CS3350B Computer Organization
Chapter 3: CPU Control & Datapath
Part 1: Introduction to MIPS

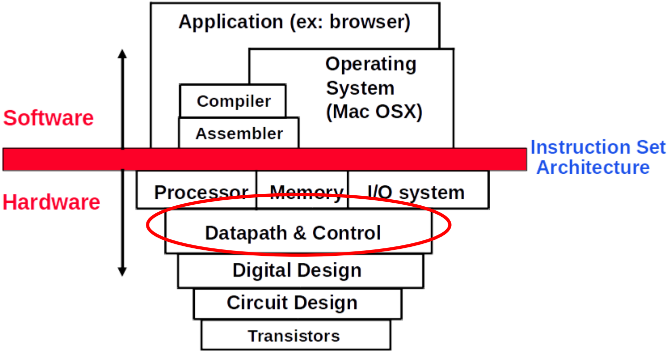
Iqra Batool

Department of Computer Science
University of Western Ontario, Canada

Monday February 12, 2024

Outline

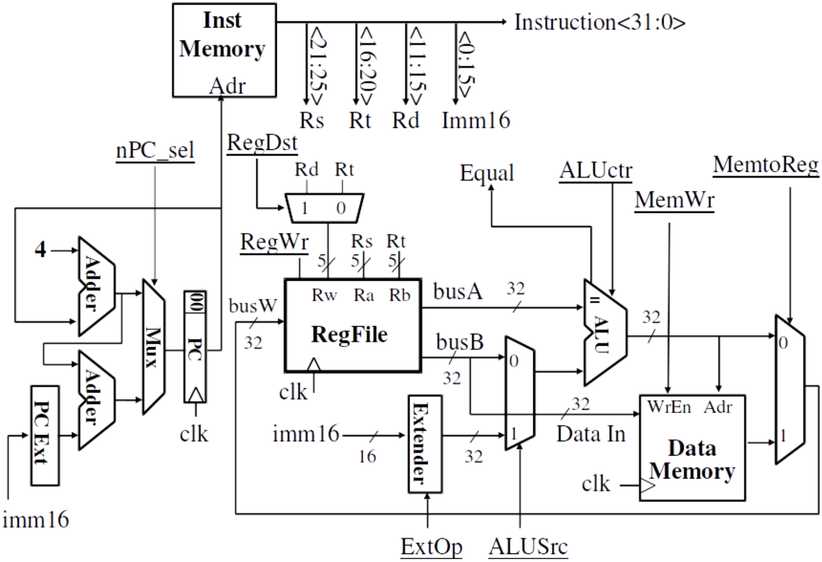
Layers of Abstraction



We will put together the combinational circuits and state circuits to see how a CPU actually works.

- How does data flow through the CPU?
- How are the path and circuits (MUX, ALU, registers) controlled?

Preview: MIPS Datapath



MIPS ISA

MIPS ISA: Microprocessor without Interlocked Pipelined Stages.

↳ ISA: The language of the computer.

↳ Microprocessor: a CPU.

↳ **Interlocking**: To come in chapter 4.

↳ **Pipelined Stages:** The data path is broken into a ubiquitous 5-stage pipeline (also chapter 4).

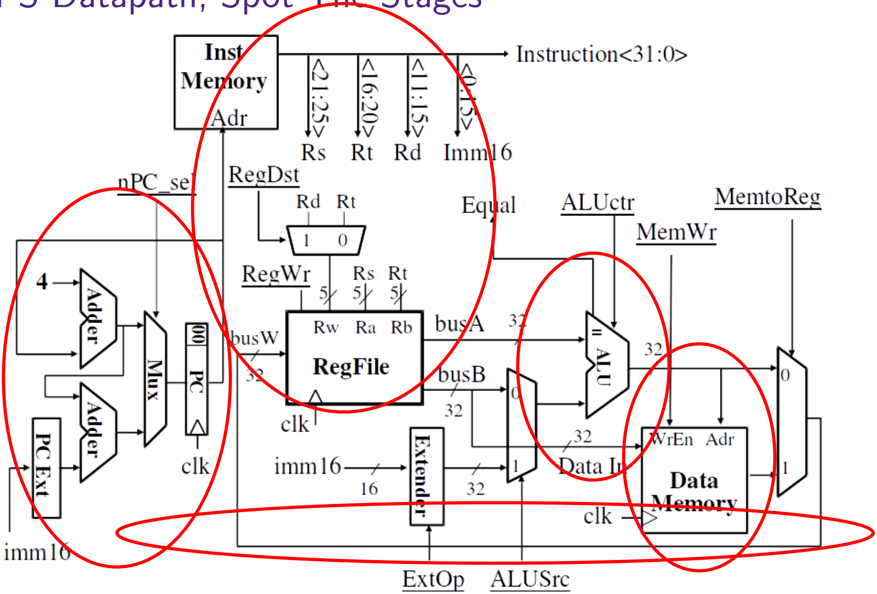
MIPS is a RISC ISA.

- **RISC:** Reduced Instruction Set Computer.
  - ↳ Provided instructions are simple, datapath is simple.
- Contrast with **CISC:** Complex Instruction Set Computer.
  - ↳ Instructions can be very "meta", each performing many lower-level instructions.

The 5-Stages of the Datapath

1. **IF**: Instruction Fetch
2. **ID**: Instruction Decode
3. **EX/ALU**: Execute/Arithmetic
4. **MEM**: Access Memory
5. **WB**: Write-back result

MIPS Datapath, Spot The Stages



Coupling of ISA and Datapath

A datapath must be built to satisfy the requirements of the ISA.

- Instructions in the ISA determine what is needed internally.
- Circuitry limit possible instructions.
- Built from combinational blocks composed together.
- Very hard to decouple the two.

We begin by looking at the MIPS ISA before looking at the datapath components.

↳ We need a common language to discuss the datapath and give concrete examples.

Layers of an ISA

Start at high-level and work down.

- MIPS assembly
- MIPS instruction formats
- MIPS instruction binary
  - ↳ *Everything* on a computer is a number
  - ↳ Recall instruction memory cache (banked L1 cache)

MIPS assembly is a type of **RTL:** Register transfer language.

- Everything in MIPS is specified by registers and movement between them or between registers and memory.
- Most often, we abstract away the concept of caches here and assume CPU talks directly with memory.
  - ↳ In reality, the circuity of the cache automatically abstracts away the memory hierarchy and handles cache hits/misses.

Outline

MIPS Assembly: The Basics

Registers:

- 32 general purpose 32-bit integer registers, denoted \$0–\$31.
- \$0 always holds the value 0.
- \$31 is reserved as the link register: stores the the point in instruction memory to return to after a function call.
- \$PC holds the **program counter** – address of current instruction
- \$HI & \$LO store results of multiplication/division

Memory:

- 32-bit words and 32-bit memory addresses.
- Byte-addressable memory.
- Indexed like a big array of bytes: Mem[0], Mem[1024], Mem[32768].

MIPS Assembly: RTL Examples

3-Operand Arithmetic:

- add \$8, \$9, \$10 ≡ $R[8] \leftarrow R[9] + R[10];$
- sub \$8, \$9, \$10 ≡ $R[8] \leftarrow R[9] - R[10];$

2-Operand Arithmetic (Immediate Arithmetic):

- addi \$8, \$9, 127 ≡ $R[8] \leftarrow R[9] + 127;$
- addi \$8, \$9, 913 ≡ $R[8] \leftarrow R[9] + 913;$
- addi \$8, \$9, -6 ≡ $R[8] \leftarrow R[9] - 6;$

Data Transfer (Memory Accesses):

- lw \$13, 32(\$10) ≡ $R[13] \leftarrow Mem[R[10] + 32];$
- sw \$13, 8(\$10) ≡ $Mem[R[10] + 8] \leftarrow R[13];$

MIPS Assembly: 3 Operand Arithmetic

$$\texttt{op \$rd, \$rs, \$rt} \quad \equiv \quad \$rd = \$rs \text{ op } \$rt$$

- \$rd is the destination register.
- \$rs is the (first) source register.
- \$rt is the second source register.
- op is some arithmetic operation:
  - ↳ `add`, `addu`, `sub`, `subu`, `and`, `or`, `xor`, ...

