MIPS Architecture

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware
- MIPS includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- MIPS is a *reduced instruction set computer* (RISC), with a small number of simple instructions

Instructions: Addition

C Code

```
a = b + c;
```

MIPS assembly code

```
add a, b, c
```

- **add:** mnemonic indicates operation to perform
- **b, c:** source operands (on which the operation is performed)
- **a:** destination operand (to which the result is written)

Instructions: Subtraction

- Similar to addition - only mnemonic changes

C Code
```
a = b - c;
```

MIPS assembly code
```
sub a, b, c
```

- **sub:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Multiple Instructions

- More complex code is handled by multiple MIPS instructions.

C Code
```
a = b + c - d;
```

MIPS assembly code
```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

Operands

- Operand location: physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

Operands: Registers

- MIPS has 32 nos of 32-bit registers
- Registers are faster than memory
- MIPS called "32-bit architecture" because it operates on 32-bit data

MIPS Register Set

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	Function return values
$a0-$a3	4-7	Function arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	Function return address

Operands: Registers

- Registers:
  - $ before name
  - Example: $0, "register zero", "dollar zero"
- Registers used for specific purposes:
  - $0 always holds the constant value 0.
  - the *saved registers*, $s0-$s7, used to hold variables
  - the *temporary registers*, $t0 - $t9, used to hold intermediate values during a larger computation

Instructions with Registers

C Code

```
a = b + c
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

Word-Addressable Memory

- Each 32-bit data word has a unique address

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Note: MIPS uses byte-addressable memory

Reading Word-Addressable Memory

- Memory read called *load*
- **Mnemonic:** *load word* (`lw`)
- **Format:**

  `lw $s0, 5($t1)`

- **Address calculation:**
  - add *base address* (`$t1`) to the *offset* (5)
  - address = (`$t1` + 5)

- **Result:**
  - `$s0` holds the value at address (`$t1` + 5)

  Any register may be used as base address

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into $s3
  - address = ($0 + 1) = 1
  - $s3 = 0xF2F1AC07 after load

Assembly code
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Writing Word-Addressable Memory

- Memory write are called *store*

- **Mnemonic:** *store word* (`sw`)

- **Example:** Write (store) the value in $t4 into memory address 7
  - add the base address ($0) to the offset (0x7)
  - address: ($0 + 0x7) = 7

  Offset can be written in decimal (default) or hexadecimal

Assembly code

```
sw $t4, 0x7($0)   # write the value in $t4
                  # to memory word 7
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Operands: Constants/Immediates

- `lw` and `sw` use constants or *immediates*
- *immediate*ly available from instruction
- 16-bit two's complement number
- `addi`: add immediate
- Subtract immediate (`subi`) necessary?

C Code

```
a = a + 4;
b = a - 12;
```

MIPS assembly code

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- 3 instruction formats:
  - **R-Type:** register operands
  - **I-Type:** immediate operand
  - **J-Type:** for jumping (discuss later)

R-Type

- *Register-type*
- 3 register operands:
  - `rs, rt:` source registers
  - `rd:` destination register
- Other fields:
  - `op:` the *operation code* or *opcode* (0 for R-type instructions)
  - `funct:` the *function*
    with opcode, tells computer what operation to perform
  - `shamt:` the *shift amount* for shift instructions, otherwise it's 0

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type Examples

Assembly Code

add $s0, $s1, $s2

sub $t0, $t3, $t5

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code:

add rd, rs, rt

I-Type

- *Immediate-type*
- 3 operands:
  - `rs, rt:` register operands
  - `imm:` 16-bit two's complement immediate
- Other fields:
  - `op:` the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by opcode

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

I-Type Examples

Assembly Code

```
addi $s0, $s1, 5

addi $t0, $s3, -12

lw   $t2, 32($0)

sw   $s1,  4($t1)
```

Field Values

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4

| 6 bits | 5 bits | 5 bits | 16 bits |

Note the differing order of registers in assembly and machine codes:

```
addi rt, rs, imm

lw   rt, imm(rs)

sw   rt, imm(rs)
```

Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)

| 6 bits | 5 bits | 5 bits | 16 bits |

J-Type

- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)

J-Type

op	addr
6 bits	26 bits

Review: Instruction Formats

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits