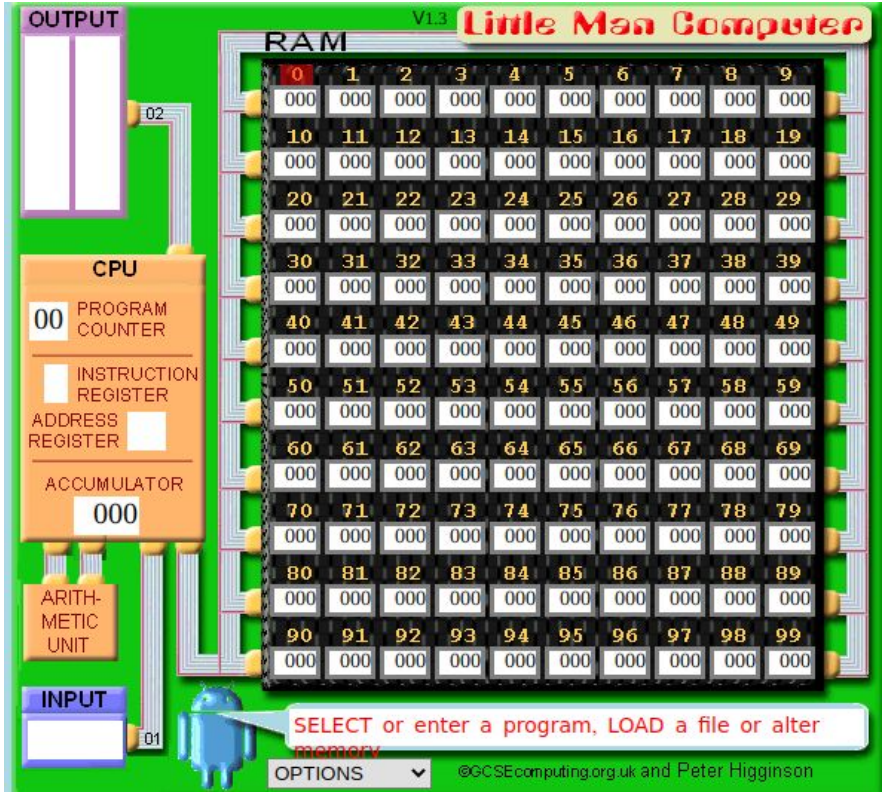

MONTANA
STATE UNIVERSITY

MIPS Assembly

Beyond the LMC

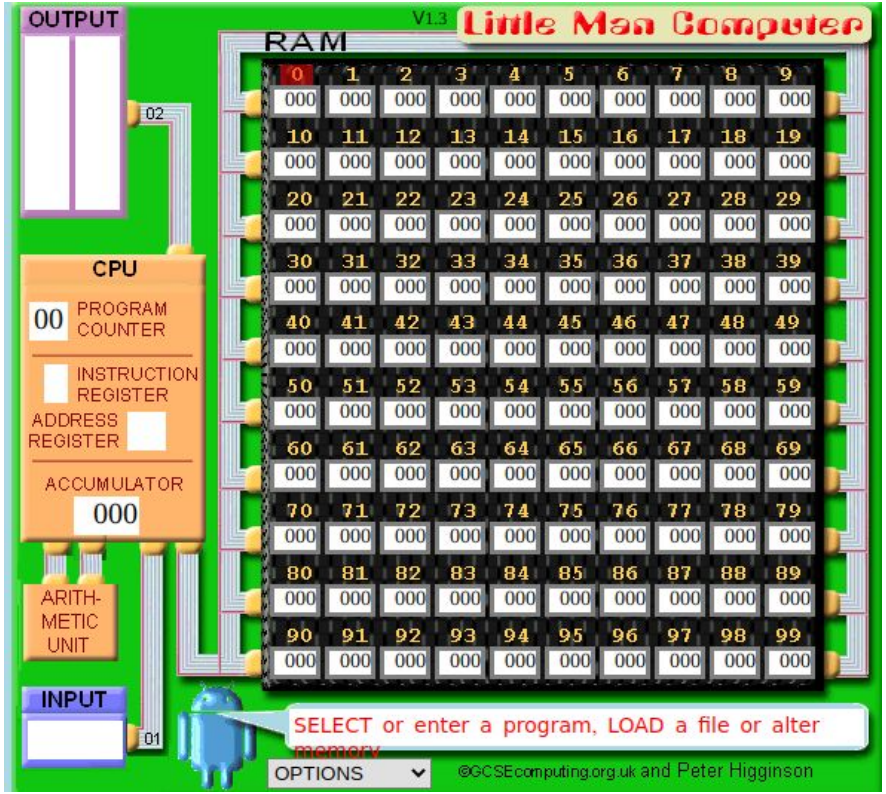
LMC Review

- Recall the LMC architecture
  - Program Counter
  - Instruction & Address registers
  - Accumulator register for work
  - Input/Output areas
  - An ALU
  - 100 memory slots



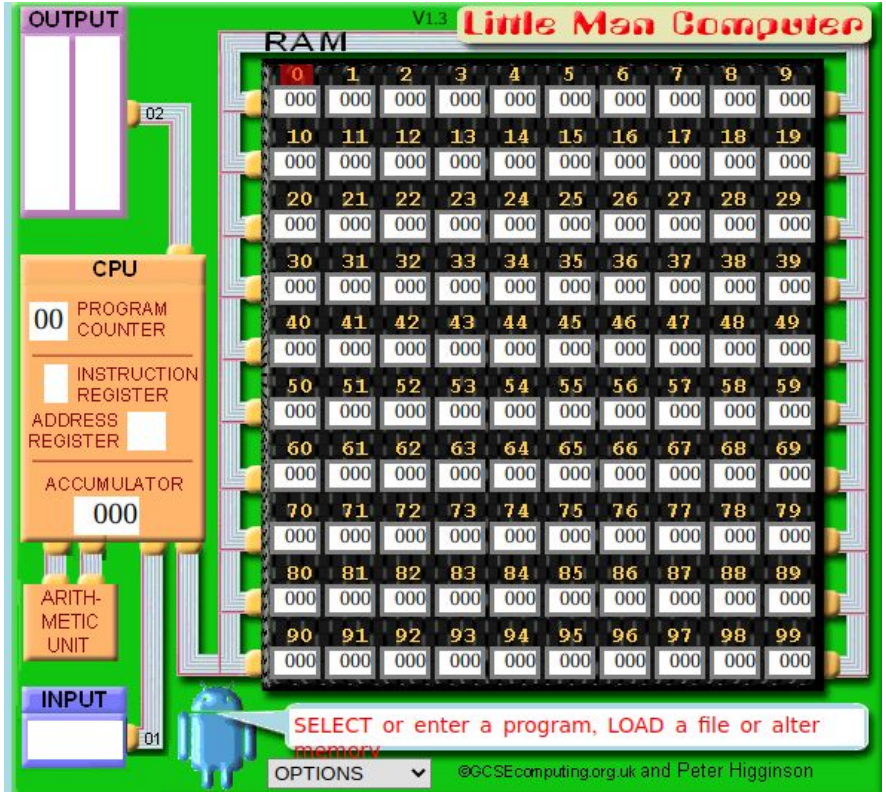
LMC Review

- LMC Execution Cycle
  - Check the Program Counter
  - Fetch the instruction from that address
  - Increment the Program Counter
  - Decode the fetched instruction into the Instruction and Address registers
  - Fetch any data needed
  - Execute the instruction
  - Branch or store the result
  - Repeat!



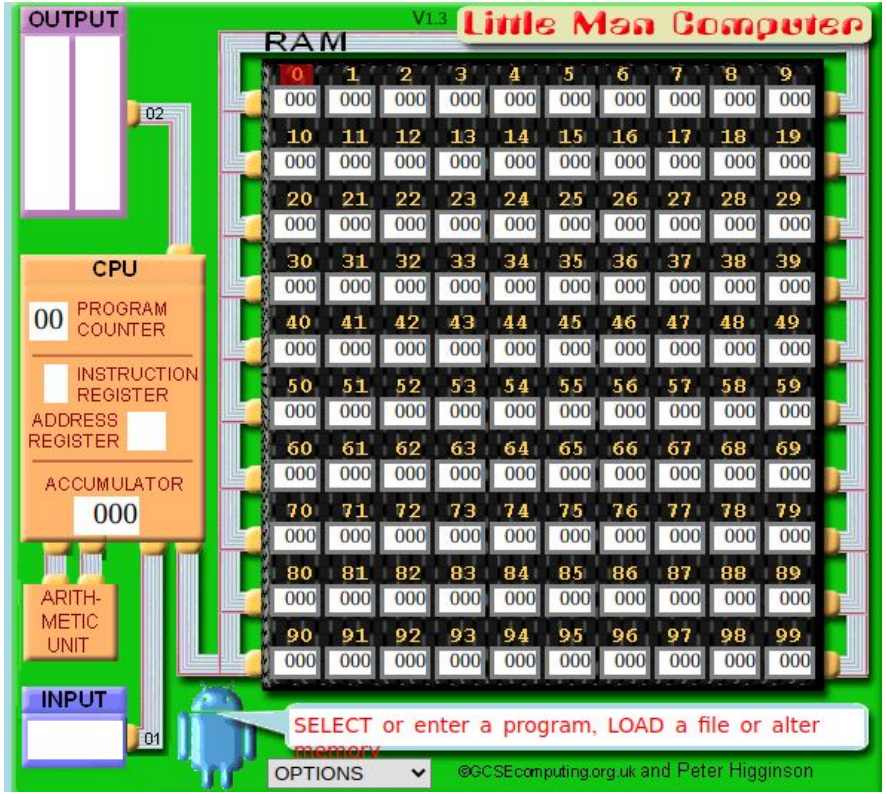
LMC Review

- LMC Instructions
  - ADD addition
  - SUB subtraction
  - STA store to memory
  - LDA load from memory
  - BRA unconditional branch
  - BRZ branch if zero
  - BRP branch if positive
  - INP get user input, put in acc
  - OUT output acc to output area
  - HLT/COB halt
  - DAT data