These instructions *assume* an interpretation of the bits stored in the register.

- Programmer/compiler must choose proper instruction for data.
- add vs addu

MIPS Assembly: 2 Operand Arithmetic

`op $rt, $rs, imm` $\equiv$ \$rt = \$rs op imma

- \$rt is the destination register.
- \$rs is the source register.
- `imm` is an **immediate**—a number whose value is hard-coded into the instruction.
  - ↳ C Example: int i = j + 12;
- op is some arithmetic operations:
  - ↳ `addi`, `addiu`, `subiu`, `andi`, `ori`, `xori`, ...
  - ↳ `sll`, `srl` (logical); `sla`, `sra` (arithmetic) (shifts are really a special case of R-type instr.)

These instructions *assume* an interpretation of the bits stored in the register.

- Programmer/compiler must choose proper instruction for data.
- Signed vs unsigned arithmetic. Logical vs arithmetic shifts.

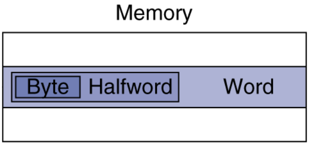
MIPS Assembly: Data Transfer

$$\texttt{lw \$rt, offset(\$rs)} \equiv \text{\$rt} = \text{Mem[\$rs + offset]}$$
$$\texttt{sw \$rt, offset(\$rs)} \equiv \text{Mem[\$rs + offset]} = \text{\$rt}$$

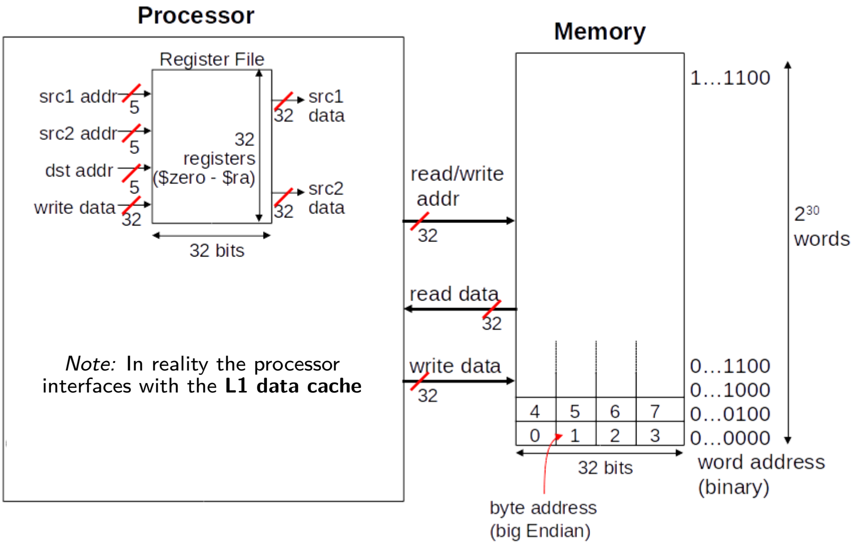
- $rt is the “value” register.
- $rs is the “address” register.
- `offset` is an **immediate**.

It is also possible to load and store bytes, **halfwords**:

- lb, sb (byte);
- lwr, swr (least-signficiant halfword);
- lwl, swl (most-significant halfword).



MIPS: A View of Memory



Note: In reality the processor interfaces with the **L1 data cache**

Aside: Endianness Defined

Endianness: The ordering of multiple bytes which are intended to be interpreted together as a single number.

