3, 0($t2)
```

Check Show answer

3) Given the following register file,

complete the store instruction to store register $t2's value into memory at address 5000.


Register file

$zero	0
$t0	
$t1	
$t2	215
$t3	
$t4	5000
$t5	5008
$t6	5012

```
sw [        ] , 0($t4)
```

Check Show answer

4) Assuming $t0 holds 5400 and $t1 holds 280, write a store instruction that stores register $t1's value into memory at address 5400.

□

Check Show answer

Instruction format summary: lw, sw

The condensed instruction format below specifies all registers using $ followed by a single character. Ex: $a.

Table 2.2.1: Instruction summary: lw, sw.

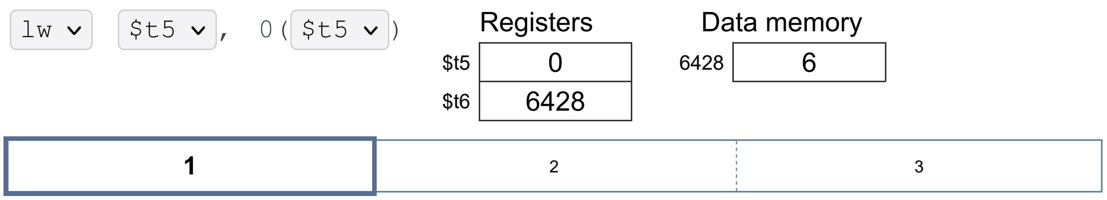
Instruction	Format	Description	Example
lw	`lw $a, 0($b)`	Load word: Copies data from memory at address $b to register $a.	`lw $t3, 0($t6)`
sw	`sw $a, 0($b)`	Store word: Copies data from register $a to memory at address $b.	`sw $t1, 0($t3)`

CHALLENGE ACTIVITY 2.2.1: Load and store instructions.

459784.3174716.qx3zqy7

Start

Compute: $t5 = DM[6428]



```
lw $t5, 0($t5)
```

Check Next

2.3 Memory alignment and endianness

Memory alignment

A particular memory may store a sequence of 32-bit wide words (instructions or data). One might assume addresses for each word would increment by 1, as in: 0, 1, 2, 3, 4, 5, etc. However, each byte in a word can be addressed individually. Thus, addresses of each word increment by 4: 0, 4, 8, 12, 16, etc. ***Memory alignment*** is the restriction of word addresses to multiples of 4 (or other multiples for different processors).

Instructions that load or store words must use addresses that are multiples of four. Instructions that load or store bytes may use any address.

PARTICIPATION ACTIVITY | 2.3.1: Memory alignment: Because each byte is addressable, word addresses are multiples of 4. □

Animation captions:

1. One might assume that word addresses increment by 1.
2. However, each byte is addressable.
3. Thus, word addresses increment by 4.
4. Byte instructions refer only to the indicated byte. Word instructions refer to the entire word.
5. Word instructions (load or store) must be aligned: Multiples of 4 only.

PARTICIPATION ACTIVITY | 2.3.2: Memory alignment. □

Consider the above animation on memory alignment.

1) How many bytes exist per word? □

Check **Show answer**

2) How many byte addresses exist for one word? □

Check Show answer

3) What are the byte addresses for the bytes in the word starting at address 12? Type as: 0, 1, 2, 3 □

Check Show answer

4) Is storing a byte into address 15 allowed? Type yes or no. □

Check Show answer

5) Is storing a word into address 15 allowed? Type yes or no. □

Check Show answer

6) A programmer wishes to load the last word of the memory shown in the animation. What address should be used in the load word instruction? □

Check Show answer

Endianness

Endianness refers to whether bytes in a word are ordered starting with the most-significant byte first (***big-endian***) or the least-significant byte first (***little-endian***). Some processors use big-endian format, others use little-endian.

PARTICIPATION ACTIVITY 2.3.3: Big-endian vs. little-endian. □

Animation captions:

1. An instruction may require one byte for op, one for regA, one for regB, and one for regC (not real; for example purposes only).
2. Big endian stores the "most-significant" byte (op) first.
3. In contrast, little-endian stores the least-significant byte first.
4. Similarly for binary numbers.

Endianness only impacts the ordering of bytes; the bits within the byte remain in the same order. Ex: 00001111 remains 00001111 for either big or little endian formats, and does not become 11110000.

Programmers usually need not be concerned with endianness, unless doing byte-level operations within a word (which is rare).

PARTICIPATION ACTIVITY 2.3.4: Big-endian.

The binary number 00000000 00001111 11111111 11000000 (1048512 in decimal) is to be stored in word 20 in big-endian format. Indicate the byte address of each byte.

If unable to drag and drop, refresh the page.

00001111 **11111111** **00000000** **11000000**

20

21

22

23

Reset

PARTICIPATION ACTIVITY 2.3.5: Little-endian.

The binary number 00000000 00001111 11111111 11000000 (1048512 in decimal) is to be stored in word 20 in little-endian format. Indicate the byte address of each byte.

If unable to drag and drop, refresh the page.

11000000 **11111111** **00000000** **00001111**

20

21

22

23

Reset

PARTICIPATION ACTIVITY | 2.3.6: Endianness.

1) Little-endian processors are faster than big-endian.

○ True

○ False

2) Programmers spend much time and effort focusing on endianness.

○ True

○ False

3) In little-endian format, 10000000 would become 00000001.

○ True

○ False

2.4 addi, add: Add instructions

Add with immediate instruction: addi

A program often needs to add a specific value to a register, such as adding register $t3 and 4. An ***add immediate*** (***addi***) instruction adds a register's value and an immediate value. An ***immediate*** is a value specified within an instruction. In MIPS, the immediate is a 16-bit number that can range from -32,768 to 32,767. A MIPS addi instruction format is shown below, which computes regA = regB + immediate.

```
addi regA, regB, immediate
```

PARTICIPATION ACTIVITY | 2.4.1: Add immediate (addi) instruction.

Animation captions:

1. The add immediate instruction adds the immediate value 4 to the value held in $t0. The sum is written to register $t1.
2. An immediate value can be negative. -10 is added to the value held in $t2, and the result 40 + -10 or 30 is written to $t3.

PARTICIPATION ACTIVITY | 2.4.2: addi instruction.

For each question, assume initial register values of:

- $t0: 20
- $t1: 50

- $t2: 60

1) After the following, what is $t4?

```
addi $t4, $t2, 1
```

Check Show answer

2) After the following, what is $t3?

```
addi $t3, $t1, -5
```

Check Show answer

3) After the following, what is $t2?

```
addi $t2, $t2, 6
```

Check Show answer

4) Type an addi instruction that writes $t5 with the sum of $t4 and 17.

Check Show answer

5) Type an instruction that adds 3 to $t4, writing the sum to $t4.

Check Show answer

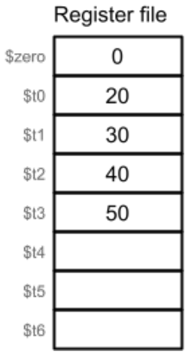
Commonly, a specific value needs to be written to a register. The addi instruction format below computes regA = immediate:

```
addi regA, $zero, immediate
```

Since $zero always holds the value 0, the sum is equal to the immediate value, and the immediate value is written to the register.

PARTICIPATION ACTIVITY 2.4.3: Initializing registers with addi.

Given the following register file contents, match the register to the value held in the register as the provided instructions.



Register file

$zero	0
$t0	20
$t1	30
$t2	40
$t3	50
$t4	
$t5	
$t6	

```
addi $t4, $zero, 40
addi $t3, $t0, 0
addi $t2, $zero, 50
```

If unable to drag and drop, refresh the page.

40 **20** **50**

$t2

$t3

$t4

Reset

Add instruction: add

An ***add instruction*** computes the sum of two register values, and writes the sum into a register. A MIPS add instruction format is shown below, which computes regA = regB + regC.