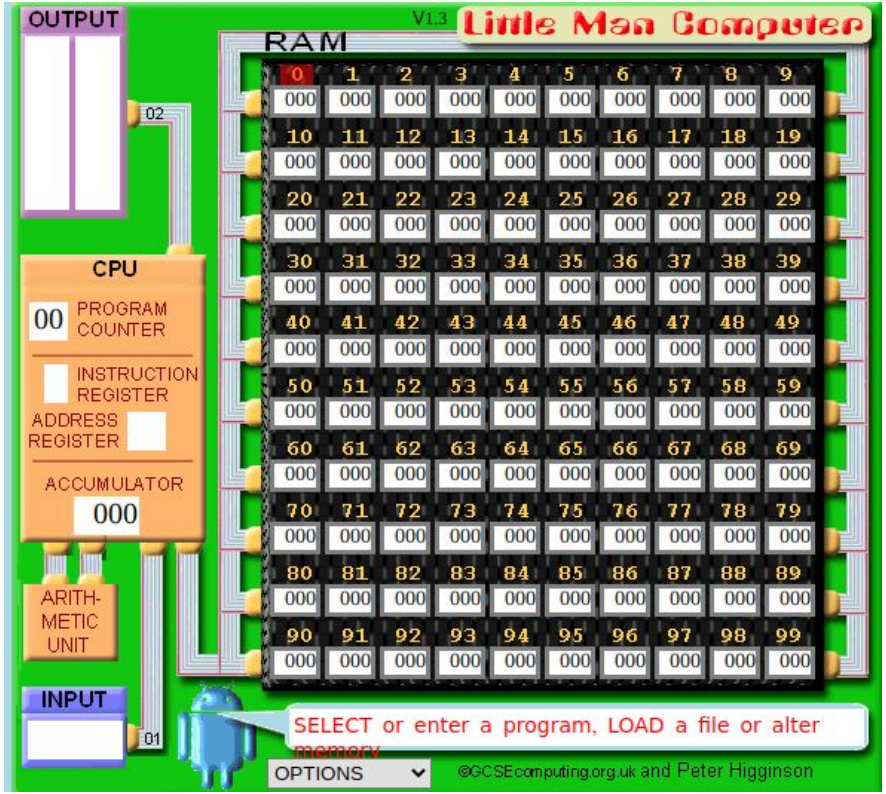
LMC Review

- This is a great teaching tool
- Reflects the layout of early hardware
- Modern hardware looks different
- We will move a step closer to modern hardware with a RISC simulator



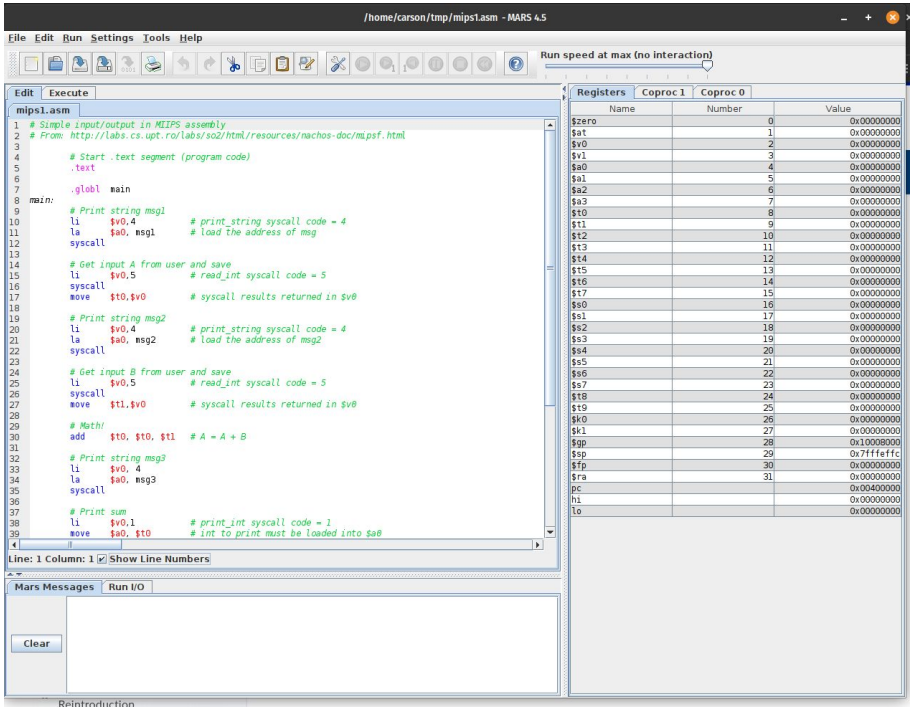
LMC Review

- This is a great teaching tool
- Reflects the layout of early hardware
- Modern hardware looks different
- We will move a step closer to modern hardware with a RISC simulator
- I will miss LMC...



