CO20-320241

Computer Architecture and Programming Languages

CAPL

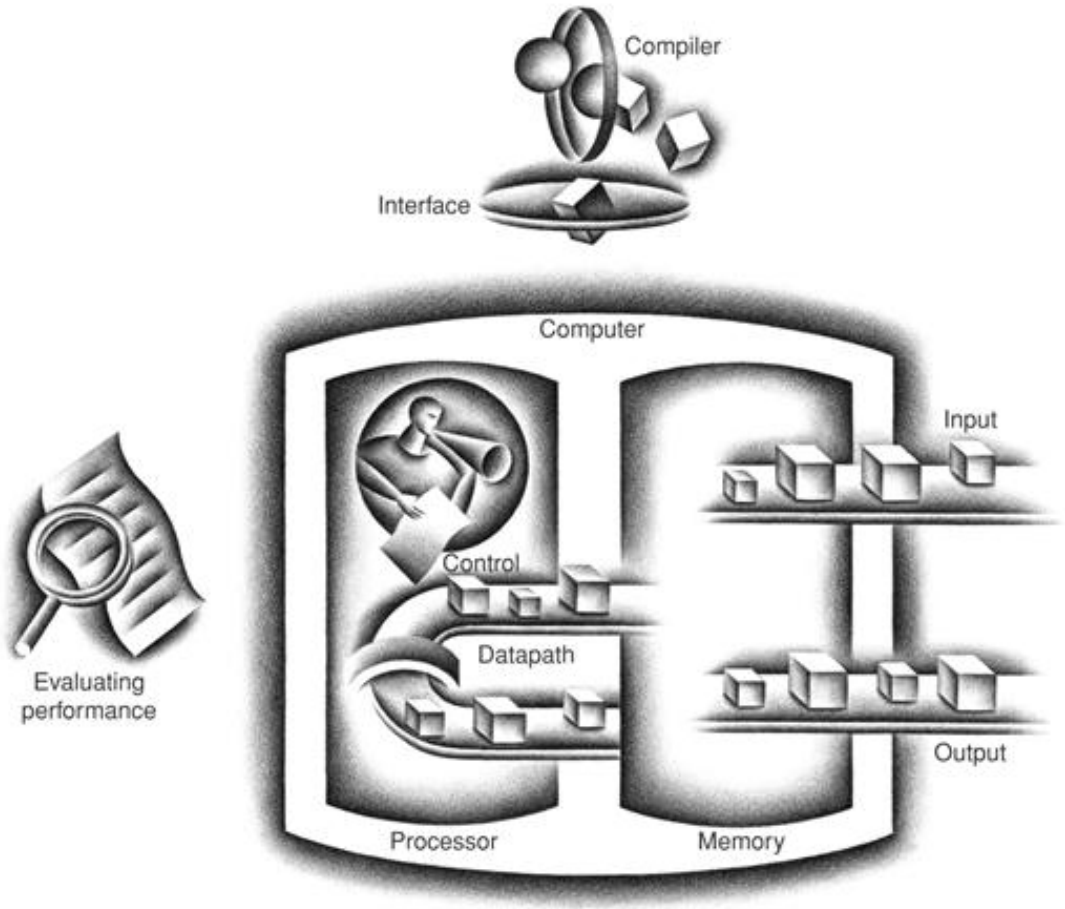
Lecture 10 & 11

Dr. Kinga Lipskoch

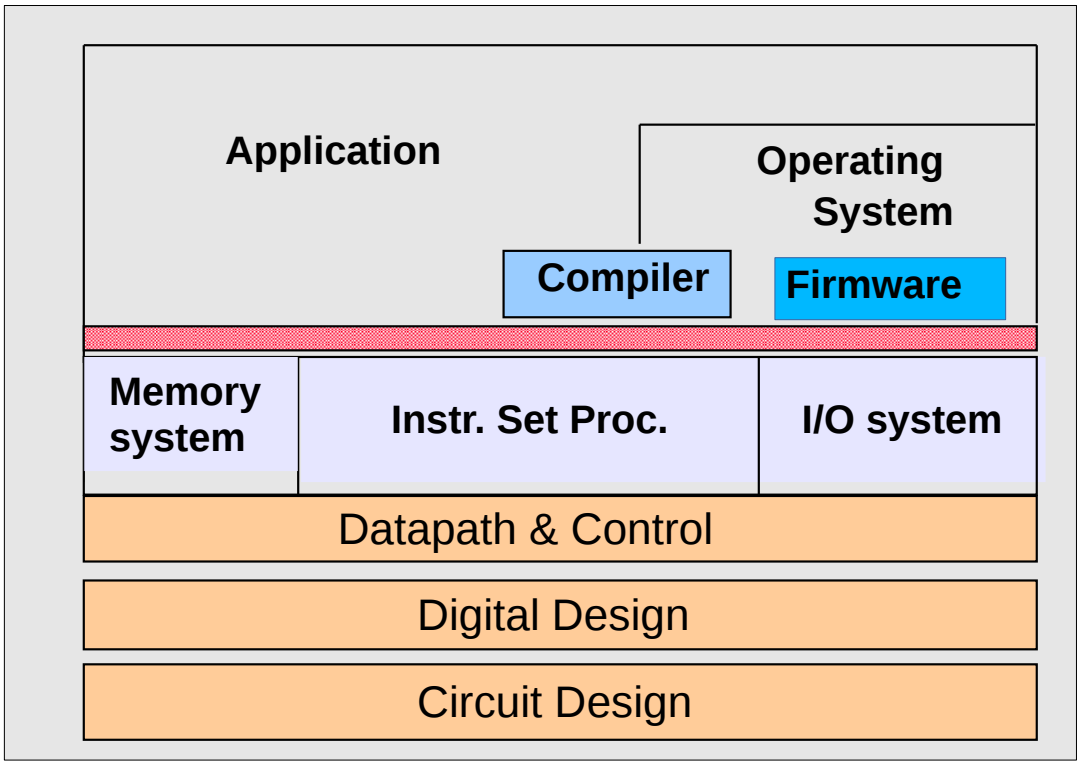
Fall 2019

Part II: Instruction Set Architectures

Five Classic components of a Computer



How do These Pieces Fit Together?

