MIPS

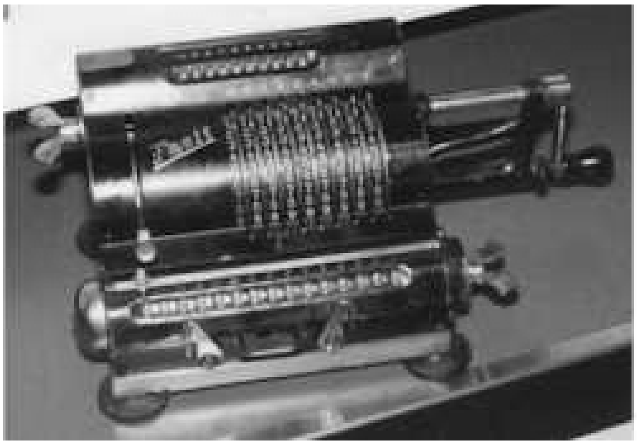
Tim Langens
Jef Neefs
Elio Struyf
Luc Verstrepen

1 Table of contents

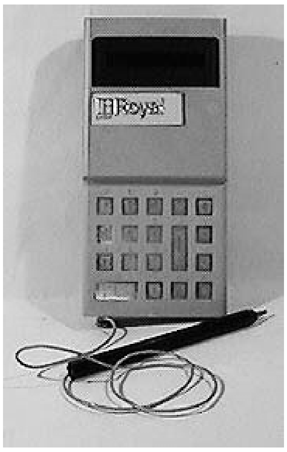
2 First things first

2.1 History

Man has always been interested by technology, he made tools to help him survive or to make life easier for him. We invented the wheel and fire, we crafted pyramids and we have been on the moon. Always looking for the best scientific answers. We needed help for our brains so we invented calculating machines to calculate immense formula's. First came the analog ones, and later after the electronic revolution, digital ones.



One of the first calculators



The first TI handheld calculator

The first microprocessor was made in the late 1960's. Actually 3 different companies came up with almost the same microprocessor design at the same moment. Intel, Texas Instruments and a military contractor making the Tomcat airplanes.

The first RISC processor (**reduced instruction set computer**) was made in the late 1970's and worked as the name says on a reduced instruction set. Which was to be quite faster than the older CISC architecture (**complex instruction set computer**). The point of RISC was using more registers and less load/store also an improvement of RISC was pipelines. Pipelines gives the processor the ability to run one instruction while it loaded the next and was storing the one before. So more work could be done. A calculation would start right after the one before ended which was off course more effective.

No Pipelines (CISC):

Load	Calculate	Store	Load	Calculate	Store

Pipelines (RISC):

Load	Calculate	Store			
	Load	Calculate	Store		
		Load	Calculate	Store	
			Load	Calculate	Store
				Load	Calculate

The MIPS (**Microprocessor without Interlocked Pipeline Stages**) sprouted, based on the RISC, from the mind of John L. Hennessy on Stanford University. MIPS focused almost entirely on the pipeline, making sure it could be run as "full" as possible. Although pipelining was already in use in other designs, several features of the MIPS chip made its pipeline far faster. The most important, and perhaps annoying, of these features was the demand that all instructions be able to complete in one cycle. This demand allowed the pipeline to be run at much higher speeds (there was no need for induced delays) and is responsible for much of the processor's speed. However, it also had the drawback of eliminating many potentially useful instructions, like a multiply or a divide.

In 1984 Hennessy was convinced of the future commercial potential of the design, and left Stanford to form MIPS Computer Systems where he released his first design, the R2000. In 1988 MIPS Computer Systems came with the improved version R3000. Both the R2000 and the R3000 were 32-bit design. Later came the R4000 a 64-bit CPU. But MIPS Computer Systems became financially unstable as a cause to extensive research for their biggest client SGI (Silicon Graphics). SGI didn’t want that research to fall in any other hands than theirs. So they bought MIPS Computer Systems and renamed it to MIPS Technologies. The company still runs under that name.

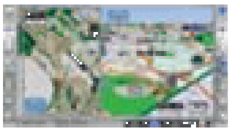
2.2 The current use of MIPS

Many devices nowadays are MIPS based. Underneath is a summation with a known model. You’ll be surprised when you’ll see what devices are MIPS based.

DVD players

Pioneer
DVR-57-H

Kenwood
HDV-810 Car Navigation System



Networking

3COM
3102 Business IP Phone

3COM
3106 Cordless Phone

Apple
Airport Extreme WLAN Access Points

Cisco Systems
7200 Series Router

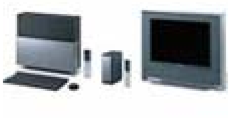
Portable Audio

Macsense
HomePod Wireless Audio Player

Residential and Small Office

Samsung
Digital Photo Frame

Sony
Media Server Vaio VGX-X90P



Pioneer
PureVision™ Plasma Television 43"
PureVision™ Plasma Television 50"

Sony
KDP-51WS550 High Definition TV
KDP-57WS550 High Definition TV
KDP-65WS550 High Definition TV

Hewlet Packard
Color Laser Jet 2500 Laser Printer

Portable Devices

Canon
EOS 10D Digital

JVC
GR-HD1

Video Game Consoles

These will be commentated a little more

2.2.1 Sony Playstation PSX

CPU
Type:LSI/MIPS R3000A
Architecture:32 Bit
Clockspeed:33,8 MHz

When this gaming console hit the markets in 1994, although Sony was already an important player, nobody could ever expect what a great impact it would have on the gaming console market.
The Sony PSX, new to the market and somewhat better performing as the also new Sega Saturn, had no problem competing with the more established Sega Brand.

Sony did this by a very good marketing campaign and a grand offer of different games. The Nintendo N64, which came out 2 years later, could not bring a change to Sony’s market position. The Playstation PSX became next to the super Nintendo one of the most sold game consoles. The PSX’ successor could easily follow in its footsteps.

2.2.2 Sony Playstation Portable

CPU
Type:MIPS R4000 32bit Core
Clockspeed:333 MHz

After Sony had taken over almost the whole game console market, it would also want to try competing with Nintendo’s handheld monopoly. The performance is again very high and is slightly less than the PS2. The PSP has a new card format named UMD. The PSP can play movies when they are converted to the right format. Movies also are available to buy on UMD’s. Of course the device can also play music.

The battery of the PSP is somewhat a drawback since it only lasts for 3 hours.

3 ISA

3.1 MIPS, a RISC computer

The reduced instruction set computer, or RISC, is a microprocessor CPU design philosophy that favors a simpler set of instructions that all take about the same amount of time to execute. The most common RISC microprocessors are Alpha, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture, and SPARC.

The idea was originally inspired by the discovery that many of the features that were included in traditional CPU designs to facilitate coding were being ignored by the programs that were running on them. Also these more complex features took several processor cycles to be performed. Additionally, the performance gap between the processor and main memory was increasing. This led to a number of techniques to streamline processing within the CPU, while at the same time attempting to reduce the total number of memory accesses.

We will now see how the MIPS architecture fits into the RISC design philosophy

3.1.1 MIPS memory

The purpose of memory is to store groups of bits, and deliver them (to the processor for loading into registers) upon demand. Most present-day computers store information in multiples of 8 bits, called a byte (or octet). Most also assign a numeric address to each byte. This is convenient because characters can be stored in bytes.

The MIPS has 32-bit address space, ranging from 0x00000000 to 0xFFFFFFFF. This is a large amount of memory, most computers do not have actual memory for all of this "address space." Memory can hold both program instructions and data. One function of the operating system is to assign blocks of memory for the instructions and data of each process (running program). Another thing a good operating system does is to allow many processes to run concurrently on the computer.

The MIPS processor always assigns your program to these fixed, even numbered locations, for your convenience:

0x00400000	Text segment	program instructions
0x10000000	Data segment	
0x7FFFFFFF	decreasing addresses	Stack segment

A word generally means the number of bits that can be transferred at one time on the data bus, and stored in a register. In the case of MIPS, a word is 32 bits, that is, 4 bytes.
Words are always stored in consecutive bytes, starting with an address that is divisible by 4.

How should one store a word (say, a register holding a number) in 4 bytes? There are two equally valid schemes: starting with the lowest numbered byte,

Little Endian	Store the little end of the number (least significant bits) first
Big Endian	Store the big end of the number first.

These terms come from Gulliver's Travels by Jonathan Swift, in which two parties are at war over whether hard-boiled eggs should be opened at the little end or the big end of the egg.

MIPS can be booted little-endian or big-endian. When booted little endian one result is that character data appear to be stored "backwards" within words.

3.1.2 MIPS Instruction Set

Before we can look to the instruction set of MIPS we need to have an overview of the CPU:

The MIPS CPU contains 32 general purpose registers that are numbered 0-31. Register $0 always contains the hardwired value 0.

Registers $at (1), $k0 (26), and $k1 (27) are reserved for the assembler and operating system and should not be used by user programs or compilers
Registers $a0-$a3 (4-7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers $v0 and $v1 (2,3) are used to return values from functions.
Registers $t0-$t9 (8-15, 24,25) are caller-saved registers that are used to hold temporary quantities that need not be preserved across calls.
Registers $s0-$s7 (16-23) are caller-saved registers that hold long-lived values that should be preserved across calls.
Register $gp (28) is a global pointer that points to the middle of a 64K block of memory in thee static data segment
Register $sp(29) is the stack pointer, which points to the last location on the stack.
Register $fp (30) is the frame pointer. The `jal` instruction writes register $ra (31), the return address from a procedure call.

The table bellow lists the registers and describes their intended uses.

Register name	Number	Usage
$zero	0	Constant 0
$at	1	Reserved for assembler
$v0	2	Expression evaluation and results of a function
$v1	3	Expression evaluation and results of a function
$a0	4	Argument 1
$a1	5	Argument 2
$a2	6	Argument 3
$a3	7	Argument 4
$t0	8	Temporary (not preserved across call)
$t1	9	Temporary (not preserved across call)
$t2	10	Temporary (not preserved across call)

$t3	11	Temporary (not preserved across call)
$t4	12	Temporary (not preserved across call)
$t5	13	Temporary (not preserved across call)
$t6	14	Temporary (not preserved across call)
$t7	15	Temporary (not preserved across call)
$s0	16	Saved temporary (not preserved across call)
$s1	17	Saved temporary (not preserved across call)
$s2	18	Saved temporary (not preserved across call)
$s3	19	Saved temporary (not preserved across call)
$s4	20	Saved temporary (not preserved across call)
$s5	21	Saved temporary (not preserved across call)
$s6	22	Saved temporary (not preserved across call)
$s7	23	Saved temporary (not preserved across call)
$t8	24	Temporary (not preserved across call)
$t9	25	Temporary (not preserved across call)
$k0	26	Reserved for OS kernel
$k1	27	Reserved for OS kernel
$gp	28	Pointer to global area
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address (used by function call)

MIPS processors don't implement a dedicated status / condition register reflecting ALU conditions such as overflow, carry, negative result, zero result, etc… Comparisons are implemented in software through general-purpose registers.

MIPS instruction set architectures are backwards compatible, i.e. a MIPS IV processor can still run MIPS I code.

MIPS instructions are 32 bits wide, even on the 64-bit processors. This maintains 32-bit instruction-set compatibility while providing high-throughput 64-bit operations.

The MIPS$_{16}$ architecture extension allows the processor to run under a 16-bit instruction set. The processor can change into MIPS$_{16}$ mode on-the-fly. MIPS$_{16}$ allows greater code density for occasions when code size is more important than outright performance.

The MIPS is a load-store machine. The only allowable memory access operations load a value into one of the registers or store a value contained in one of the registers into a memory location. All arithmetic operations operate on values that are contained in registers, and the results are placed in a register.

Instructions can be grouped into the following categories.

Load and Store: Load data from memory into a register, or store register contents in memory.

Examples:

```
lw  $t0, num1  #load word in num1 into $t0
sw  $t0, num2  #store word in $t0 into num2, ie. num2 := num1
li  $v0, 4     #load immediate value (constant) 4 into $v0
```

Arithmetic and Logic operations: perform the operation on data in 2 registers, store the result in a 3rd register.

Example:

```
add $t0, $t3, $t4     # $t0 := $t3 + $t4
```

Jump and branch : alter the PC to control flow of the program, producing the effects of IF statements and loops

There are also a few specialized instructions, floating point instructions and registers, for real numbers.

3.1.3 MIPS program structure

A MIPS program is a plain text file with data declarations and program code. The name of the file should have the suffix .s if used with the SPIM simulator. First is the data declaration section followed by the program code section. MIPS uses different language formats depending on the type of instruction. We will have a better look at this in MIPS instruction descriptions.

Data Declaration
Data declaration is placed in the section of the program identified with the assembler directive .data. It declares variable names used in the program or storage allocated in the main memory.

Code
Code is placed in the section of the text identified with the assembler directive .text. It contains instructions. The starting point for code is a label (e.g. execution label is main:).
The ending point of the main code should use the exit system.

Comments
Anything following # on a line is comment.

Example:

```
# This stuff would be considered a comment
```

Example
Putting it all together a template for a MIPS assembly language program should look like this:

```
# Comment giving name of program and description of function
# Template.s
# Bare-bones outline of MIPS assembly language program

.data     # variable declarations follow this line
          # ...

.text     # instructions follow this line


main:     # indicates start of code (first instruction to
          # execute)
     # ...
     # End of program, leave a blank line afterwards to
     # make SPIM happy
```

3.1.4 MIPS instruction formats

The MIPS ISA has fixed-width 32 bit instructions. Fixed-width instructions are common for RISC processors because they make it easy to fetch instructions without having to decode. These instructions must be stored at word-aligned addresses (i.e., addresses dividable by 4). The MIPS ISA instructions fall into three categories: R-type, I-type, and J-type. Not all ISA's divide their instructions this neatly. The format is simple.

R-type

R-type instructions refer to register type instructions. Of the three formats, the R-type is the most complex.

This is the format of the R-type instruction, when it is encoded in machine code.

B31-26	B25-21	B20-16	B15-11	B10-6	B5-0
opcode	register s	register t	register d	shift amount	function

The prototypical R-type instruction is:

```
add $rd, $rs, $rt
```

where $rd refers to some register d (d is shown as a variable, however, to use the instruction, you must put a number between 0 and 31, inclusive for d). $rs, $rt are also registers.

The semantics of the instruction are:

```
R[d] = R[s] + R[t]
```

where the addition is signed addition.

You will notice that the order of the registers in the instruction is the destination register ($rd), followed by the two source registers ($rs and $rt).

However, the actual binary format (shown in the table above) stores the two source registers first, then the destination register. Thus, how the assembly language programmer uses the instruction, and how the instruction is stored in binary, do not always have to match.

Let's explain each of the fields of the R-type instruction

opcode (B31-26):
Opcode is short for "operation code". The opcode is a binary encoding for the instruction. Opcodes are seen in all ISA’s. In MIPS, there is an opcode for add.
The opcode in MIPS ISA is only 6 bits. Ordinarily, this means there are only 64 possible instructions. Even for a RISC ISA, which typically has few instructions, 64 is quite small. For R-type instructions, an additional 6 bits are used (B5-0) called the function. Thus, the 6 bits of the opcode and the 6 bits of the function specify the kind of instruction for R-type instructions.

rd (B25-21): This is the destination register. The destination register is the register where the result of the operation is stored.

rs (B20-16): This is the first source register. The source register is the register that holds one of the arguments of the operation.

rt (B15-11): This is the second source register.

shift amount (B10-6): The amount of bits to shift. Used in shift instructions.

function (B5-0): An additional 6 bits used to specify the operation, in addition to the opcode.

I-type instructions
I-type is short for "immediate type". The format of an I-type instruction looks like:

B31-26	B25-21	B20-16	B15-0
opcode	register s	register t	immediate

The prototypical I-type instruction looks like:

```
add $rt, $rs, immed
```

In this case, $rt is the destination register, and $rs is the only source register. It is unusual that $rd is not used, and that $rd does not appear in bit positions B25-21 for both R-type and I-type instructions. Presumably, the designers of the MIPS ISA had their reasons for not making the destination register at a particular location for R-type and I-type.

The semantics of the addi instruction are:

$R[t] = R[s] + (IR_{15})^{16}\ IR_{15-0}$

where IR refers to the instruction register, the register where the current instruction is stored. $(IR_{15})^{16}$ means that bit B15 of the instruction register (which is the sign bit of the immediate value) is repeated 16 times. This is then followed by IR_{15-0}, which is the 16 bits of the immediate value.

Basically, the semantics says to sign-extend the immediate value to 32 bits, add it (using signed addition) to register R[s], and store the result in register $rt.

J-type instructions

J-type is short for "jump type". The format of an J-type instruction looks like:

B31-26	B25-0
opcode	target

The prototypical I-type instruction looks like:

```
j target
```

The semantics of the j instruction (j means jump) are:

$PC <- PC_{31-28}\ IR_{25-0}\ 00$

where PC is the program counter, which stores the current address of the instruction being executed. You update the PC by using the upper 4 bits of the program counter, followed by the 26 bits of the target (which is the lower 26 bits of the instruction register), followed by two 0's, which creates a 32 bit address.

Why Five Bits?

If you look at the R-type and I-type instructions, you will see 5 bits reserved for each register. You might wonder why.

MIPS supports 32 integer registers. To specify each register, the register are identified with a number from 0 to 31. It takes log2 32 = 5 bits to specify one of 32 registers.

If MIPS has 64 register, you would need 6 bits to specify the register.

The register number is specified using unsigned binary. Thus, 00000 refers to $r0 and 11111 refers to register $r31.

3.1.5 MIPS data types

MIPS supports operations on signed and unsigned data types. Remember that MIPS can be booted little endian and big endian, this means we have to be careful with converting a byte to a word. But it is possible.

Integer data types:

- Byte: 8 bits
- Half-words: 16 bits
- Words: 32 bits
- Double-words: 64 bits (not in basic MIPS I)

Floating Point:

- 32-bit single precision
- 64-bit double precision

3.1.6 MIPS R3000 instruction set

Now that we know the instruction formats, we can create detailed descriptions of the instructions for the MIPS R3000, one of the popular MIPS processors.

MIPS Operands

Name	Example	Comments
32 registers	\$0, \$1, \$2,..., \$31	Fast location for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$0 always equal 0. Register \$1 is reserved for the assembler to handle pseudo instructions and large constants
2^{30} memory words	Memory[0], Memory[4],..., Memory[4293967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

MIPS Assembler Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; exception possible
	subtract	sub \$1,\$2,\$3	\$1 = \$2 - \$3	3 operands; exception possible
	add immediate	addi \$1,\$2,100	\$1 = \$2 + 100	+ constant; exception possible
	add unsigned	addu \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; exception possible
	subtract unsigned	subi \$1,\$2,\$3	\$1 = \$2 - \$3	3 operands; exception possible
	add immediate unsigned	addi \$1,\$2,100	\$1 = \$2 + 100	+ constant; exception possible
	Move from	mfc0 \$1,\$epc	\$1 = \$epc	Used to get of

	coprocessor register			Exception PC
Logical	and	and $1,$2,$3	$1 = $2 & $3	3 register operands; Logical AND
	or	or $1,$2,$3	$1 = $2 | $3	3 register operands; Logical OR
	and immediate	and $1,$2,100	$1 = $2 & 100	Logical AND register, constant
	or immediate	or $1,$2,100	$1 = $2 | 100	Logical OR register, constant
	shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
	shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
Data transfer	load word	lw $1,(100)$2	$1 = Memory[$2+100]	Data from memory to register
	store word	sw $1,(100)$2	Memory[$2+100] = $1	Data from memory to register
	load upper immediate	lui $1,100	$1 = 100 * 2^{16}	Load constant in upper 16bits
Conditional branch	branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100	Equal test; PC relative branch
	branch on not equal	bne $1,$2,100	if ($1 != $2) go to PC+4+100	Not equal test; PC relative
	set on less than	slt $1,$2,$3	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; 2`s complement
	set less than immediate	slti $1,$2,100	if ($2 < 100) $1 = 1; else $1 = 0	Compare < constant; 2`s complement
	set less than unsigned	sltu $1,$2,$3	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; natural number
	set less than immediate unsigned	sltiu $1,$2,100	if ($2 < 100) $1 = 1; else $1 = 0	Compare constant; natural number
Unconditional jump	jump	j 10000	goto 10000	Jump to target address
	jump register	j $31	goto $31	For switch, procedure return
	jump and link	jal 10000	$31 = PC + 4;go to 10000	For procedure call

MIPS Floating-Point Operands

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2,..., $f31	MIPS floating point register are used in pairs for double precision numbers. Odd numbered registers cannot be used for arithmetic or branch, just for data transfer of the right "half" of double precision register pairs.
2^{30} memory words	Memory[0], Memory[4],..., Memory[4293967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled

		registers, such as those saved on procedure calls

MIPS Floating-Point Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	Floating-Point add (single precision)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6	Floating-Point sub (single precision)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 * $f6	Floating-Point multiply (single precision)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	Floating-Point divide (single precision)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	Floating-Point add (double precision)
	FP.dubtract double	.dub.d $f2,$f4,$f6	$f2 = $f4 - $f6	Floating-Point sub (double precision)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 * $f6	Floating-Point multiply (double precision)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	Floating-Point divide (double precision)
Data transfer	load word coprocessor 1	lwc1 $f1,100($2)	$f1 = Memory[$2+100]	32-bit data to FP register
	store word coprocessor 1	swc1 $f1,100($2)	Memory[$2+100] = $f1	32-bit data to memory
Conditional branch	branch on FP true	bc1t 100	if (cond == 1) go to PC+4+100	PC relative branch if FP condition
	branch on FP false	bc1f 100	if (cond == 0) go to PC+4+100	PC relative branch if not condition
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond=1; else cond=0	Floating-point compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond=1; else cond=0	Floating-point compare less than double precision

Pseudo instructions

The MIPS standard defines the CPU instruction set as well as pseudo instructions. These pseudo instructions are translated by the assembler into real MIPS instructions.

Examples:

Pseudo instruction	MIPS instruction	Remark
not r, s	nor r, s, $0	
move r, s	or r, s, $0	
li r, c	ori r, $0, c	load immediate (c: 16 bit constant)

3.2 Pseudo-Ops

We will now describe some pseudo op-codes (directives). These pseudo op-codes influence the assembler's later behavior. Note that unlike processor Opcodes, which are specific to a given machine, the kind and nature of the pseudo-ops are specific to a given assembler, because they are executed by the assembler itself.

Pseudo-Op	Description
`.2byte` *expression1* [, *expression2*] ... [, *expressionN*]	Truncates the expressions in the comma-separated list to 16-bit values and assembles the values in successive locations. The expressions must be absolute or in the form of a label difference (*label1* - *label2*) if both labels are defined in the same section. This directive optionally can have the form *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive does no automatic alignment. (64-bit and N32 only)
`.4byte`*expression1* [, *expression2*] ... [, *expressionN*]	Truncates the expressions in the comma-separated list to 32-bit values and assembles the values in successive locations. The expressions must be absolute or in the form of a label difference (*label1* - *label2*) if both labels are defined in the same section. This directive optionally can have the form *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive does no automatic alignment. (64-bit and N32 only)
`.8byte`*expression1* [, *expression2*] ... [, *expressionN*]	Truncates the expressions in the comma-separated list to 64-bit values and assembles the values in successive locations. The expressions must be absolute or in the form of a label difference (*label1* - *label2*) if both labels are defined in the same section. This directive optionally can have the form *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive does no automatic alignment. (64-bit and N32 only)
`.aent`*name,*`symno`	Sets an alternate entry point for the current procedure. Use this information when you want to generate information for the debugger. It must appear inside an `.ent`/`.end` pair.
`.align`*expression*	Advances the location counter to make the expression low order bits of the counter zero. Normally, the `.half`, `.word`, `.float`, and `.double` directives automatically align their data appropriately. For example, `.word` does an

	implicit `.align 2` (`.double` does an `.align 3`). You disable the automatic alignment feature with `.align 0`. The assembler reinstates automatic alignment at the next `.text`, `.data`, `.rdata`, or `.sdata` directive. Labels immediately preceding an automatic or explicit alignment are also realigned. For example, `foo: .align 3; .word 0` is the same as `.align 3; foo: .word 0`.
`.ascii`*string* [, *string*]...	Assembles each string from the list into successive locations. The `.ascii` directive does not null pad the string. You **must** put quotation marks (") around each string. You can use the backslash escape characters..
`.asciiz` *string* [, *string*]...	Assembles each string in the list into successive locations and adds a null. You can use the backslash escape characters.
.`byte` *expression1* [, *expression2*] ... [, *expressionN*]	Truncates the expressions from the comma-separated list to 8-bit values, and assembles the values in successive locations. The expressions must be absolute. The operands can optionally have the form: *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times.
.`comm`*name*, *expression* [*alignment*]	Unless defined elsewhere, *name* becomes a global common symbol at the head of a block of expression bytes of storage. The linker overlays like-named common blocks, using the maximum of the *expressions*. The 64-bit and N32 assembler also accepts an optional value which specifies the alignment of the symbol.
.`cpadd` *reg*	Emits code that adds the value of "`_gp`" to reg.
.`cpload`*reg*	Expands into the three instructions function prologue that sets up the `$gp` register. This directive is used by position-independent code.
.`cplocal`*reg*	Causes the assembler to use *reg* instead of *$gp* as the context pointer. This directive is used by position-independent code. (64-bit and N32 only)
.`cprestore`*offset*	Causes the assembler to emit the following at the point where it occurs: `sw $gp, offset ($sp)` Also, causes the assembler to generate `lw $gp, offset ($sp)` after every JAL or BAL operation. Offset should point to the saved register area. This directive is used by position-independent code following the caller-saved `gp` convention.
`.cpreturn`	Causes the assembler to emit the following at the point where it occurs: `ld $gp, offset ($sp)` The *offset* is obtained from the previous `.cpsetup` pseudo-op. (64-bit and N32 only)

`.cpsetup` *reg1*, {*offset* | *reg2*}, *label*	Causes the assembler to emit the following at the point where it occurs:
	`sd $gp, offset ($sp)`
	`lui $gp, 0 {label}`
	`daddiu $gp, $gp, 0 { label }`
	`daddu $gp, $gp, reg1`
	`ld $gp, offset ($sp)`
	This sequence is used by position-independent code following the callee-saved `gp` convention. It stores `$gp` in the saved register area and calculates the virtual address of *label* and places it in *reg1*. By convention, *reg1* is `$25` (`t9`).
	If *reg2* is used instead of offset, `$gp` is saved and restored to and from this register.
	(64-bit and N32 only)

`.data`	Tells the assembler to add all subsequent data to the data section.
`.double`*expression* [, *expression2*] ...[, *expressionN*]	Initializes memory to 64-bit floating point numbers. The operands optionally can have the form: *expression1* [: *expression2*]. The *expression1* is the floating point value. The optional *expression2* is a non-negative expression that specifies a repetition count. The *expression2* replicates *expression1*'s value *expression2* times. This directive aligns its data and any preceding labels automatically to a double-word boundary. You can disable this feature by using `.align0`.
`.dword` *expression* [, *expression2*] ...[, *expressionN*]	Truncates the expressions in the comma-separated list to 64-bits and assembles the values in successive locations. The expressions must be absolute. The operands optionally can have the form: *expression1* [:*expression2*]. The *expresssion2* replicates *expression1*'s value *expression2* number of times. The directive aligns its data and preceding labels automatically to a double word boundary. You can disable this feature by using `.align0`.
`.dynsym` *namevalue*	Specifies an ELF `st_other` value for the object denoted by *name* (64-bit and N32 only).
`.end` [*proc_name*]	Sets the end of a procedure. Use this directive when you want to generate information for the debugger. To set the beginning of a procedure, see `.ent`.
`.endr`	Signals the end of a repeat block. To start a repeat block, see `.repeat`.
`.ent`*proc_name*	Sets the beginning of the procedure *proc_name*. Use this directive when you want to generate information for the debugger. To set the end of a procedure, see `.end`.
`.extern`*name expression*	*name* is a global undefined symbol whose size is assumed to be *expression* bytes. The advantage of using this directive, instead of permitting an undefined symbol to become global by default, is that the assembler can