```
add regA, regB, regC
```

The register written by an instruction is called the ***destination register***. A register read by an instruction is called a ***source register***. For the add instruction, regA is the destination register, and regB and regC are source registers.

PARTICIPATION ACTIVITY | 2.4.4: Add instruction.

Assume initial register values of

- $t0: 20
- $t1: 30
- $t2: 40

1) After the following, what is $t0?

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

□

Check **Show answer**

2) After the following, what is $t2?

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

Check Show answer

3) After the following, what is $t2?

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

Check Show answer

4) After the following, what is $t2?

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

Check Show answer

5) Type an instruction that writes $t3 with the sum of $t5 and $t6.

Check Show answer

Table 2.4.1: Instruction summary: addi, add.

Instruction	Format	Description	Example
addi	`addi $a, $b, C`	Add immediate: Adds register $b and the immediate value C, and writes the sum into register $a.	`addi $t3, $t2, 7`
add	`add $a, $b, $c`	Add: Computes the sum of registers $b and $c, and writes the sum into register $a.	`add $t4, $t1, $t2`

CHALLENGE ACTIVITY 2.4.1: Add immediate and add instructions.

459784.3174716.qx3zqy7

Start

Compute: $t2 = $t1 + 9

addi ⌄ $t1 ⌄, $t1 ⌄, 0

Registers

$t1	2
$t2	0

1 2 3 4

Check Next

2.5 Comments

A ***comment*** is text in a program intended just for humans reading the program, rather than for the processor executing the program. In MIPS, a comment is any text on a line following the # symbol.

```
lw $t1, 0($t6)     # This is a comment
add $t1, $t1, $t1  # Another comment
# And yet another comment
```

PARTICIPATION ACTIVITY 2.5.1: Comments.

Given the following code, indicate which is a comment.

```
lw $t2, 0($t6)  # Load DM[2000]
# add $t2, $t3, $t4
# FIXME: Finish the
program soon
#####
# 6/3/16
```

1) lw

○ Comment

○ Not a comment

2) Load DM[2000]

○ Comment

○ Not a comment

3) add $t2, $t3, $t4

○ Comment

○ Not a comment

4) FIXME: Finish the

○ Comment

○ Not a comment

5) program soon

○ Comment

6) ####

○ Not a comment

○ Comment

○ Not a comment

7) 6/3/16

○ Comment

○ Not a comment

2.6 A small assembly program

Creating and executing a small program

Given desired behavior, a programmer must create an instruction sequence implementing such behavior, using only the processor's available instructions.

PARTICIPATION ACTIVITY 2.6.1: Creating and executing a small program.

Animation captions:

1. A sequence of instructions are specified to implement the desired goal.
2. The addi instruction is used to initialize $t4, $t5, and $t6 with the memory addresses 5020, 5024, and 5032, respectively.
3. $t4 and $t5 hold the memory addresses 5020 and 5024. The data in memory address 5020 and 5024 is copied into registers $t0 and $t1, respectively.
4. The sum of registers $t0 and $t1 is computed, and then written into register $t2.
5. The data in register $t2 is copied to memory at address 5032.
6. Executing the program illustrates that the desired goal is met: DM[5032] = DM[5020] + DM[5024].

PARTICIPATION ACTIVITY 2.6.2: Creating small programs.

Indicate the WRONG item. DM refers to data memory. Assume initial values:
$t0: 5000
$t1: 5004
$t2: 5008.

1) Desired behavior: DM[5000] = DM[5004] + DM[5008]

```
lw $t3, 0($t1)     # Load DM[5004]
lw  $t4
, 0($t2)   # Load DM[5008]
add $t5,  $t3
, $t4  # Add DM[5004] +  DM[5008]
```

sw $t5, 0($t1)   # Store result into DM[5000]

2) Desired behavior: DM[5000] = DM[5000] + DM[5004] □

lw $t3, 0($t0)
lw $t4, 0($t2)
add $t3 , $t3, $t4
sw $t3, 0($t0)

3) Desired behavior: DM[5008] = DM[5004] + DM[5004] + DM[5004] □

lw $t3, 0($t1)
add $t4, $t3, $t3
add $t4, $t4, $t4
sw $t4, 0($t2)

PARTICIPATION ACTIVITY 2.6.3: Load, store, and memory.

1. Run the simulation step-by-step, observing memory values.
2. Change DM[5000]'s value to 45 by clicking the memory value on the right, then run again.
3. Store the addition result in DM[5004] by appending `sw $t2, 0($t4)`

Conserving registers in assembly programs

Registers are limited, so programmers should conserve registers. If a value in a register is not read later, the register can be reused by writing another value. Ex: Assume $t4 holds a memory address used in a lw instruction. If that memory address is not used by another instruction, $t4 can be reused to hold a different memory address or used to hold the result of a computation.

PARTICIPATION ACTIVITY 2.6.4: Conserving registers.

Animation captions:

1. $t4, $t5, and $t6 are initialized with the memory addresses 5020, 5024, and 5032. Each of the addresses is only used once, and a single register could be used.
2. For each lw and sw instruction, an addi instruction initializes $t4 with the memory address. The following lw or sw instruction uses $t4 to access memory.

PARTICIPATION ACTIVITY 2.6.5: Conserving register in assembly programs.

1) Which register can be reused in the addi □

and sw instructions to store $t6 to memory address 5048?