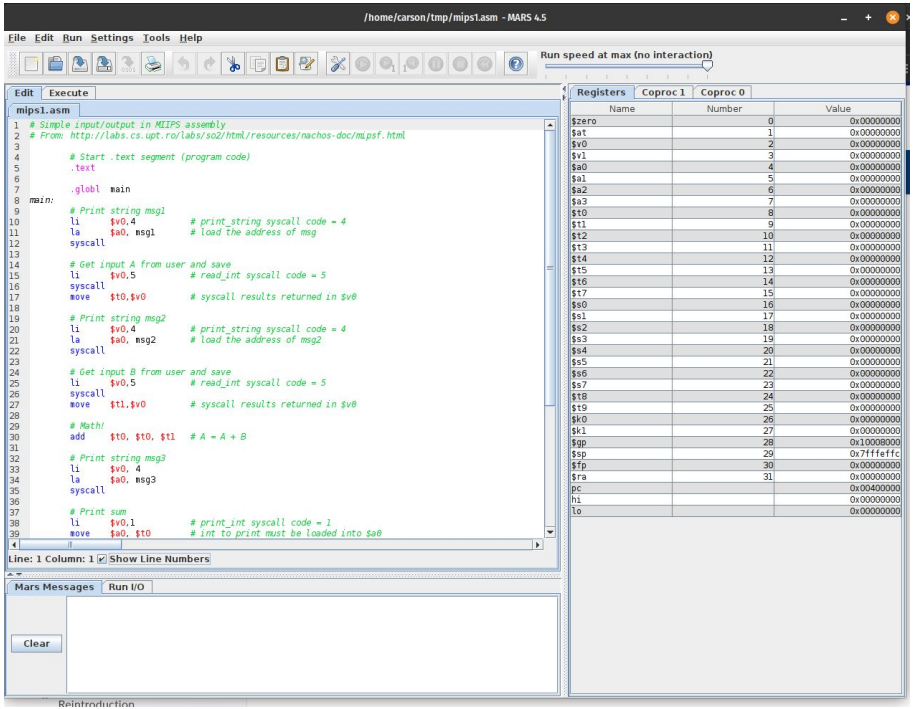
MIPS

- We now turn to MIPS, a common RISC architecture
  - RISC - Reduced Instruction Set Computer
    - Fewer instructions
    - More instructions needed to complete tasks
    - Simpler logic
  - Compared with CISC - Complex Instruction Set Computer
    - X86 is CISC



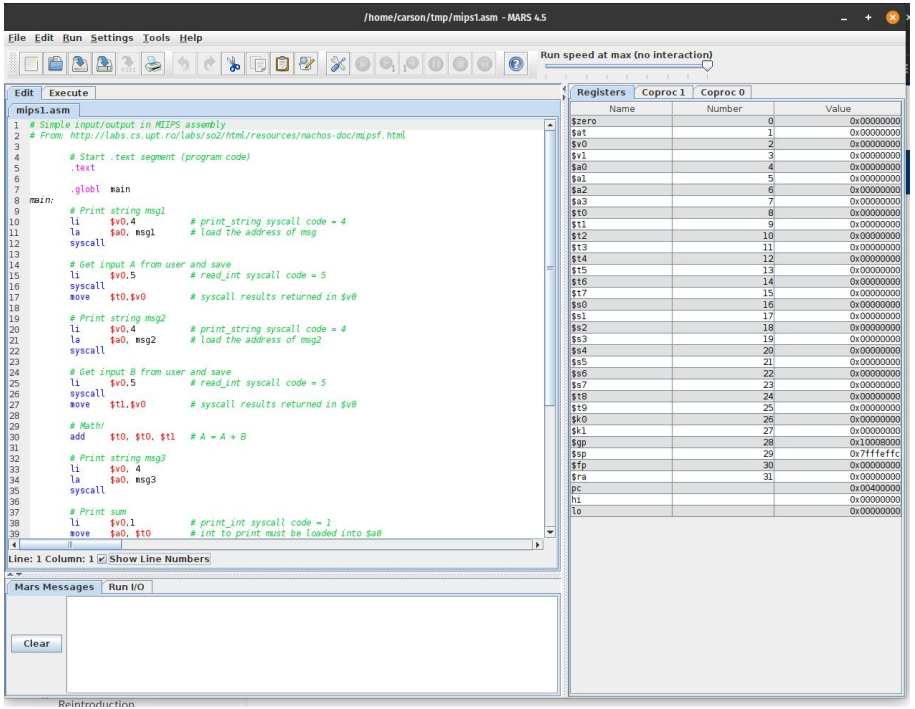
MIPS

- We will be looking at the 32 bit MIPS architecture
- MIPS was introduced in 1985 and is extremely popular in teaching institutions
- Despite its popularity in academia, it has had a bumpy road in the commercial space