	decide whether to use the economical `$gp`-relative addressing mode, depending on the value of the `-G` option. As a special case, if expression is zero, the assembler refrains from using `$gp` to address this symbol regardless of the size specified by `-G`.
`.file`*file_number file_name_string*	Specifies the source file corresponding to the assembly instructions that follow. For use only by compilers, not by programmers; when the assembler sees this, it refrains from generating line numbers for `dbx` to use unless it also sees `.loc` directives.
`.float` *expression1* [, *expression*2] ... [, *expressionN*]	Initializes memory to single precision 32-bit floating point numbers. The operands optionally can have the form: *expression1*<`_newline`>[: *expression2*]. The optional *expression2* is a non-negative expression that specifies a repetition count. This optional form replicates *expression1*'s value *expression2* times. This directive aligns its data and preceding labels automatically to a word boundary. You can disable this feature by using `.align0`.
`.fmask`*mask offset*	Sets a mask with a bit turned on for each floating point register that the current routine saved. The least-significant bit corresponds to register `$f0`. The offset is the distance in bytes from the virtual frame pointer at which the floating point registers are saved. The assembler saves higher register numbers closer to the virtual frame pointer. You must use `.ent` before `.fmask` and only one `.fmask` may be used per `.ent`. Space should be allocated for those registers specified in the `.fmask`.
`.frame`*frame-register offset return_pc_register*	Describes a stack frame. The first register is the frame-register, the offset is the distance from the frame register to the virtual frame pointer, and the second register is the return program counter (or, if the first register is `$0`, this directive shows that the return program counter is saved four bytes from the virtual frame pointer). You must use `.ent` before `.frame` and only one `.frame` may be used per `.ent`. No stack traces can be done in the debugger without `.frame`.
`.globl`*name*	Makes the *name* external. If the *name* is defined otherwise (by its appearance as a label), the assembler will export the symbol; otherwise it will import the symbol. In general, the assembler imports undefined symbols (that is, it gives them the UNIX storage class "global undefined" and requires the linker to resolve them).
`.gpvalue`*number*	Sets the offset to use in gp_rel relocations; 0 by default. (64-bit and N32 only).
`.gpword`*local-sym*	This directive is similar to `.word` except that the relocation entry for *local-sym* has the `R_MIPS_GPREL32` type. After

	linkage, this results in a 32-bit value that is the distance between *local-sym* and `gp`. *local-sym* must be local. This directive is used by the code generator for PIC switch tables.
`.half`*expression1* [, *expression2*] ... {, *expressionN*]	Truncates the expressions in the comma-separated list to 16-bit values and assembles the values in successive locations. The *expressions* must be absolute. This directive optionally can have the form: *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive automatically aligns its data appropriately. You can disable this feature by using `.align0`.
`.lab`*label_name*	Associates a named label with the current location in the program text. For use by compilers.
`.lcomm`*name, expression*	Makes the *name*'s data type `bss`. The assembler allocates the named symbol to the `bss` area, and the *expression* defines the named symbol's length. If a `.globl` directive also specifies the name, the assembler allocates the named symbol to external `bss`. The assembler puts `bss` symbols in one of two `bss` areas. If the defined size is smaller than (or equal to) the size specified by the assembler or compiler's `-G` command line option, the assembler puts the symbols in the `sbss` area and uses `$gp` to address the data.
`.loc`*file_number line_number [column]*	Specifies the source file and the line within that file that corresponds to the assembly instructions that follow. For use by compilers. The assembler ignores the file number when this directive appears in the assembly source file. Then, the assembler assumes that the directive refers to the most recent `.file` directive. The 64-bit and N32 assembler also supports an optional value that specifies the column number.
`.mask`*mask, offset*	Sets a mask with a bit turned on for each general purpose register that the current routine saved. For use by compilers. Bit one corresponds to register `$1`. The offset is the distance in bytes from the virtual frame pointer where the registers are saved. The assembler saves higher register numbers closer to the virtual frame pointer. Space should be allocated for those registers appearing in the mask. If bit zero is set it is assumed that space is allocated for all 31 registers regardless of whether they appear in the mask.
`.nada`	Tells the assembler to put in an instruction that has no effect on the machine state. It has the same effect as `nop` (described below), but it produces more efficient code on an R8000. (64-bit and N32 only)

`.nop`	Tells the assembler to put in an instruction that has no effect on the machine state. While several instructions cause no-operation, the assembler only considers the ones generated by the `nop` directive to be wait instructions. This directive puts an explicit delay in the instruction stream. Note: Unless you use `.set noreorder`, the reorganizer may eliminate unnecessary `nop` instructions.
`.option`*options*	Tells the assembler that certain options were in effect during compilation. (These options can, for example, limit the assembler's freedom to perform branch optimizations.) This *option* is intended for compiler-generated `.s` files rather than for hand-coded ones.
`.origin`*expression*	Specifies the current offset in a section to the value of *expression*. (64-bit and N32 only)
`.repeat`*expression*	Repeats all instructions or data between the `.repeat` directive and the `.endr` directive. The expression defines how many times the data repeats. With the `.repeat` directive, you cannot use labels, branch instructions, or values that require relocation in the block. To end a `.repeat`, see `.endr`.
`.rdata`	Tells the assembler to add subsequent data into the `rdata` section.
`.sdata`	Tells the assembler to add subsequent data to the `sdata` section.
`.section`*name* [, *section type, section flags, section entry size, section alignment*]	Instructs the assembler to create a section with the given name and optional attributes. Legal *section type* values are denoted by variables prefixed by `SHT_ in`<*elf.h*>. Legal *section flags* values are denoted by variables prefixed by `SHF_ in`<*elf.h*>. The *section entry size* specifies the size of each entry in the section. For example, it is 4 for `.text` sections. The *section alignment* specifies the byte boundary requirement for the section. For example, it is 16 for `.text` sections. (64-bit and N32 only)
`.set`*option*	Instructs the assembler to enable or to disable certain options. Use `.set` options only for hand-crafted assembly routines. The assembler has these default options: `reorder`, `macro`, and `at`. You can specify only one option for each `.set` directive. You can specify these `.set` options: The `reorder` option lets the assembler reorder machine language instructions to improve performance. The `noreorder` option prevents the assembler from