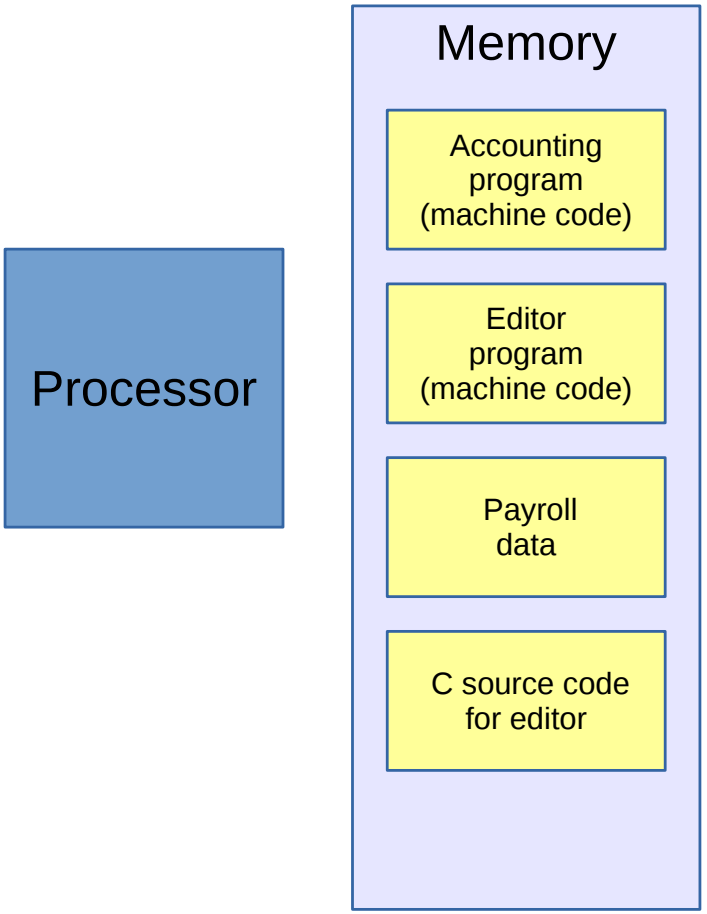


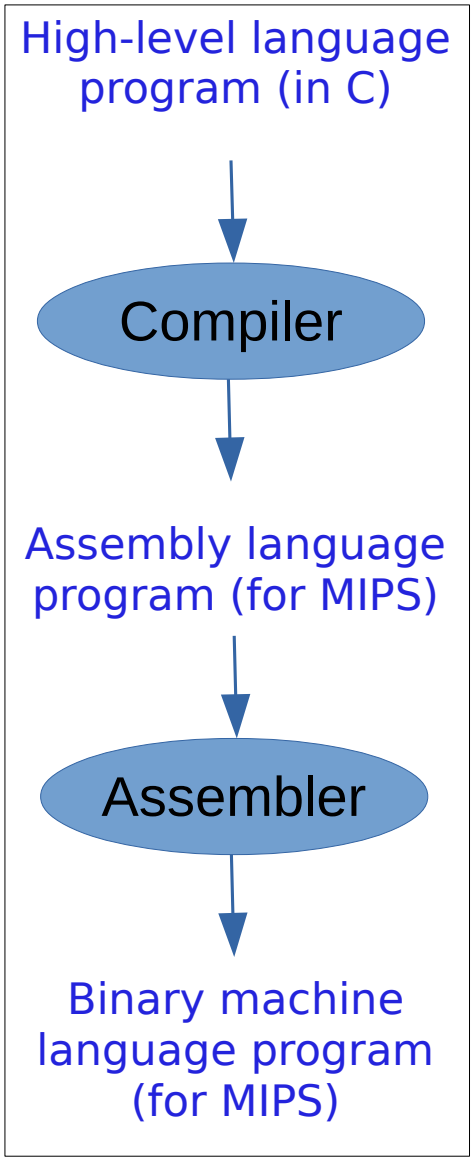
Instruction Set Architecture

Coordination of many levels of abstraction

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  - to be read or written just like data
- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" subsequent actions
  - Fetch the "next" instruction and continue





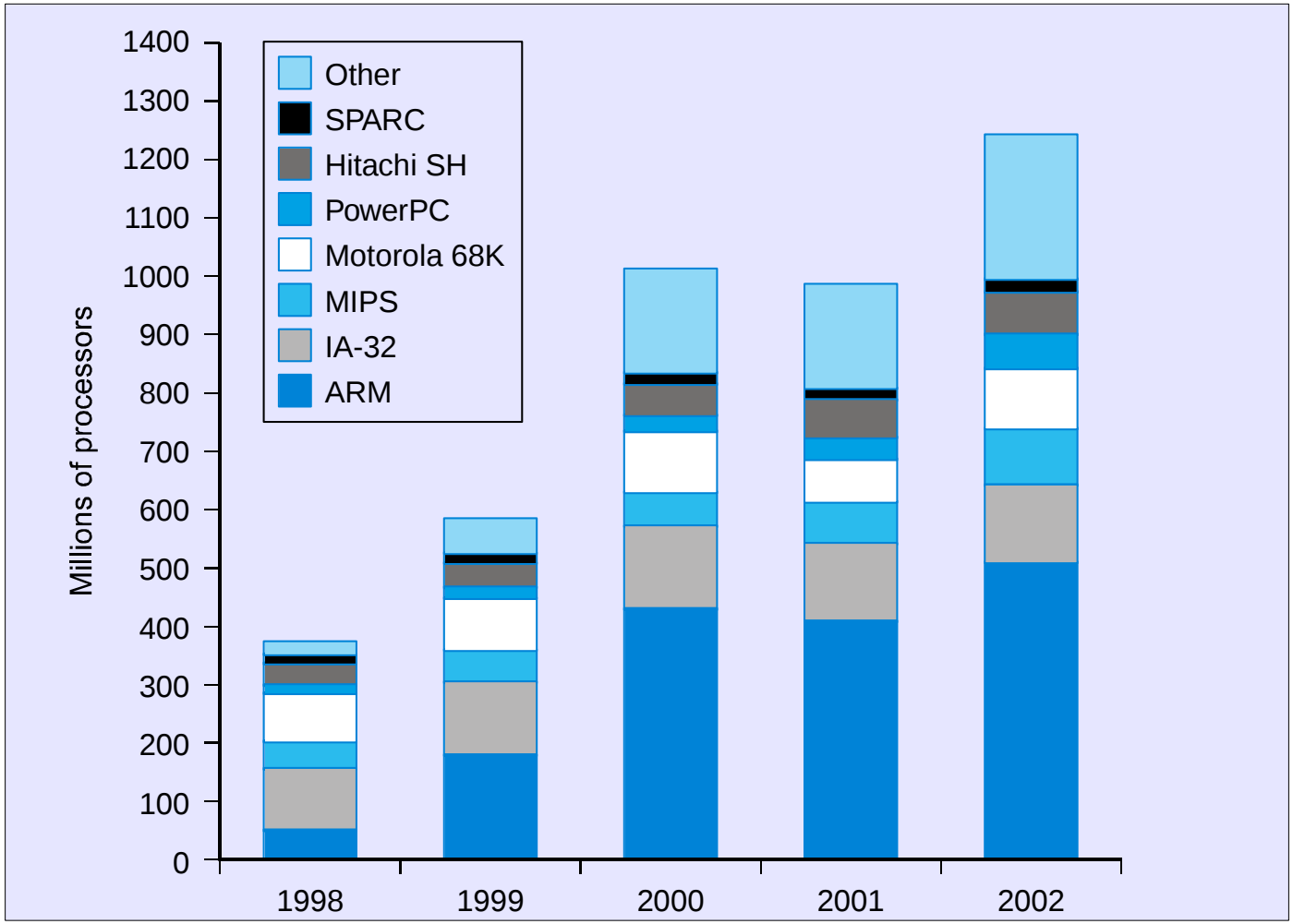

```
swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}


swap:
    muli   $2, $5, 4
    add    $2, $4, $2
    lw     $15, 0($2)
    lw     $16, 4($2)
    sw     $16, 0($2)
    sw     $15, 4($2)
    jr     $31


00000000101000010000000000011000
00000000000011000000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Instruction Set Architecture (ISA)

ISA: An abstract interface between the hardware and the lowest level software of a machine that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.

- "... the attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation." – Amdahl, Blaauw, and Brooks, 1964
- Enables implementations of varying cost and performance to run identical software

ISA Type Sales



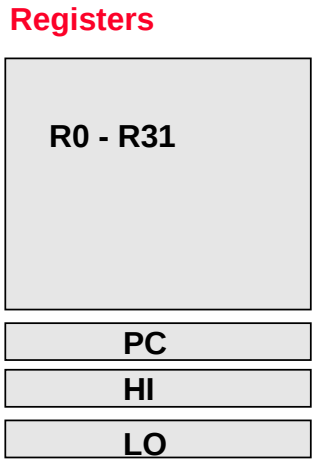
MIPS

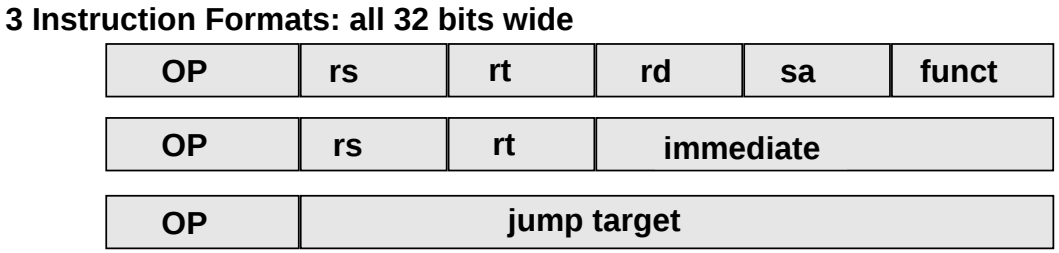
- Abbreviation for Microprocessor without Interlocked Pipeline Stages
- Approx. 100 million sold in 2002
  - Playstation I, Nintendo 64, Cisco Routers
- 2010:
  - DVD recorders, wireless routers, NAS (Network-Attached Storage)
  - LED monitors
- Example of typical instruction set since the 1980s
- Other instruction sets similar
  - design based on hardware
  - few basic operations that need to be supported

Instruction Classes of MIPS

- Minimal number of instructions required
- Information flow
  - Load from memory
  - Store in memory
- Logic operations
  - Logic and/or
  - Negation
- Shift
- Arithmetic operations
  - Addition, Subtraction, ...
- Branch operation

MIPS R3000 Instruction Set Architecture