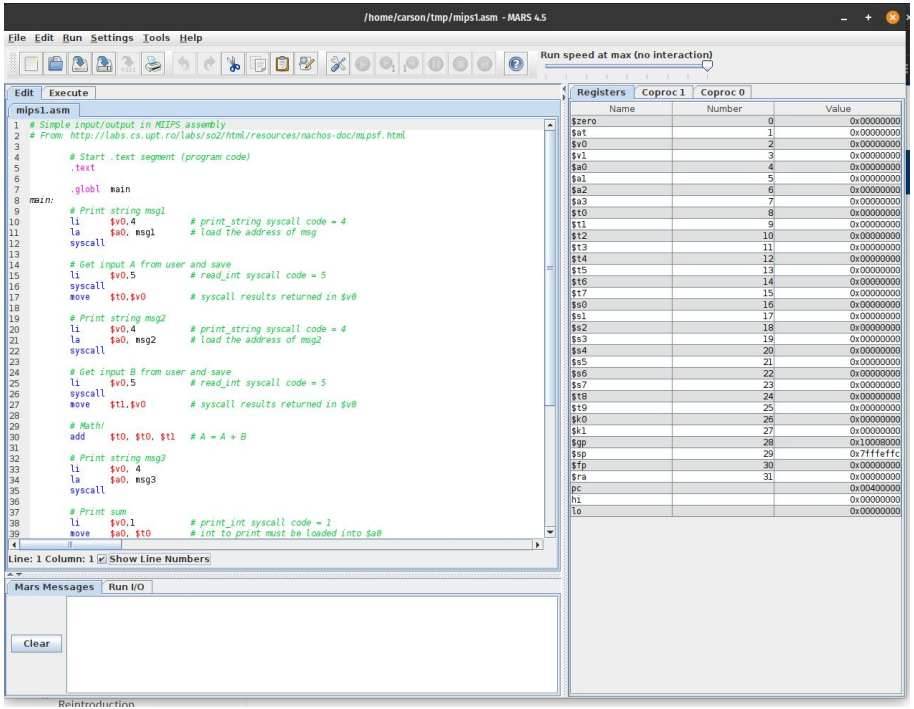
MIPS

- Despite not being popular in industry, the architecture has been extremely influential
- ARM64, in particular, shares a lot of features with MIPS
  - ARM32 did not
- With the advent of apple's M4 chip, MIPS ideas are finally breaking into mainstream computing



MIPS

- Differences with LMC
  - MIPS has 32 general purpose registers! vs 1 accumulator
  - Some registers have special functions, so you can't use all of them
  - 32 bit memory space
  - Many more instructions



MIPS Registers

- Registers have a number as well as a name associated with them
- You will program using the name, in general
- $zero is hardwired to zero
- $at - temporary, used for pseudo-instructions

Register Number	Register Name	Function
$0	$zero	Hard-wired to 0
$1	$at	Assembler temporary, reserved for pseudo-instructions
$2-$3	$v0, $v1	Function return values
$4-$7	$a0-$a3	Function parameters, not preserved by sub-programs
$8-$15	$t0-$t7	Temporary data, not preserved by subprograms
$16-$23	$s0-$s7	Saved registers, preserved by subprograms
$24-$25	$t8, $t9	Temporary data, not preserved by subprograms
$26-$27	$k0, $k1	Reserved for interrupt handler. Don't use!
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

MIPS Registers

- $v0, $v1 - function return values
- $a0-a3 - function parameters (not preserved)
- $t0-t9 - temporary values (not preserved)
- $s0-s7 - temporary values (preserved)

Register Number	Register Name	Function
$0	$zero	Hard-wired to 0
$1	$at	Assembler temporary, reserved for pseudo-instructions
$2-$3	$v0, $v1	Function return values
$4-$7	$a0-$a3	Function parameters, not preserved by sub-programs
$8-$15	$t0-$t7	Temporary data, not preserved by subprograms
$16-$23	$s0-$s7	Saved registers, preserved by subprograms
$24-$25	$t8, $t9	Temporary data, not preserved by subprograms
$26-$27	$k0, $k1	Reserved for interrupt handler. Don't use!
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

MIPS Registers

- $k0, $k1 - used for interrupt handling by the OS & hardware, DO NOT USE
- $gp - global area pointer (points to heap memory)
- $sp - stack pointer (points to stack memory)

Register Number	Register Name	Function
$0	$zero	Hard-wired to 0
$1	$at	Assembler temporary, reserved for pseudo-instructions
$2-$3	$v0, $v1	Function return values
$4-$7	$a0-$a3	Function parameters, not preserved by sub-programs
$8-$15	$t0-$t7	Temporary data, not preserved by subprograms
$16-$23	$s0-$s7	Saved registers, preserved by subprograms
$24-$25	$t8, $t9	Temporary data, not preserved by subprograms
$26-$27	$k0, $k1	Reserved for interrupt handler. Don't use!
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

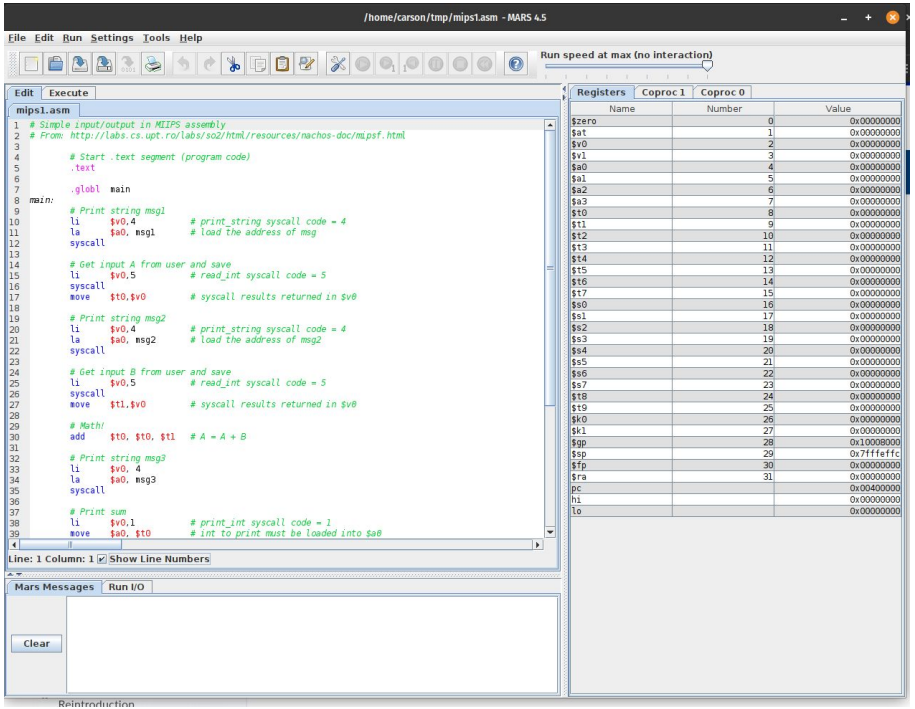
MIPS Registers

- $fp - Frame pointer (related to the stack pointer)
- $ra - Return address (will make sense when we talk about how function calls work)