	reordering machine language instructions. If a machine language instruction violates the hardware pipeline constraints, the assembler issues a warning message. The `macro` option lets the assembler generate multiple machine instructions from a single assembler instruction. The `nomacro` option causes the assembler to print a warning whenever an assembler operation generates more than one machine language instruction. You must select the `noreorder` option before using the `nomacro` option; otherwise, an error results. The `at` option lets the assembler use the $at register for macros, but generates warnings if the source program uses `$at`. When you use the `noat` option and an assembler operation requires the `$at` register, the assembler issues a warning message; however, the `noat` option does let source programs use `$at` without issuing warnings. The `nomove` option tells the assembler to mark each subsequent instruction so that it cannot be moved during reorganization. Because the assembler can still insert `nop` instructions where necessary for pipeline constraints, this option is less stringent than `noreorder`. The assembler can still move instructions from below the nomove region to fill delay slots above the region or vice versa. The `nomove` option has part of the effect of the "volatile" C declaration; it prevents otherwise independent loads or stores from occurring in a different order than intended. The `move` option cancels the effect of `nomove`. The `notransform` option tells the assembler to mark each subsequent instruction so that it cannot be transformed by `pixie`(1) into an equivalent set of instructions. There are restrictions on the use of this option in order to guarantee `pixie`'s ability to still produce code that will execute correctly with the preceding/following transformed code. The sequence of instructions marked `notransform` must behave like a single basic block (i.e., there is only one entry and exit from the sequences and they are the first and last instructions, respectively). If this restriction cannot be met, correct transformed execution can still be guaranteed if the sequence of instructions does not use any of the saved registers `$16..$23, $30` (s0-s8). The `transform` option cancels the effect of `notransform`.
`.size`*name, expression*	Specifies the size of an object denoted by *name* to the value of *expression.*
`.space` *expression*	Advances the location counter by the value of the specified *expression* bytes. The assembler fills the space

	with zeros.
`.struct` *expression*	This permits you to lay out a structure using labels plus directives like `.word`, `.byte`, and so forth. It ends at the next segment directive (`.data`, `.text`, etc.). It does not emit any code or data, but defines the labels within it to have values which are the sum of *expression* plus their offsets from the `.struct` itself.
(*symbolic equate*)	Takes one of these forms: name = *expression* or *name* = *register*. You must define the name only once in the assembly, and you **cannot** redefine the name. The expression must be computable when you assemble the program, and the expression must involve operators, constants, and equated symbols. You can use the name as a constant in any later statement.
`.text`	Tells the assembler to add subsequent code to the text section. (This is the default.)
`.type` *name, value*	Specifies the `elf` type of an object denoted by name to value. Legal `elf` type values are denoted by variables prefixed by `STT_ in` *<elf.h>*. (64-bit and N32 only)
`.verstamp`*major minor*	Specifies the major and minor version numbers (for example, version 0.15 would be `.verstamp` 0 15).
`.weakext`*weak_name* [*strong_name*]	Defines a weak external name and optionally associates it with the *strong_name*.
`.word`*expression1* [, *expression2*] ... [, *expressionN*]	Truncates the *expression*s in the comma-separated list to 32-bits and assembles the values in successive locations. The *expression*s must be absolute. The operands optionally can have the form: *expression1*`<_newline>`[: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive aligns its data and preceding labels automatically to a word boundary. You can disable this feature by using `.align0`.

The directives listed below are only accepted in `-o32` compiles; they are only meant for compiler-generated code, and should not be used in hand-written assembly code.

```
.alias
.asm0
.bgnb
.endb
.err
.gjaldef
.gjallive
.gjrlive
.livereg
.noalias
.set bopt/nobopt
```

```
.vreg
```

3.3 Accessing Data in Memory – Addressing Modes

3.3.1 Accessing Data

There are a number of ways to refer to data, either as source operands, or destination locations for storage. This section starts with these different methods from the programmer's point of view, and discusses how these are encoded in the MIPS instruction formats.

Immediate
Source operands can be constants. A constant value is usually encoded directly in the machine language instruction, so that it is available immediately after the instruction is decoded, without fetching it from memory. Understandably, this does not provide any location for storing data.

MIPS instructions encode immediate constants in the lower 16 bits of the immediate instruction layout. For constants larger than 16 bits, the assembler generates 2 machine instructions, the upper half of the constant is loaded into the $at register with the "load upper immediate" instruction.

Examples of instructions with immediate data:

```
    addi $t0,t1,65

    sub  $t0,7             # assembled as: addi $t0, $t0, -7

    li   $t3, 0x12345678   # assembled as
    lui  $at, 0x1234
    ori  $t3, $at, 0x5678  # puts it all together

    bgez $t5, 16           # skip 4 instructions ahead if $t5
                           # is non-negative   (4 is encoded
                           # in immediate field)
```

The last example is also referred to as ***PC-relative addressing***, because the immediate value is (shifted left 2 bits and) added to the PC when the branch is taken. The constant is often referred to as an offset or displacement. It is a signed number, so the new PC value can be before (a loop) or after the current instruction.

This mode is very useful because the operating system can move the program to another part of memory without affecting branch instructions.

Most processors use ***PC-relative addressing*** for branch instructions. The number of bits used for the displacement limits the distance to the branch target. In older designs, such as the 6502 and 80x86, 8 bits are used, limiting the displacement to + or - 128 bytes before or after.

With MIPS, the limit is 128Kbytes. Since branch instructions are usually used within procedures, this is unlikely to cause you any problems!

Register addressing

Registers may be used as sources and destinations of data. Access to registers is extremely fast, much faster than fetching data from memory. All the instructions listed above reference 1 or 2 registers as well as the immediate constant. In addition, many instructions reference registers exclusively. Most processors require at least one operand of arithmetic instructions to be in a register. Instructions with all operands in registers (or immediate) are the fastest to execute.

MIPS provides 32 registers, and encourages you to load your data into them and work with it there, resulting in very fast execution. Since it takes only 5 bits to specify a register, as opposed to 32 bits for a memory location, the register instruction layout can easily accommodate 3 register designations.

Register name	Number	Usage
$zero	0	Constant 0
$at	1	Reserved for assembler
$v0	2	Expression evaluation and results of a function
$v1	3	Expression evaluation and results of a function
$a0	4	Argument 1
$a1	5	Argument 2
$a2	6	Argument 3
$a3	7	Argument 4
$t0	8	Temporary (not preserved across call)
$t1	9	Temporary (not preserved across call)
$t2	10	Temporary (not preserved across call)
$t3	11	Temporary (not preserved across call)
$t4	12	Temporary (not preserved across call)
$t5	13	Temporary (not preserved across call)
$t6	14	Temporary (not preserved across call)
$t7	15	Temporary (not preserved across call)
$s0	16	Saved temporary (not preserved across call)
$s1	17	Saved temporary (not preserved across call)
$s2	18	Saved temporary (not preserved across call)
$s3	19	Saved temporary (not preserved across call)
$s4	20	Saved temporary (not preserved across call)
$s5	21	Saved temporary (not preserved across call)
$s6	22	Saved temporary (not preserved across call)
$s7	23	Saved temporary (not preserved across call)
$t8	24	Temporary (not preserved across call)
$t9	25	Temporary (not preserved across call)
$k0	26	Reserved for OS kernel
$k1	27	Reserved for OS kernel
$gp	28	Pointer to global area

$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address (used by function call)

The MIPS (and SPIM) central processing unit contains 32 general purpose 32-bit registers that are numbered 0-31. Register n is designated by $n. Register $0 always contains the hardwired value 0. MIPS has established a set of conventions as to how registers should be used. These suggestions are guidelines, which are not enforced by the hardware. However a program that violates them will not work properly with other software.

Registers $at (1), $k0 (26), and $k1 (27) are reserved for use by the assembler and operating system.

Registers $a0-$a3 (4-7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers $v0 and $v1 (2, 3) are used to return values from functions. Registers $t0-$t9 (8-15, 24, 25) are caller-saved registers used for temporary quantities that do not need to be preserved across calls. Registers $s0-$s7 (16-23) are callee-saved registers that hold long-lived values that should be preserved across calls.