Instruction Categories: Load/Store, Computational, Jump and Branch, Floating Point (coprocessor), Memory Management, Special

Information Transfer

- Storage locations for information
  - Registers
    - 32 Registers, \$0 ... \$31
    - Register \$0 is always 0
    - Directly inside the CPU
    - Required for arithmetic and logic operations
    - Access time $\approx$ Clock frequency of processor
  - Memory
    - 2^{32} memory cells are addressable
    - Access requires more time, in the area of four times the processor clock frequency
- Before an operation can start, the operands have to be moved into the register set

GNU MIPS Register Allocation

MIPS register allocation		
Name	**Register #**	**Comments**
`$zero`	0	constant 0
`$at`	1	assembler
`$v0 ... $v1`	2 – 3	result value register
`$a0 ... $a3`	4 – 7	arguments
`$t0 ... $t7`	8 – 15	temporary variables
`$s0 ... $s7`	16 – 23	saved
`$t8 ... $t9`	24 – 25	temporary variables
`$k0 ... $k1`	26 – 27	operating system
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

MIPS Arithmetic (1)

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:
C code: `a = b + c`
MIPS 'code': `add a, b, c`
(we will talk about registers later)

"The natural number of operands for an operation like addition is three . . . requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

MIPS Arithmetic (2)

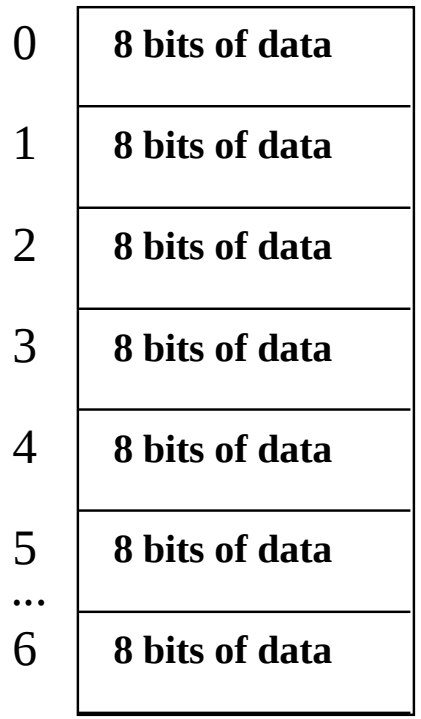
- Design principle 1: simplicity favors regularity
- Of course this complicates some things ...
  C code: `a = b + c + d;`
  MIPS code: `add a, b, c`