```
addi $t2, $zero, 5048
lw $t4, 0($t2)
add $t6, $t6, $t4
sw $t6, 0(___)
```

○ $t2

○ $t6

2) Which register can be reused in the addi and lw instructions to load $t5 with data at memory address 5012?

```
addi $t1, $zero, 5000
lw $t4, 0($t1)
addi $t2, $zero, 5008
lw $t5, 0($t2)
addi ___, $zero, 5012
lw $t6, 0(___)
add $t4, $t4, $t5
sw $t4, 0($t2)
```

○ $t1

○ $t2

3) If the add instruction's destination register is changed from $t3 to $t2, what other instruction must be updated?

```
addi $t4, $zero, 5020
lw $t1, 0($t4)
addi $t4, $zero, 5024
lw $t2, 0($t4)
add $t3, $t2, $t1
addi $t4, $zero, 5032
sw $t3, 0($t4)
```

○ lw $t2, 0($t4)

○ sw $t3, 0($t4)

PARTICIPATION ACTIVITY 2.6.6: Conserving registers used for memory addresses.

The program below computes DM[5000] = DM[5000] + DM[5004] + DM[5008]

1. Run the simulation step-by-step, observing that program uses $t4, $t5, and $t6 for the three memory addresses 5000, 5004, and 5008.
2. Revise the program to only use $t4 for the memory addresses. Before each lw or sw instruction, add an addi instruction to initialize $t4 with the memory address. Then, use

$t4 in the lw or sw instructions.

CHALLENGE ACTIVITY | 2.6.1: Load, store, add, and addi instructions.

459784.3174716.qx3zqy7

2.7 sub, mul: Subtraction and multiplication instructions

Subtract instruction: sub

A ***subtract instruction*** (***sub***) computes the difference of two register values, and writes the difference into a register. A MIPS subtract instruction format is shown below, which computes regA = regB - regC.

```
sub regA, regB, regC
```

PARTICIPATION ACTIVITY | 2.7.1: sub instruction.

Assume initial register values of:

- $t0: 30
- $t1: 10
- $t2: -5
- $t3: 5

1) After the following, what is $t4?

```
sub $t4, $t1, $t3
```

Check **Show answer**

2) After the following, what is $t0?

```
sub $t0, $t0, $t3
```

Check **Show answer**

3) After the following, what is $t5?

```
sub $t5, $t1, $t2
```

Check Show answer

4) Type an instruction to subtract $t3 from $t4, writing the difference to $t5.

Check Show answer

Multiply instruction: mul

A ***multiply instruction*** (***mul***) computes the product of two register values, and writes the product into a register. A MIPS multiply instruction format is shown below, which computes regA = regB * regC. The multiply instruction computes a 32-bit product, and ignores any overflow that may result from multiplying two 32-bit values.