Register Number	Register Name	Function
$0	$zero	Hard-wired to 0
$1	$at	Assembler temporary, reserved for pseudo-instructions
$2-$3	$v0, $v1	Function return values
$4-$7	$a0-$a3	Function parameters, not preserved by sub-programs
$8-$15	$t0-$t7	Temporary data, not preserved by subprograms
$16-$23	$s0-$s7	Saved registers, preserved by subprograms
$24-$25	$t8, $t9	Temporary data, not preserved by subprograms
$26-$27	$k0, $k1	Reserved for interrupt handler. Don't use!
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

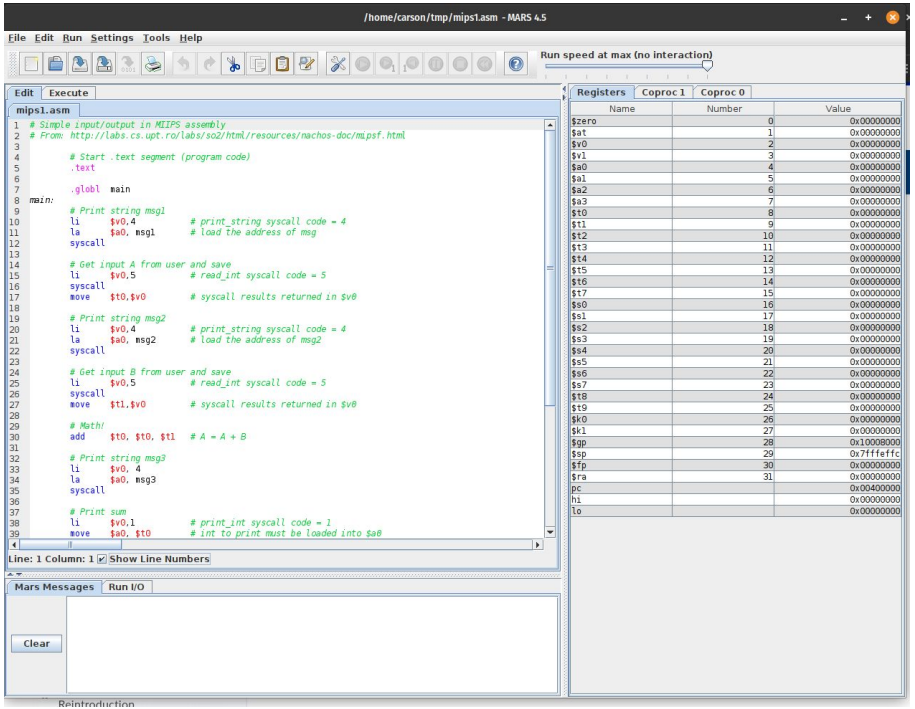
MIPS

- We will be using the MARS MIPS emulator
  - Written in Java
  - Provides an editor with some autocompletion
  - Allows you to step through assembly instructions and see what's happening in the computer



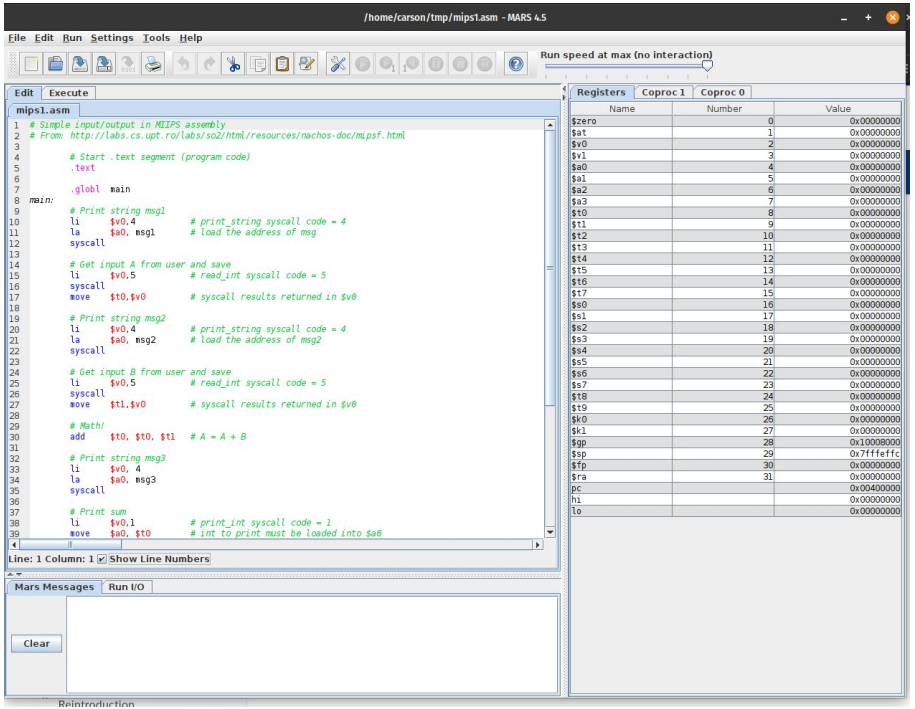
MIPS

- Similarities to LMC
  - Same basic instruction cycle
    - Fetch instruction
    - Decode
    - Execute
    - Increment Program Counter register (PC)
  - The PC is visible in the MARS UI