  `add a, a, d`
- Operands must be registers, only 32 registers provided
- Each register contains 32 bits
- Design principle 2: smaller is faster Why?
- Design principle 3: Good design demands compromise – make all the instructions the same length
- Design principle 4: Make the common case fast – making constants part of arithmetic instructions

Registers vs. Memory

- Arithmetic instructions operands must be registers
  - only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?

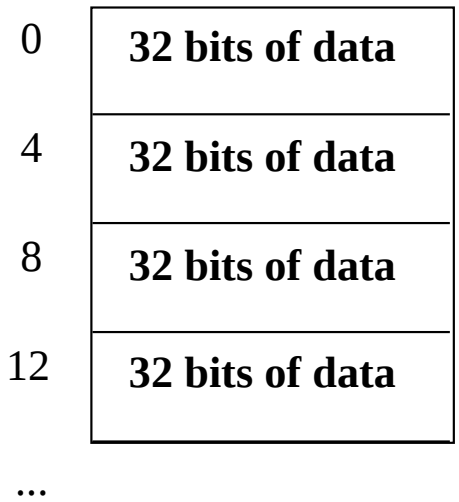
Memory Organization (1)

- Viewed as a large, single-dimension array, with an address
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory



Memory Organization (2)

- Bytes are nice, but most data items use larger "words"
- Word has different definitions
  - some define it to be of fixed size (e.g., 16 bits, 32 bits, . . .)
- A word is the amount of data that a machine can process at one time
- The size of the processor registers is equal to its word size
- For MIPS, a word is 32 bits or 4 bytes



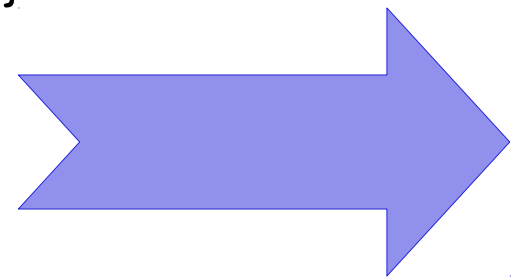
Instructions

- Load and store instructions
- Example:
  Address of `A` is `$s3`, `h` ist stored at `$s2`
  C code: `A[12] = h + A[8];`
  MIPS code:
  ```
  lw $t0, 32($s3)
  add $t0, $s2, $t0
  sw $t0, 48($s3)
  ```
- Can refer to registers by name (e.g., `$s2`, `$t2`) instead of number
- Store word has destination last
- Remember arithmetic operands are registers, not memory locations
  Cannot write: `add 48($s3), $s2, 32($s3)`

The First Example

Can we figure out the code?