- Important in memory layout, digital signals, networks, etc.

Consider the number: `0xAABBCCDD`

Little-Endian: The least-significant byte is stored/sent first.

- Ordering: `0xDD`, `0xCC`, `0xBB`, `0xAA`

Big-Endian: The most-significant byte is stored/sent first.

- Ordering: `0xAA`, `0xBB`, `0xCC`, `0xDD`
- MIPS is big-endian.

Big-endian conceptually easier but little-endian has performance benefits. In reality, hardware handles all conversions to and from, so we *rarely* care.

Outline

1 Overview

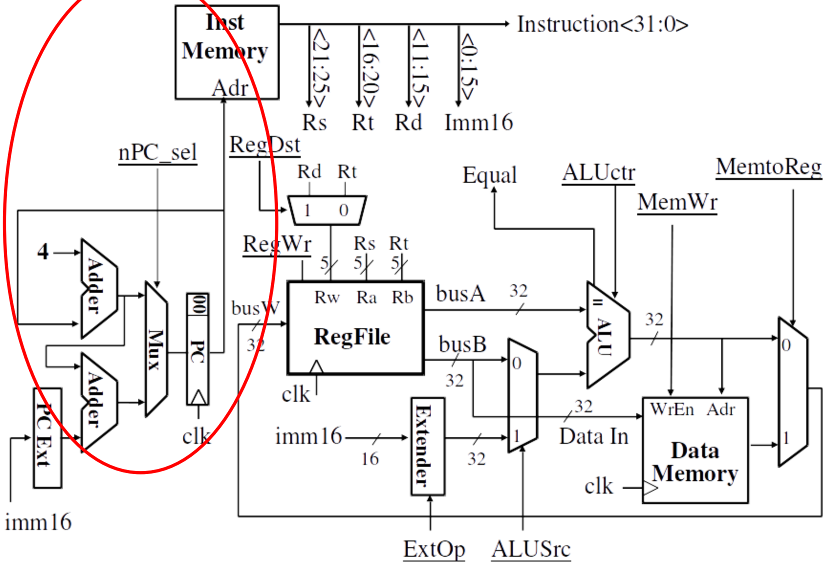
2 MIPS Assemebly

3 Instruction Fetch & Instruction Decode

4 MIPS Instruction Formats

5 Aside: Program Memory Space

MIPS Datapath, Instruction Fetch



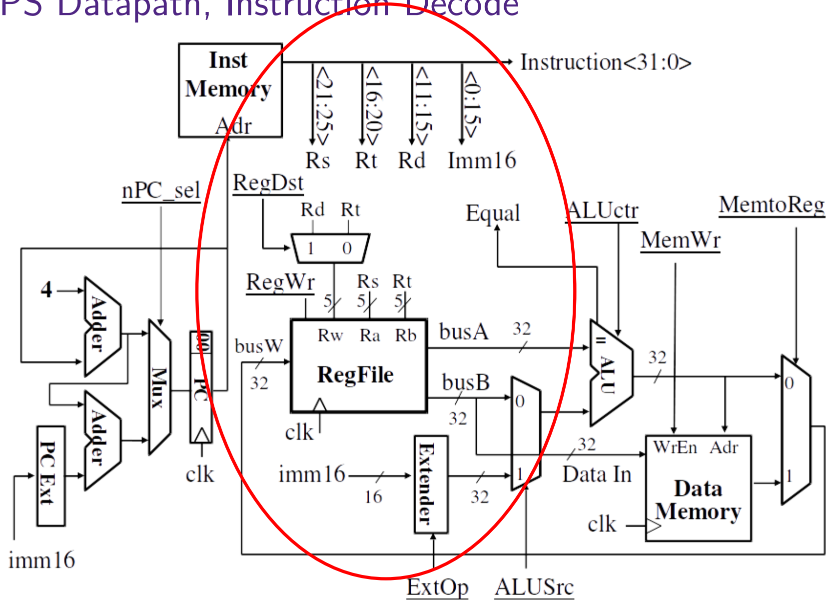
Instruction Fetch

IF — Instruction Fetch

- The simplest part of the datapath.
- One job: fetch the next instruction to execute from memory.
- *Banked L1 cache* $\Longrightarrow$ separate cache just for instructions