Register $sp (29) is the stack pointer, which points to the last location in use on the stack. Register $fp (30) is the frame pointer. Register $ra (31) is written with the return address for a call by the jal instruction.

Register $gp (28) is a global pointer that points into the middle of a 64K block of memory in the heap that holds constants and global variables. The objects in this heap can be quickly accessed with a single load or store instruction.

Examples of register instructions:

```
addu $t3, $t1, $t2      # add (unsigned) $t3 := $t1 + $t2
sub  $t0, $t3           # subtract (signed) $t0 := $t0 - $t3
```

Memory addressing

With 32 bit addresses, a computer can access as much memory as you can afford to buy. Well, up to 4 Gigabytes. You can store lots of data there, and transfer it to and from the processor as needed. In return for large amounts of storage, access is slower. Typically an instruction will allow access to at most one memory address. MIPS, in common with other RISC processors and most supercomputers, restricts this to load and store instructions.

There are several addressing modes in which we could specify the address:

Name	Format	Address Computation
Direct addressing	Label	Address of label

Indexed addressing	Label(register)	Address of label + contents of register
Indirect addressing	(register)	Contents of register
Base addressing	Imm(register)	Immediate + contents of register

Direct addressing - fixed address built in to the instruction

Also called direct addressing, the known address can be built into the instruction as a constant. Programmers usually specify this address using a variable name or label in the data segment, the compiler or assembler assigns it a numeric value. Since it is a 32-bit value, a MIPS assembler will break it into two 16-bit immediate quantities, and assemble 2 machine instructions to do the job.

Examples:

```
lw  $t0, MyNumber        # load the word into $t0
sb  $t9, firstInitial    # store rightmost 8 bits of $t9 in
                         # data segment
```

Indexed addressing

Suppose we want to index an array. We know the address of the (start of) the array, and want a register to index it. Now the address is fixed, and the register will hold a variable offset. The following code will copy 10 bytes (the assumed length of both strings, including final 0) from str1 to str2, using $t0 going from 9 down to 0.

Example:

```
      li   $t0,9            # t0 indexes arrays
copyloop:
      lb   $t1, str1($t0)   # t1 used to transfer character
      sb   $t1, str2($t0)   # to str2
      sub  $t0, 1           # decrement array index
      bgez $t0, copyloop    # repeat until t0 < 0
```

Note that if we were moving words, we would need to decrement the index by 4 instead of 1.

To implement this in MIPS, the assembler needs to produce 3 machine instructions for each indexed instruction, because the 32-bit constant address must be split into 2 16 bit immediate values.

MIPS, in keeping with the RISC philosophy, actually has only one memory addressing mode at the machine level, it corresponds to base addressing. Indirect addressing is simply a special case with offset = 0.

Indirect addressing - register contains the address

Besides holding data, a register can be used as a pointer, holding an address that points to the data. The data is transferred by putting the contents of the register on

the address bus, the data is transferred on the data bus. This is called indirect addressing because the register itself is not the target, rather it points to the target in memory indirectly. A common use of this is to step through a string one character at a time. One first loads a register with the address of the string, and then
uses it to access the characters in succession, incrementing the register each time. For example, suppose we have the string:

  catStr: .asciiz   "cat"

The address can be loaded into a register with the load-address instruction (in machine language it is just like load immediate, except the constant is the address rather than data), and then refer to the characters indirectly.

```
    la  $t0, catStr
    lb  $t1, ($t0)         # 'c' is now in $t1
    addi      $t0, 1      # point to next character
    lb  $t2, ($t0)        # 'a' is now in $t2
```

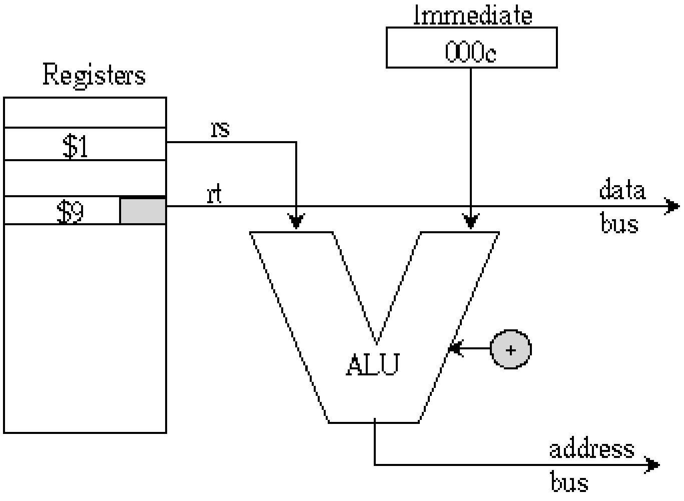
Base addressing

A variation of indirect addressing is useful for referring to fields of a record or struct. Suppose we store telephone numbers in a record of 4 short integers (16-bit half words), holding area code, prefix, 4-digit number, and extension. Then the extension number starts 6 bytes from the beginning of the record. We can get it by adding the offset 6 to a register pointing to the record:

```
    la    $t0, MyPhone     # $t0 points to a telephone record
    lh    $t1, 6($t0)      # $t1 loaded with the extension
                           # in that record
```

Another example shows the layout of the actual sb $9, 12($1) base addressed instruction, and a diagram explaining how the immediate 12 is added to $1, the sum placed on the address bus, and $9 (a character) on the data bus.

op code sb	rs ($1)	rt - $9	immediate (offset) 0x000c
101000	00001	01001	0000 0000 0000 1100



3.4 Subroutine Linkage and stacks

3.4.1 MIPS function call and return

When we call a function (procedure, subroutine), we normally want to come back to the place we left, the following instruction. To do so, we need a means of remembering our place, so the function can return to it. Some convention needs to be agreed upon as to where to store a return address. Some conventions that are in use are:
The upper left-hand corner of the envelope, for a letter sent by Canada Post,
The word preceding the jump target, in a computer program
The top of a stack data structure, which has the LIFO (last in first out) property
A designated register.

MIPS uses register **31** as the ***return address register***

We have actually seen the jump instructions involved in function call and return, they are:

jal - jump and link. It has the same format as jump. In addition to jumping, it stores the return address, that is the address of the following instruction, in register 31, named $ra, the return address register.
jr - jump register. Instead of a fixed target address, given as a label, this instruction jumps to the location contained in a register. jr $ra in particular jumps back to the location last saved in the return address register.
As an example, we could have a simple function to print an integer, that takes care of the detail of remembering the system call number, and another to print an end-of-line character:

```
print_int:
     li  $v0, 1
     syscall
     jr  $ra

endline:
     la  $a0, endl  #define in    .data as "\n"
     li  $v0, 4
     syscall
     jr  $ra
```