```
swap(int v[], int k) {
   int temp;
   temp = v[k];
   v[k] = v[k + 1];
   v[k + 1] = temp;
}
```



```
$2 carries index

swap:
   ...
   lw    $15, 0($2)
   lw    $16, 4($2)
   sw    $16, 0($2)
   sw    $15, 4($2)
   jr    $31
```

So Far We Have Learned

- MIPS
  - loads words but addresses bytes
  - arithmetic on registers only
- Instruction Meaning

```
add $s1, $s2, $s3       $s1 = $s2 + $s3
sub $s1, $s2, $s3       $s1 = $s2 - $s3
# Version 1: assumes Memory is an array of bytes
lw $s1, 100($s2)        $s1 = Memory[$s2+100]
sw $s1, 100($s2)        Memory[$s2+100] = $s1
# Version 2: assumes Memory is an array of words
lw $s1, 400($s2)        $s1 = Memory[$s2+100]
sw $s1, 400($s2)        Memory[$s2+100] = $s1
```

GNU MIPS Register Allocation

MIPS register allocation		
Name	**Register #**	**Comments**
`$zero`	0	constant 0
`$at`	1	assembler
`$v0 ... $v1`	2 – 3	result value register
`$a0 ... $a3`	4 – 7	arguments
`$t0 ... $t7`	8 – 15	temporary variables
`$s0 ... $s7`	16 – 23	saved
`$t8 ... $t9`	24 – 25	temporary variables
`$k0 ... $k1`	26 – 27	operating system
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Example

- Add the four variables `b`, `c`, `d`, and `e` and place the result in variable `a`

  ```
  a = b + c + d + e
  ```

- Add operation, first parameter is the destination, second and third are the source
- Sequence:
  - `add a, b, c` `#a = b + c`
  - `add a, a, d` `#a = b + c + d`
  - `add a, a, e` `#a = b + c + d + e`

MIPS assembly language				
Category	**Instruction**	**Example**	**Meaning**	**Comments**
Arithmetic	add	add $s1, $s2, $s3	`$s1 = $s2 + $s3`	Three operands, data in register
	subtract	sub $s1, $s2, $s3	`$s1 = $s2 - $s3`	Three operands, data in register

Example (version A)

- Complex statement

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

- Version A

  ```
  add t0, g, h
  add t1, i, j
  sub f, t0, t1
  ```

  2 intermediate variables required: `t0`, `t1`

- Compiler assigns registers (MIPS has 32) `f`, `g`, `h`, `i`, `j` are assigned to `$s0`, `$s1`, `$s2`, `$s3`, `$s4`

  ```
  add $t0, $s1, $s2
  add $t1, $s3, $s4
  sub $s0, $t0, $t1
  ```

Example (version B)

- Version B

```
add f, g, h
sub f, f, i
sub f, f, j
```

- MIPS code

```
add $s0, $s1, $s2
sub $s0, $s0, $s3
sub $s0, $s0, $s4
```

no intermediate variables required

Information Transfer

- Storage locations for information
  - Registers
    - 32 Registers, \$0 ... \$31
    - Register \$0 is always 0
    - Directly inside the CPU
    - Required for arithmetic and logic operations
    - Access time $\approx$ Clock frequency of processor
  - Memory
    - 2^{32} memory cells are addressable
    - Access requires more time, in the area of four times the processor clock frequency
- Before an operation can start, the operands have to be moved into the register set

Memory Organization

- Viewed as a large, single-dimension array, with an address
- A memory address is an index into the array
- Sequentially ordered memory addresses
- Byte addressing means that the index points to a byte of memory
- MIPS operation

  Load

  ```
  lw reg mem
  ```

  Save

  ```
  sw reg mem
  ```

Memory

- Example assumption
  - `A` is an array of 100 words
  - `g` and `h` are in `$s1` and `$s2`
  - Base address (i.e., starting address) of `A` is in `$s3`
- Function `g = h + A [8]`
- Code for the core operation

```
lw $t0, 32($s3)          # temp. reg. $t0 gets A[8]
                         # index * 4 to get word addr
add $s1, $s2, $t0        # g = h + A[8]
```

Example (1)

- Assumption
  - `A` is the base address of an array of 100 elements
  - `A` is stored in the base register: `$s3`
  - `i` is the offset between the elements
  - `g`, `h`, `i` are associated to `$s1`, `$s2`, `$t4`
- Function `g = h + A[i]`
- Steps
  - Calculate the offset defined by `i`
  - Add to the base address `A`
  - Perform the arithmetic function

Example (2)

- Calculating the address
- Derive correct offset: `4 * i`
- Implemented by double add

```
add $t1, $t4, $t4    # t1 = 2 * t4
add $t1, $t1, $t1    # t1 = 2 * t1
add $t1, $t1, $s3
    # $t1 = address of A[i] (4 * i + $s3)
```

- Code for the core operation