```
mul regA, regB, regC
```

PARTICIPATION ACTIVITY 2.7.2: mul instruction.

Assume initial register values of:

- $t0: 40
- $t1: 20
- $t2: 70000
- $t3: 30

1) After the instruction `mul $t3, $t1, $t0` what is $t3?

○ 600

○ 800

2) Which instruction multiplies $t4 with $t5, writing the product to $t5.

○ mul $t5, $t5, $t4

○ mul $t4, $t5, $t5

3) Does the following instruction result in an overflow?

mul $t5, $t1, $t2

○ Yes

○ No

4) Does the following instruction result in an overflow?

mul $t5, $t2, $t2

○ Yes

Pseudoinstructions

*A **native instruction** is an assembly instruction directly supported by a processor's hardware. A **pseudoinstruction** is an assembly instruction that must be replaced by one or more native instructions before being executed. Pseudoinstructions are used to keep the number of native instructions small, which leads to more efficient processor hardware, while providing programmers a large set of instructions for common operations. The MIPS mul instruction is a pseudoinstruction implemented using mult and mflo native instructions, discussed elsewhere.*

PARTICIPATION ACTIVITY | 2.7.3: sub and mul instructions.

The assembly program below calculates the total taxi fare as $15 plus $2 per mile traveled. DM[5000] holds the total miles traveled, and the total fare is stored in DM[5004].

1. Run the simulation step-by-step, observing memory values.
2. Change DM[5000]'s value to 25, then run again.
3. Modify the program to subtract $2 for a frequent rider discount, store the discounted total fare in DM[5008].

Instruction format summary: sub, mul

Table 2.7.1: Instruction summary: sub, mul.

Instruction	Format	Description	Example
sub	`sub $a, $b, $c`	Subtract: Subtracts $c from $b, and writes the difference into register $a.	`sub $t3, $t2, $t5`
mul	`mul $a, $b, $c`	Multiply: Multiplies register $b and $c, and writes the lower 32-bits of the product into register $a.	`mul $t3, $t2, $t1`

CHALLENGE ACTIVITY | 2.7.1: Subtract and multiply instructions.

459784.3174716.qx3zqy7

2.8 beq, bne, j: Branch and jump instructions

Branch instructions: beq, bne

A ***branch*** instruction specifies the location of the next instruction to execute, depending on the branch instruction's condition. A ***branch on equal*** (***beq***) instruction branches to an instruction at a specified location if the values held in two registers are equal. If the values are equal, the branch is taken, and the instruction at the specified location is executed. Otherwise, the branch is not taken, and the instruction immediately following the branch instruction is executed.

A branch instruction typically uses a label to specify the next instruction's location. A ***label*** is a named position in a program that represents an instruction's memory address. The MIPS beq instruction format below branches to the instruction at location Label if the values held in regA and regB are equal.

```
beq regA, regB, Label
```

A label is a sequence of letters (a-z, A-Z, _) and digits (0-9) starting with a letter and followed by a colon (:).

PARTICIPATION ACTIVITY 2.8.1: Branch on equal (beq) instruction.

Animation captions:

1. The beq instruction compares the values held in $t1 and $t0. 5 is equal to 5, so the execution will branch to the instruction following the label Cont.
2. The label Cont represents the addi instruction's memory address. Labels allow a program to branch to instructions without yet knowing the instruction's address.
3. Execution continues with the addi instruction. The add instruction was not executed, because the branch was taken.

PARTICIPATION ACTIVITY 2.8.2: beq instruction.

Which instruction is executed immediately after the branch instruction.
Assume initial register values of:

- $t0: 5
- $t1: 10
- $t2: 0
- $t3: 10

1)
```
   beq $t1, $t3, Cont
   sub $t1, $t1, $t5

Cont: sw $t4, 0($t6)
```

2)
```
   beq $t0, $t1, Cont
   sw $t1, 0($t5)

Cont:  addi $t1, $t1, -2
```

3)

```
    beq $t2, $zero, Cont
    addi $t4, $t4, 11
    sw $t4, 0($t6)
Cont:  lw $t2, 0($t6)
```

4)

```
    beq $t3, $t1, Cont
    addi $t3, $t3, 2
Cont:
    sub $t3, $t3, $t5
```

PARTICIPATION ACTIVITY 2.8.3: Labels.

Which are valid labels for the addi instruction?

1)

```
Cont: addi $t2, $t2, 1
```

○ Valid

○ Invalid

2)

```
After_Adjust: addi $t2, $t2, 1
```

○ Valid

○ Invalid

3)

```
userValEq3: addi $t2, $t2, 1
```

○ Valid

○ Invalid

4)

```
IsGood?: addi $t2, $t2, 1
```

○ Valid

○ Invalid

5)

```
Grade equals 100: addi $t2,
$t2, 1
```

○ Valid

○ Invalid

6)

```
CheckResult:
   addi $t2, $t2, 1
```

○ Valid

7) ○ Invalid

```
2ndTask: addi $t2, $t2, 1
```

○ Valid

○ Invalid

A ***branch on not equal*** (***bne***) instruction branches to an instruction at a specified location if the values held in two registers are not equal. The MIPS bne instruction format below branches to the instruction at Label if the values held in regA and regB are not equal.

```
bne regA, regB, Label
```

PARTICIPATION ACTIVITY | 2.8.4: Branch instructions: bne and beq.

For each question, assume initial register values of:

- $t0: 20
- $t1: 15
- $t2: 15
- $t3: 21

1) After the following, what is $t3?

```
   bne $t0, $t1, Cont
   addi $t3, $t3, 5
Cont: addi $t2, $t2, 2
```

Check **Show answer**

2) After the following, what is $t3?

```
   bne $t1, $t2, Cont
   addi $t3, $t3, 7
Cont: addi $t2, $t2, 3
```

Check **Show answer**

3) After the following, what is $t2?

```
   bne $t1, $t2, Cont
   addi $t3, $t3, 7
Cont: addi $t2, $t2, 3
```

Check **Show answer**

4) After the following, what is $t3?

```
   bne $t2, $t1, Cont
   addi $t3, $t3, 8
Cont: addi $t3, $t3, 4
```

Check **Show answer**

5) How many instructions execute in the following?

```
   bne $t2, $t0, Cont1
   addi $t3, $t3, 8
Cont1:
   bne $t2, $t1, Cont2
   addi $t3, $t3, 5
Cont2:
   addi $t3, $t3, 7
```

Check **Show answer**

Jump instruction

A ***jump*** (***j***) instruction specifies the location of the next execution to execute. A jump instruction is also known as an unconditional branch. The MIPS j instruction format below jumps to the instruction at Label.

```
j Label
```

PARTICIPATION ACTIVITY 2.8.5: Jump (j) instruction.

Animation captions:

1. The jumps instruction specifies the location of the next execution to execute, so the addi instruction is executed after the jump.

PARTICIPATION ACTIVITY 2.8.6: Jump instructions.

1) A jump instruction will always jump to the labeled instruction.

○ True

○ False

2) A jump instruction can only jump to a

labeled instruction located after the jump instruction.

○ True

○ False

3) Which instruction is executed after the jump instruction?