1. On the clock's rising edge, update the value of **PC**—program counter.
   ↳ Essentially the index into instruction memory.
2. *Fetch* the instruction from memory and pass it to next stage: instruction decode.
3. Prepare for next instruction: calculate PC + 4 (4 bytes since instructions are word-aligned).

MIPS Datapath, Instruction Decode



Instruction Decode

ID — Instruction Decode

- Break the *binary value* of an instruction into its parts and decide what to do.
  - ↳ Recall: instructions eventually get compiled down to *bytecode* (i.e. binary).
- Get the values ready for arithmetic: registers, immediates.

1. Break the instruction into individual bit segments.
2. Access operand values from registers.
3. Extend immediate to 32 bits (if using).

But how to break the instruction?

Aside: MIPS Special Register Names

- \$zero: the zero-valued register (\$0)
- \$at: reserved for compiler (\$1)
- \$v0, \$v1: result values (\$2, \$3)
- \$a0 - \$a3: arguments (\$4–\$7)
- \$t0 - \$t9: temporaries (\$8–\$15, \$24, \$25)
  - ↳ Can be overwritten by callee
- \$s0 - \$s7: saved (\$16–\$23)
  - ↳ Must be saved/restored by callee
- \$gp: global pointer for static data (\$28)
- \$sp: stack pointer (\$29)
- \$fp: frame pointer (\$30)
- \$ra: return address (\$31)

Outline

MIPS Instruction Formats

Every instruction in MIPS is 32-bits.

- A memory word is 32 bits, after all.

All instructions belong to 3 pre-defined formats:

- **R-Type**: “Register”
- **I-Type**: “Immediate”
- **J-Type**: “Jump”

Each format defines how those 32 bits of instruction data are broken up into individual “bit-fields” and how they are interpreted during ID stage.

- The first 6 bits always encode the **opcode**.
- The opcode determines the type of instruction and format of the remaining bits.

R-Type Instructions

R-Type instructions usually have 3 registers as its operands.

↳ "Register type".

↳ General arithmetic operations.

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

- op — the opcode.
- rs — first source register.
- rt — second source register.
- rd — destination register.
- shamt — shift amount; used for shift instructions, 0 otherwise.
- funct — the arithmetic *function* the ALU should perform.

R-Type Examples 1

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

`add $t0, $s1, $s2`

op	$s1	$s2	$t0	shamt	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

`sub $t0, $s1, $s2`

op	$s1	$s2	$t0	shamt	sub
0	17	18	8	0	34
000000	10001	10010	01000	00000	100010

- For R-Type instructions the opcode and funct *together* determine the operations to perform.

R-Type Examples 2

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

`sll $s0, $t0, 4`

op	rs	\$t0	\$s0	4	shift left
0	0	8	16	4	0
000000	00000	01000	10000	00100	000000

Note: shift instruction have two registers and an immediate, but are *not* immediate instructions.

- Here, the allowed value of the shift amount is only 5 bits, not 16 bits as in an immediate-type instruction.

R-Type Examples 3: Bit-Wise Logical Operations

- Useful to mask (remove) bits in a word.
  - ↳ Select some bits, clear others to 0.

```
and $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

- Useful to include bits in a word.
  - ↳ Set some bits to 1, leave others unchanged.

```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

I-Type Instructions

I-Type instructions always have 2 registers and an **immediate**.

↳ "Immediate type".

↳ Immediate arithmetic, data transfer, branch.