```
lw $t0, 0($t1)
    # temporary register $t0 = A[i]
add $s1, $s2, $t0
    # g = h + A[i]
```

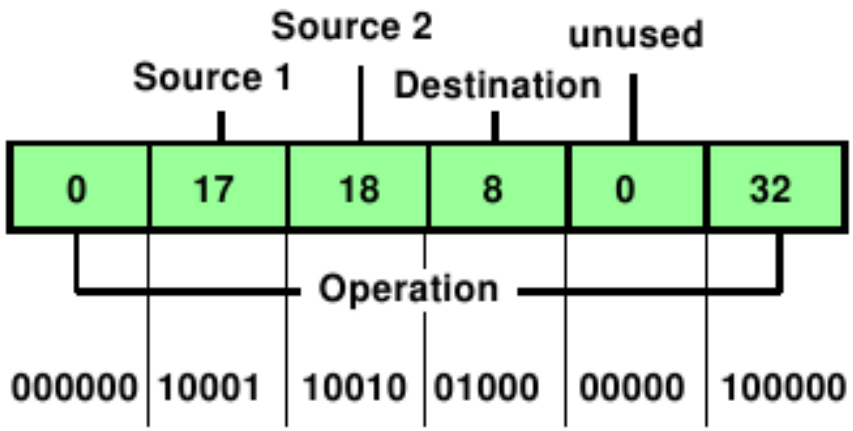
Machine Oriented Presentation

- Assembler code: mnemonic notation

  `add $t0, $s1, $s2`
- Mapping of instruction to the machine level
- Binary representation
  - Sequence of 32 '1's and '0's
  - `0000 0010 0011 0010 0100 0000 0010 0000`
  - `000000 10001 10010 01000 00000 100000`
- Well structured for minimal effort when finally mapped to hardware

Arithmetic Operations

- Elements of an addition
  - Source register 1
  - Source register 2
  - Target register
  - Operation type
- Example: `add $t0, $s1, $s2`



Logical Operations

- Shifts
  - Move bits to left or right, filling emptied with 0
- `0000 0000 0000 0000 0000 0000 0000 1001 = 9`
- Shift left by 4
- `0000 0000 0000 0000 0000 0000 1001 0000 = 144`
- Shift left logical

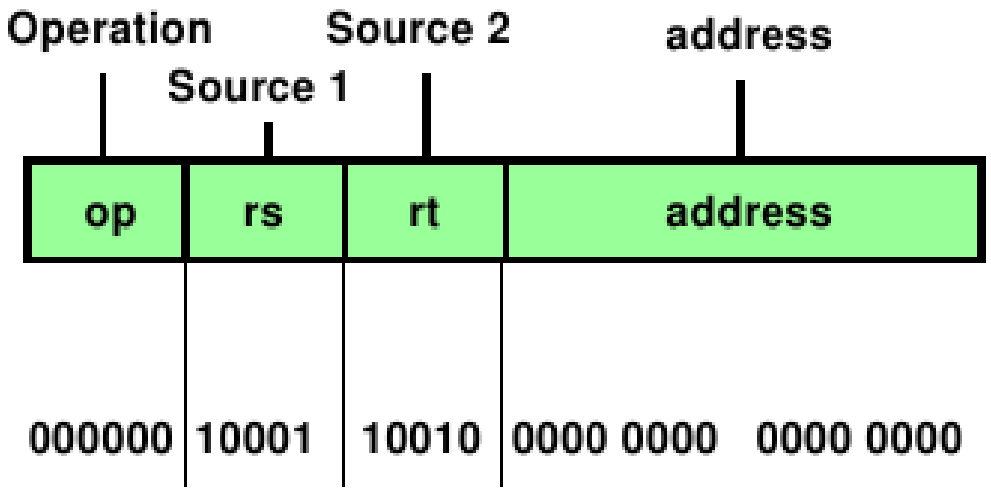
```
sll $t2, $s0, 4    # $t2 = $s0 << 4 bits
```

6 Fields Form the Instruction

- op (6 bits)
  operation of the instruction
- rs (5 bits)
  first register source operand
- rt (5 bits)
  second register source operand
- rd (5 bits)
  register destination operand
- shamt (5 bits)
  shift amount
- funct (6 bits)
  function, selects the variant of operation

Extended Format

- 1 word fine for instructions on registers (R-type for registers)
- Different format required whenever an address is used (I-type for immediate operations)



Instruction Formats

Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32	n.a.
`sub`	R	0	reg	reg	reg	0	34	n.a.
`lw`	I	35	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
`sw`	I	43	reg	reg	*n.a.*	*n.a.*	*n.a.*	address

Example (1)

- What is the binary code for the instruction `add $t0, $s2, $t0` ?
  - op (6 bits, operation) $= 0$ 000000
  - rs (5 bits, first source reg.) $= 18$ 10010
  - rt (5 bits, second source reg.) $= 8$ 01000
  - rd (5 bits, destination reg.) $= 8$ 01000
  - shamt (5 bits, shift amount) $= 0$ 00000
  - funct (6 bits, function) $= 32$ 100000
- Keep in mind the sequence of operands:
- Assembler: target source source
- Binary: source source target

Example (2)

- C code `A[300] = h + A[300]`
- Assembler code (`$t1` is the base of the array `A`, `h` is in `$s2`)