MIPS

- Similarities to LMC
  - MIPS is still a *Load/Store* architecture
  - Data must be loaded into a register from memory
  - Mutation operations (+, -, etc.) are performed only between registers
  - Data is then stored back to memory



MIPS Basic Instructions

- MIPS Instructions come in three major flavors:
  - R format (registers) - three register arguments
  - I format (immediate) - two register arguments + an immediate value
  - J format (jump) - no register arguments, 26 bits for address information

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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 Basic Instructions

- MIPS Instructions
  - LW - load 32-bit word to register
  - SW - store from register
  - ADD, SUB - signed math operators
    - Immediat versions too
  - MOVE - copy from register to register
    - MOVE is a *pseudo instruction* and becomes an ADD under the covers

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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 Basic Instructions

- MIPS Instructions
  - AND, OR, NOR - bitwise logical operators
    - Immediate versions available as well

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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 Immediates

- Immediates
  - I-style instructions
  - Values are encoded in the instruction directly
  - Only two registers are involved in the computation:

    ADDI $v0, $v0, 1

    “Add one to the value in $v0 and save it back to $v0”

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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 Branch Instructions

- Branch Instructions in MIPS
  - BEQ - Branch On Equal
  - BNE - Branch On Not Equal
  - J - Jump
  - JAL - Jump and link (function related)
  - JR - Jump based on the value of a register

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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

Psuedoinstructions

- Many common instructions in MIPS assembly do not correspond 1-1 with machine instructions
- Rather, they are transformed into other instructions
- These are called pseudoinstructions

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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

Psuedoinstructions

- To the assembly programmer, they look and act like regular expressions
- They make working in MIPS assembly more pleasant, without adding additional complexity to the circuitry

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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

Psuedoinstructions

- Examples:
  - MOVE
  - BLT - Branch Less Than
  - BGT - Branch Greater Than
  - ...
  - LI - Load immediate value
  - LA - Load address
- For example

  LI $v0, <value>

  becomes

  ADDI $v0, $zero, <value>

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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

- So, a lot more going on, but the basics of assembly remain the same
- A lot of the complexity here is around supporting function calls, which we will discuss in our next lecture

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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

I/O In MIPS

- Note that unlike LMC, we do not have instructions for getting input directly from the user
- Instead, to do this, we will need to make *system calls*
- This is a more realistic model for modern computers

```
1
2   .text
3   main:
4     li $v0, 4 # print
5     la $a0, hello
6     syscall
7
8     li $v0, 10  # exit
9     syscall
10
11  .data
12  hello:   .asciiz "Hello World"
13
```

System Calls

- A *syscall* in general is a call to the system (typically the OS) to perform some computation
- This is typically done with some sort of *calling convention*
  - A register is used to select which system call to make
  - Arguments are passed via standard registers
    - Recall on MIPS, $a0-a4

```
1
2   .text
3   main:
4     li $v0, 4 # print
5     la $a0, hello
6     syscall
7
8     li $v0, 10  # exit
9     syscall
10
11  .data
12  hello:   .asciiz "Hello World"
13
```

Storing Data

- MIPS, like most modern assembly, splits code and data off into a separate areas
- .text is a code area
- .data is a data area
- Note that labels work as they do in LMC
  - The LA instruction loads the address of a given label

```
1
2   .text
3   main:
4     li $v0, 4 # print
5     la $a0, hello
6     syscall
7
8     li $v0, 10  # exit
9     syscall
10
11  .data
12  hello:   .asciiz "Hello World"
13
```

I/O In MIPS

- So, here we have a simple “Hello World” assembly program
- We load the immediate value 4 into $v0
  - $v0 communicates which system call to make
- We then load the address of the string into $a0, to pass through to the system

```
1
2   .text
3   main:
4     li $v0, 4 # print
5     la $a0, hello
6     syscall
7
8     li $v0, 10  # exit
9     syscall
10
11  .data
12  hello:   .asciiz "Hello World"
13
```

I/O In MIPS

- Finally we invoke the syscall instruction, which passes control over to the system
- The next line sets $v0 to 10, which is the system exit code and invoke the system once again, which ends the program

```
1
2   .text
3   main:
4     li $v0, 4 # print
5     la $a0, hello
6     syscall
7
8     li $v0, 10  # exit
9     syscall
10
11  .data
12  hello:   .asciiz "Hello World"
13
```

I/O In MIPS

- Not so bad, right?
- OK, how do we get input?
- Another syscall
  - System call 5: read int
  - Results are saved to $v0
- We move this value to a temp register so we can then print out a new string, and then the integer that was entered...

```
1
2   .text
3   main:
4     li $v0, 4 # print string
5     la $a0, enter
6     syscall
7
8     li $v0, 5 # read int
9     syscall
10
11    move $t0, $v0 # save to temporary reg
12
13    li $v0, 4 # print string
14    la $a0, you_entered
15    syscall
16
17    li $v0, 1 # print int
18    move $a0, $t0
19    syscall
20
21    li $v0, 10  # exit
22    syscall
23
24  .data
25  enter:  .asciiz "Enter a number: "
26  you_entered:    .asciiz "You entered: "
27
28
```