```
j Comp2
Comp1: addi $t2, $zero, -5
Comp2: sw $t3, 0($t5)
```

○ addi

○ sw

Branch and jump instructions are commonly used together to direct a program to conditionally execute either one group of instructions or another group, but not both. A branch instruction is used to decide which group of statements to execute. If the branch is taken, the instruction group at the label specified in the branch is executed. If the branch is not taken, the instruction group after the branch is executed. That instruction group ends with a jump instruction to the first instruction after the other instruction group, so the other instruction group is not executed.

PARTICIPATION ACTIVITY | 2.8.7: Using branch and jump instructions to execute one of two instruction groups.

Animation captions:

1. If the branch is taken, the instruction group at the label Equal is executed.
2. Then, execution continues with the instructions after the label After.
3. If the branch is not taken, the instruction group after the branch instruction executed.
4. The jump instruction jumps to the label After, which is after the instructions that would execute if the branch was taken. So, only one of the instruction groups is executed.

PARTICIPATION ACTIVITY | 2.8.8: Branch and jump instructions.

Refer to the animation above.

1) Assume initial register values of $t1: 4, $t2: 7.

How many instructions are executed?

Check **Show answer**

2) Assume initial register values of $t1: 10, $t2: 10.

How many instructions are

executed?

Check Show answer

3) Assume initial register values of $t0: 5, $t1: 10, $t2: 10, $t3: 20. □

What is $t3 when execution reaches the label After?

Check Show answer

4) Assume initial register values of $t0: 5, $t1: 4, $t2: 18, $t3: 20. □

What is $t3 when execution reaches the label After?

Check Show answer

PARTICIPATION ACTIVITY 2.8.9: Branch and jump instruction example. □

The assembly program below adds 5 to DM[5004] if DM[5000] is 100. Otherwise, the program adds 10 to DM[5004]. The sum is stored in DM[5008].

1. Run the simulation step-by-step, observing memory values.
2. Change DM[5000]'s value to 95, then run again.
3. Modify the program to add 5 to DM[5004] if DM[5000] is 100, add 10 if DM[5000] is 95, and add 20 otherwise.

Table 2.8.1: Instruction summary: beq, bne, j.

Instruction	Format	Description	Example
beq	`beq $a, $b, BLabel`	Branch on equal: Branches to the instruction at BLabel if the values held in $a and $b are equal. Otherwise, instruction immediately after beq is executed.	`beq $t3, $t2, SumEq5`
bne	`bne $a, $b, BLabel`	Branch on not equal: Branches to the instruction at BLabel if the values held in $a and $b are not equal. Otherwise, instruction immediately after bne is executed.	`bne $t4, $t5, GuessNeqCorrect`
j	`j JLabel`	Jump: Causes execution to continue with the instruction at JLabel.	`j CalcTip`

CHALLENGE ACTIVITY 2.8.1: Branch and jump instructions.

459784.3174716.qx3zqy7

2.9 slt: Set on less than instruction

Set on less than instruction

The ***set on less than*** (***slt***) instruction sets a register to 1 if the value held in the register is less than the value held in another register , otherwise the register is set to 0. Ex: `slt $t1, $t4, $t5` sets $t1 to 1 if $t4's value is less than $t5's value, otherwise $t1 is set to 0. The slt instruction is typically used with the beq or bne instructions to branch to an instruction based on the relational comparisons to two registers.

PARTICIPATION ACTIVITY 2.9.1: Set on less than instruction.

Animation captions:

1. The slt instruction compares the values held in $t0 and $t1. 5 is less than 10, so $t2 is "set" by writing 1 to $t2.
2. $t2 and $zero are not equal, so the bne instruction branches to BLabel.

PARTICIPATION ACTIVITY 2.9.2: Set on less than.

Assume initial register values of:

- $t0: 40
- $t1: 20
- $t2: 55

1) What is $t5 after the instruction below?

```
slt $t5, $t0, $t2
```

○ 0

○ 1

2) What is $t5 after the instruction below?

```
slt $t5, $t2, $t1
```

○ 0

○ 1

3) Determine if the branch instruction is taken or not taken.

```
slt $t4, $t0, $t2
beq $t4, $zero, BLabel
```

○ Taken

○ Not taken

4) Determine if the branch instruction is taken or not taken.

```
slt $t5, $t0, $t1
```

```
beq $t5, $zero, BLabel
```

○ Taken

○ Not taken

5) Determine if the branch instruction is taken or not taken.