```
lw $t0, 1200($t1)
    # temporary reg. $t0 get A[300]
add $t0, $s2, $t0
    # add h to t0
sw $t0, 1200($t1)
    # store result ($t0) back to A[300]
```

- Binary code (decimal notation)

op	rs	rt	rd	shamt	adr / funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Immediate Operands (1)

- Constant/Immediate operands
- Up to now:
  `add register1, register2, register3`
- Wasting of a register for a constant value e.g., `$t0` for the increment of 4
- Loading a constant from memory
- Solution: immediate operands, e.g.,
  `addi register1, register2, immediate`

Immediate Operands (2)

- Instructions with immediate operands
  - Add immediate: `addi`
  - Set less than immediate: `slti`
  - Or immediate: `ori`
  - And immediate: `andi`
- What happens if you need to load a value larger than 16 bits?

Immediate Operands (3)

- How to load at 32-bit constant $(4,000,000)$

```
load 0000 0000 0011 1101 0000 1001 0000 0000
```

- Load upper 16 bits, which is 61 decimal using `lui`

```
lui $s0, 61                 # 61 = 0000 0000 0011 1101
```

- `$s0` has now following value

```
0000 0000 0011 1101 0000 0000 0000 0000
```

- Add lower 16 bits

```
ori $s0, $s0, 2304      # 2304 = 0000 1001 0000 0000
```

- Final value of `$s0`

```
0000 0000 0011 1101 0000 1001 0000 0000
```

1. Load upper half first: `lui`, resets lower 16 bits
2. `ori`, loads 0s into upper bits of constant

Swap Example

- Swapping words

```
1 swap(int v[], int k) {
2   int temp;
3   temp = v[k];
4   v[k] = v[k+1];
5   v[k+1] = temp;
6 }
```

```
1 ...
2 lw    $15, 0($2)
3 lw    $16, 4($2)
4 sw    $16, 0($2)
5 sw    $15, 4($2)
6 jr    $31
7 ...
```

- $2 is base address + index
- Offset is being used

Summary

- Operands
  - Registers (32 bits)
  - Memory words (30 bits)
- Assembly language – using mnemonics
  - Add
  - Subtract
  - Load
  - Store
- Machine language – binary numbers
  - Instruction format
  - Corresponding fields

Overview

Example	Comments
`$s0, $s1, ..., $s7` `$t0, $t1, ..., $t7`	Fast location for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16-23 and $t0-$t7 map to 8-15.
Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte-addresses, so sequential words differ by 4. Memory holds data structures such as arrays and spilled registers.

Instruction	Example	Meaning	Comments
add	add $s1, $s2, $s3	`$s1 = $s2 + $s3`	Three operands, data in register
subtract	sub $s1, $s2, $s3	`$s1 = $s2 - $s3`	Three operands, data in register
add immediate	addi $s1, $s2, 100	`$s1 = $s2 + 100`	Used to add constants
load word	lw $s1, 100($s2)	`$s1 = Memory($s2 + 100)`	Data from memory to register
store word	sw $s1, 100($s2)	`Memory($s2 + 100) = $s1`	Data from register to memory

Format	Example						Comments
R	0	18	19	17	0	32	`add $s1, $s2, $s3`
R	0	18	19	17	0	34	`sub $s1, $s2, $s3`
I	8	18	17	100			`addi $s1, $s2, 100`
I	35	18	17	100			`lw $s1, 100($s2)`
I	43	18	17	100			`sw $s1, 100($s2)`
	6 bits	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**	**All MIPS instructions 32 bits**
R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I	op	rs	rt	constant or 16 – bit address			Data transfer format