I/O In MIPS

- You can find all the syscalls available in MARS either in the Help menu, or online here:

  http://courses.missouristate.edu/KenVollmar/MARS/Help/SyscallHelp.html
-

```
1
2   .text
3   main:
4     li $v0, 4 # print string
5     la $a0, enter
6     syscall
7
8     li $v0, 5 # read int
9     syscall
10
11    move $t0, $v0 # save to temporary reg
12
13    li $v0, 4 # print string
14    la $a0, you_entered
15    syscall
16
17    li $v0, 1 # print int
18    move $a0, $t0
19    syscall
20
21    li $v0, 10  # exit
22    syscall
23
24  .data
25  enter:  .asciiz "Enter a number: "
26  you_entered:    .asciiz "You entered: "
27
28
```

MIPS Add One

- Let’s implement some of our LMC programs in MIPS
- Recall the LMC implementation for “Add One”

```
        INP
        ADD ONE
        OUT
        HLT
ONE     DAT 1
```

MIPS Add One

- Pretty much the same as our previous MIPS example
- We add the ADDI instruction to add an immediate value to the temporary register we are storing the value in
- A lot more book keeping, but this is much more realistic assembly

```
1
2  .text
3  main:
4    li $v0, 4 # print string
5    la $a0, enter
6    syscall
7
8    li $v0, 5 # read int
9    syscall
10
11   move $t0, $v0 # save to temporary r
12
13   addi $t0, $t0, 1  # add one
14
15   li $v0, 4 # print string
16   la $a0, you_entered
17   syscall
18
19   li $v0, 1 # print int
20   move $a0, $t0
21   syscall
22
23   li $v0, 10  # exit
24   syscall
25
26 .data
27 enter:  .asciiz "Enter a number: "
28 you_entered:    .asciiz "Added 1: "
```

MIPS Count Down

- Print the numbers 10 to 1 in decreasing order
- LMC Code

```
        LDA TEN
LOOP    BRZ EXIT
        OUT
        SUB ONE
        BRA LOOP
EXIT    HLT
ONE     DAT 1
TEN     DAT 10
```

MIPS Count Down

- MIPS Implementation
  - Move 10 into $t0
  - Print via a syscall
  - Subtract 1 from $t0 (using an immediate value)
  - Branch back to the loop if $t0 is greater than zero
- Note that immediates make things clearer

```
1
2   .text
3
4   li $t0, 10
5   loop:
6     li $v0, 1 # print int
7     move $a0, $t0
8     syscall
9     subi $t0, $t0, 1
10    bgt $t0, $zero, loop
11
12    li $v0, 10  # exit
13    syscall
```

MAX

- Max: “Ask the user for two numbers and print the maximum of the two”
- Recall that this was difficult in LMC because we had two values plus a difference value in play
  - Requires storing working data to memory

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

MIPS MAX

- Ask for two numbers via syscalls
- The crux is at line 20, where we only move the result of the second syscall into $t0 if it is greater than the current value in $t0
  - Otherwise we jump over to the skip label

```
1  .text
2
3  main:
4    li $v0, 4 # print string
5    la $a0, enter
6    syscall
7                          syscall  Issue a system
8    li $v0, 5 # read int
9    syscall
10
11   move $t0, $v0 # save to temp
12
13   li $v0, 4 # print string
14   la $a0, enter
15   syscall
16
17   li $v0, 5 # read int
18   syscall
19
20   bgt $t0, $v0, skip
21   move $t0, $v0 # v0 is greater
22 skip:
23
24   li $v0, 4 # print string
25   la $a0, max
26   syscall
27
28     li $v0, 1 # print int
29   move $a0, $t0
30   syscall
31
32   li $v0, 10  # exit
33   syscall
34
35 .data
36 enter:   .asciiz "Enter a number: "
37 max:     .asciiz "Max: "
38
```

MIPS MAX

- A little more difficult than LMC, but note that we didn't have to deal with memory at all
- More registers means things are easier in this regard

```
1  .text
2
3  main:
4    li $v0, 4 # print string
5    la $a0, enter
6    syscall
7                          syscall  Issue a system
8    li $v0, 5 # read int
9    syscall
10
11   move $t0, $v0 # save to temp
12
13   li $v0, 4 # print string
14   la $a0, enter
15   syscall
16
17   li $v0, 5 # read int
18   syscall
19
20   bgt $t0, $v0, skip
21   move $t0, $v0 # v0 is greater
22 skip:
23
24   li $v0, 4 # print string
25   la $a0, max
26   syscall
27
28     li $v0, 1 # print int
29   move $a0, $t0
30   syscall
31
32   li $v0, 10  # exit
33   syscall
34
35 .data
36 enter:   .asciiz "Enter a number: "
37 max:     .asciiz "Max: "
38
```

MIPS

- OK, that's a brief introduction to MIPS assembly code
  - More realistic model of modern computers (still not perfect!)
  - Many more registers than LMC
  - Many more instructions
    - Especially the branching logic
  - Immediate values available with instructions
  - Syscalls for interacting with the system
- Again, remember: *IT'S JUST CODE*


MONTANA
STATE UNIVERSITY