```
slt $t6, $t1, $t2
bne $t6, $zero, BLabel
```

○ Taken

○ Not taken

Branch pseudoinstructions: blt, ble, bgt, bge

Common comparisons used for branch instructions include less than (<), less than or equal (≤), greater than (>), and greater than or equal (≥). However, MIPS natively supports only two branch instructions: beq (branch on equal) and bne (branch on not equal). An slt followed by either a beq or bne can implement any comparison <, ≤, >, ≥. However, the slt + beq/bne approach is non-intuitive to many programmers. Thus, MIPS supports the following pseudoinstructions.

- A ***branch on less than*** (***blt***) instruction branches if the first register is less than the second.
- A ***branch on less than or equal*** (***ble***) instruction branches if the first register is less than or equal to the second.
- A ***branch on greater than*** (***bgt***) instruction branches if the first register is greater than the second.
- A ***branch on greater than or equal*** (***bge***) instruction branches if the first register is greater than or equal to the second.

A MIPS assembler will convert each pseudoinstruction into the indicated two native instructions. The assembler uses a temporary register to store the result of an slt instruction, which is used in the following beq or bne instruction. The ***$at*** (or ***assembler temporary***) register is reserved for use by the assembler to hold temporary values needed to implement pseudoinstructions.

Table 2.9.1: Branch pseudoinstructions and equivalent native instructions.

Branch pseudoinstruction	Native instructions

`blt $t0, $t1, BLabel`	`slt $at, $t0, $t1`
	`bne $at, $zero, BLabel`

`ble $t0, $t1, BLabel`	`slt $at, $t1, $t0`
	`beq $at, $zero, BLabel`

`bgt $t0, $t1, BLabel`	`slt $at, $t1, $t0`
	`bne $at, $zero, BLabel`

`bge $t0, $t1, BLabel`	`slt $at, $t0, $t1`
	`beq $at, $zero, BLabel`

PARTICIPATION ACTIVITY | 2.9.3: Branch pseudoinstructions.

Refer to the above table.

1) beq is a pseudoinstruction.

○ True

○ False

2) slt is a native instruction.

○ True

○ False

3) A blt $t0, $t1 is replaced by an slt $at, $t0, $t1 followed by a bne.

○ True

○ False

4) A bgt $t0, $t1 is replaced by an slt $at, $t0, $t1 followed by a bne.

○ True

○ False

5) A bge $t0, $t1 is replaced by an slt $at, $t0, $t1 followed by a beq.

○ True

○ False

6) A ble $t0, $t1 is replaced by an slt $at, $t1, $t0 followed by a beq.

○ True

○ False

Table 2.9.2: Instruction summary: slt.

Instruction	Format	Description	Example
slt	`slt $a, $b, $c`	Set on less than: Write 1 to register $a if value held in register $b is less than value held in register $c, and otherwise writes 0.	`slt $t1, $t5, $t6`

CHALLENGE ACTIVITY | 2.9.1: Set on less than, and branch pseudoinstructions.

459784.3174716.qx3zqy7

2.10 Input / output

To be useful, a computer system needs to produce output that a user can examine. Ex: A program that outputs a table of powers of 2 (1, 2, 4, 8, 16, ...) needs a way to provide that table to a user. Typical output methods include screens and files. MIPSzy uses the following simple output approach:

- To output an integer, a program writes the integer to memory location 8200.
- Special hardware detects a write to location 8200, and automatically outputs that integer to a screen.

Likewise, a computer system needs to accept user input. Ex: A program may accept user input, then output 2 times that input. Typical input methods include keyboards and files. MIPSzy uses the following simple input approach:

- When input is available, special hardware puts the next input value into memory location 8196, and sets location 8192 with 1.
- A program can load from location 8196 to get the next input value (typically after checking that location 8192 is 1). The hardware detects such a load, and automatically puts the next input value into 8196 (or if no further input exists, sets 8192 with 0).

PARTICIPATION ACTIVITY | 2.10.1: MIPSzy output and input: Stores to 8200 are also automatically output to the screen. Input is automatically put in 8196.

Animation captions:

1. For output, the program stores an integer at location 8200. Special hardware automatically outputs that integer to the screen at the current cursor position.
2. For input, special hardware puts an input integer in location 8196, and puts 1 in 8192 indicating an input value is ready.
3. A program can load 8196 to get the input value. The hardware detects the load and automatically puts the next input value into 8196 (or if no more input, sets 8192 with 0).

The program below outputs the first few powers of 2 (1, 2, 4, 8) to the screen.

PARTICIPATION ACTIVITY | 2.10.2: MIPSzy output.

PARTICIPATION ACTIVITY | 2.10.3: MIPSzy output.

1) To output 9 to the screen, a program should store 9 into memory location _____ .

○ 8192

○ 8200

2) If a program should output 3 2 1 to the

screen, the program should store 3 in 8200, 2 in 8204, and 1 in 8208.

○ True

○ False

3) To output the letter 'c' on the screen, the program should store the ASCII value of 'c', namely 99, into location 8200.

○ True

○ False

The program below reads input values as long as another value exists, outputting 2 times each value.

PARTICIPATION ACTIVITY 2.10.4: Input and output.

PARTICIPATION ACTIVITY 2.10.5: MIPSzy input.

1) An input integer from a keyboard is automatically placed by special hardware into memory location _____ .

○ 8192

○ 8196

2) A program _____ store values into 8196 using sw instructions.

○ should

○ should not

3) When a program loads a value from 8196, the hardware automatically puts the next input value into 8196.

○ True

○ False

4) If 8192 holds 0, then 8196 ____ the next input value.

○ has

○ does not have