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- op — the opcode.
- rs — first source register.
- rt — second source (or destination) register.
- imm — the immediate/constant.

I-Type Examples 1

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

`addi $t1, $t0, 10`

op	rs	rt	immediate
8	$t0	$t1	10
001000	01000	01001	0000000000001010

`addiu $t1, $t0, 10`

op	rs	rt	immediate
9	$t0	$t1	10
001001	01000	01001	0000000000001010

Note: unsigned instructions will *not* signal exception on overflow.

I-Type Examples 2

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

`lw $t1, 12($t0)`

op	rs	rt	immediate
35	\$t0	\$t1	12
100011	01000	01001	0000000000001100

`sw $t1, 32($t0)`

op	rs	rt	immediate
43	\$t0	\$t1	32
101011	01000	01001	0000000000100000

I-Type Examples 3

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

bne $t0, $t1, 24

```
if ($t0 != $t1)
    PC = PC + 4 + (24 << 2);
else
    PC = PC + 4;
```

op	rs	rt	immediate
5	$t0	$t1	24
000101	01000	01001	0000000000011000

J-Type Instructions

J-Type instructions have just one big immediate, called a **target**.

↳ "Jump type".

↳ Only two instructions: `j` (jump) and `jal` (jump and link).

op	target (jump address)
6 bits	26 bits

- op — the opcode.
- target — the target memory address to jump to.

Note: target is always multiplied by 4 before being applied to program counter...

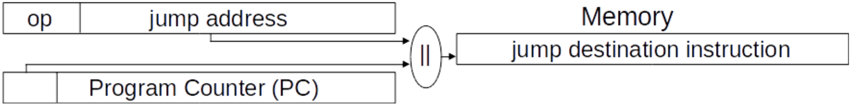
J-Type Instructions and Pseudo-Direct Addressing

Pseudo-Direct Addressing: Almost a direct addressing of instruction memory.

- Compiler usually handles the calculation of the exact jump target.

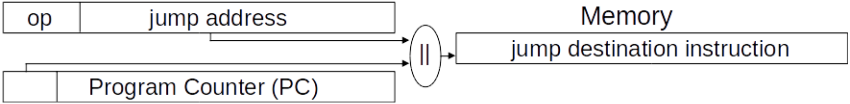
Next value of PC is target $\times$ 4 combined with upper 4 bits of current PC.

```
nPC = (PC & 0xf0000000) | (target << 2);
```

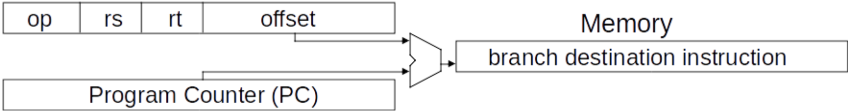


Addressing Instruction Memory in MIPS

- **Pseudo-Direct:** J-Type instructions



- **PC-Relative:** Branch instructions

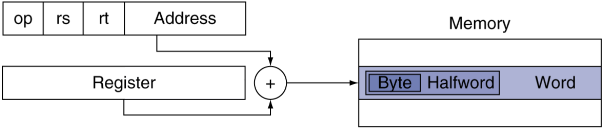


Addressing Operands in MIPS

- **Immediate Addressing**, I-Type instruction.

- **Register Addressing**, Almost all instructions.

- **Base Addressing**, Data transfer instructions.



MIPS ISA: Some Important Instructions

Category	Instruction		OP/ funct	Example	Meaning
Logic & Arith.	add	R	0/32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	R	0/34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	I	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	and/or	R	0/(36/37)	(and/or) $s1, $s2, $s3	$s1 = $s2 ($\wedge$/$\vee$) $s3
	(and/or) immediate	I	12/13	(andi/ori) $s1, $s2, 6	$s1 = $s2 ($\wedge$/$\vee$) 6
	shift right logical	R	0/2	srl $rd, $rt, 4	$rd = $rt >> 4
	shift right arithmetic	R	0/3	sra $rd, $rt, 4	$rd = $rt >> 4
Data Transfer	load word	I	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	I	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	I	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	I	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
Cond. Branch	br on equal	I	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	I	5	bne $s1, $s2, L	if ($s1 != $s2) go to L
	set less than	R	0/42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set less than immediate	I	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump	jump	J	2	j 250	go to 1000
	jump register	R	0/8	jr $t1	go to $t1
	jump and link	J	3	jal 250	go to 1000; $ra=PC+4

Note: knowing the binary values of each bit-field is not neccesary, but understanding the semantic meaning of each instruction *is* important.

Full Method Example: C to MIPS

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

```
swap:  sll $t1, $a1, 2    # $t1 = k * 4
       add $t1, $a0, $t1  # $t1 = v+(k*4)
                          # (address of v[k])
       lw $t0, 0($t1)     # $t0 (temp) = v[k]
       lw $t2, 4($t1)     # $t2 = v[k+1]
       sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
       sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
       jr $ra             # return to calling routine
```

Note: words and `int`-type are both 32-bits here.

Outline

Revisiting Program Basics

- **Frame:** The encapsulation of one method call; arguments, local variables.
  - ↳ "Enclosing subroutine context".
- (Call) **Stack** (of frames): The stack of method invocations.
  - ↳ Base of stack is the main method, each method call adds a frame to the stack.
- **Heap**: globally allocated data that lives beyond the scope of the frame in which it was allocated.
- **Static Data:** Global data which is stored in a *static* memory address throughout life of program.

MIPS Special Registers

- \$v0, \$v1: result values (\$2, \$3)
- \$a0 - \$a3: arguments (\$4–\$7)
- \$t0 - \$t9: temporaries (\$8–\$15, \$24, \$25)
  - ↳ Can be overwritten by callee
- \$s0 - \$s7: saved (\$16–\$23)
  - ↳ Must be saved/restored by callee
- \$gp: global pointer for static data (\$28)
- \$sp: stack pointer (\$29)
- \$fp: frame pointer (\$30)
  - ↳ The stack pointer *before* the current frame's invocation.
- \$ra: return address (\$31)

Memory Layout in MIPS (and most languages)

Text: program code

Static data: global variables

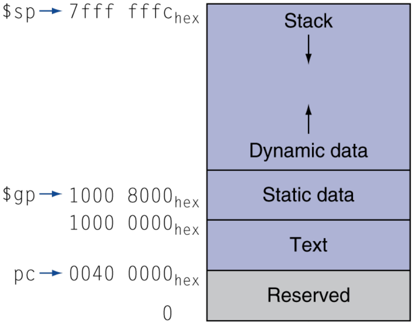
↳ static/global variables, constant arrays, etc.

↳ \$gp initialized to address allowing ±offsets into this segment

Dynamic data: heap

↳ e.g., malloc in C, new in Java

Stack: "automatic" storage



Note: In this diagram the higher memory addresses are at top.

Handling the Stack in MIPS

```
 sort:   addi $sp, $sp, -20   # make room on stack for 5 registers
         sw $ra, 16($sp)      # save $ra on stack
         sw $s3,12($sp)       # save $s3 on stack
         sw $s2, 8($sp)       # save $s2 on stack
         sw $s1, 4($sp)       # save $s1 on stack
         sw $s0, 0($sp)       # save $s0 on stack
---------------------------------------------------------------------
         ...                  # procedure body
         ...                  # call swap a bunch to do bubble sort
---------------------------------------------------------------------
exit1:   lw $s0, 0($sp)       # restore $s0 from stack
         lw $s1, 4($sp)       # restore $s1 from stack
         lw $s2, 8($sp)       # restore $s2 from stack
         lw $s3,12($sp)       # restore $s3 from stack
         lw $ra,16($sp)       # restore $ra from stack
         addi $sp, $sp, 20    # restore stack pointer
         jr $ra               # return to calling routine
```