These can then be called to output some numbers

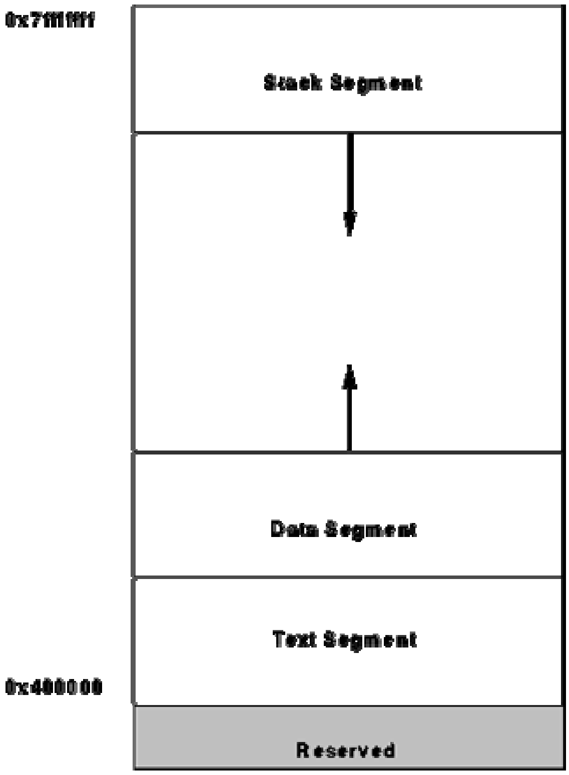
```
li  $a0, 42        # writeln (42)
jal print_int
jal endline
li  $a0, 1999      # writeln (1999)
jal print_int
jal endline
```

This suffices for a main program calling functions sequentially. However, it does not allow for a function to call another function, so later on we will see how to save return addresses systematically, using a stack.

3.4.2 The Principles of the stack

Use a stack like a stack of papers, a stack is known to accountants as a "Last In, First Out" (LIFO) data structure. This is ideal for functions as typically they need to save some registers on the way in, and retrieve the values on the way out. As functions call each other, they pile up more items on the stack, as each one returns, they (must!) remove the values they piled on. This is such a useful method that **all** operating systems set up space for a stack for programs, and initialize a **stack pointer.**

First, let’s have a look at the location of the stack in the memory:



The organization of memory in MIPS systems is conventional. A program's address space is composed of three parts.

At the bottom of the user address space (0x400000) is the text segment, which holds the instructions for a program.

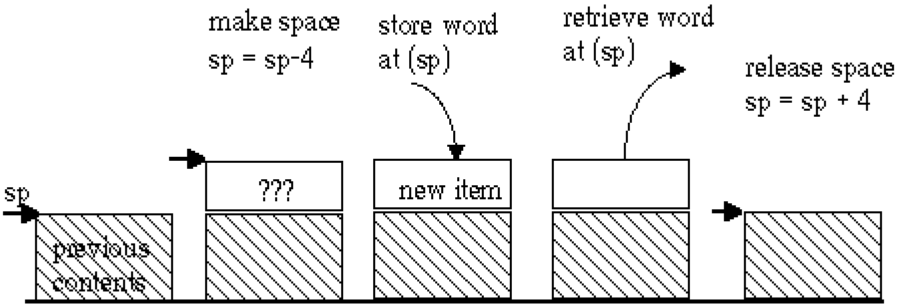
Above the text segment is the data segment (starting at 0x10000000), which is divided into two parts. The static data portion contains objects whose size and address are known to the compiler and linker. Immediately above these objects is dynamic data. As a program allocates space dynamically (i.e., by malloc), the sbrk system call moves the top of the data segment up.
The program stack resides at the top of the address space (0x7fffffff). It grows down, towards the data segment. The *stack pointer* always points to the *top* word on the stack.

The permitted stack operations are:

- **push:** To place a new item on the stack, the *stack pointer is first decremented*, and then the item is stored at the new location pointed to.
- **pop:** To remove an item from the stack, the value pointed to by the stack pointer is copied (usually into a register), and *then* the stack pointer is incremented, exactly reversing the push operation.

These customary terms allude to a stack of plates on a cafeteria counter, the kind that uses a spring pushed down by the weight of the plates. When plates are removed, the stack "pops" up. Ideally, it is impossible to take any but the top plate off the stack.

This figure shows the operations on a stack. In any 32-bit machine, a 32 bit word is always pushed or popped, so the sp is always kept a multiple of 4.



CISC processors typically have instructions that *implicitly* manipulate the stack pointer, usually named SP. They are PUSH, POP, CALL and RET. The last two store and retrieve return addresses directly to and from the stack.

In MIPS, the stack pointer is $29, named $sp. It is only made special by the fact that the operating system sets up the stack for you, with $sp pointing to the initial top of stack.

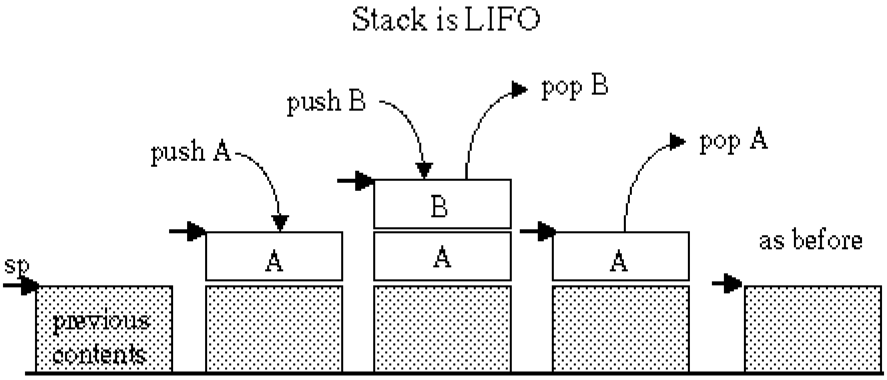
To push, the sequence is

```
sub   $sp, 4      #PUSH on the stack
sw    $ra, ($sp)  #our return address
```

Pop reverses this order, first retrieving the value previously pushed, then incrementing the stack pointer:

```
lw    $ra, ($sp) #POP our return address from the stack
add   $sp, 4     #top of stack back as it was before the PUSH
```

The stack is a last-in-first-out data structure. This means you need to pop items in the reverse order they were pushed. It is generally important to pop precisely the items that were pushed, rather than abandoning them.



3.4.3 MIPS Calling Convention

For convenience, we repeat the register usage conventions here:

Number	Name	Purpose
$0	$0	Always 0
$1	$at	The *Assembler Temporary* used by the assembler in expanding pseudo-ops.
$2-$3	$v0-$v1	These registers contain the *Returned Value* of a subroutine; if the value is 1 word only $v0 is significant.
$4-$7	$a0-$a3	The *Argument* registers, these registers contain the first 4 argument values for a subroutine call.
$8-$15, $24,$25	$t0-$t9	The *Temporary Registers*.
$16-$23	$s0-$s7	The *Saved Registers*.
$26-$27	$k0-$k1	The *Kernel Reserved registers.* DO NOT USE.
$28	$gp	The *Globals Pointer* used for addressing static global variables.
$29	$sp	The *Stack Pointer.*
$30	$fp	The *Frame Pointer*, if needed
$31	$ra	The *Return Address* in a subroutine call.

The shaded text indicates registers whose value must be preserved across subroutine calls.

Stack Frames

Remember, every time a subroutine is called, a unique stack frame is created for that instance of the subroutine call. (In the case of a recursive subroutine call, multiple stack frames are created, one for each instance of the call.)

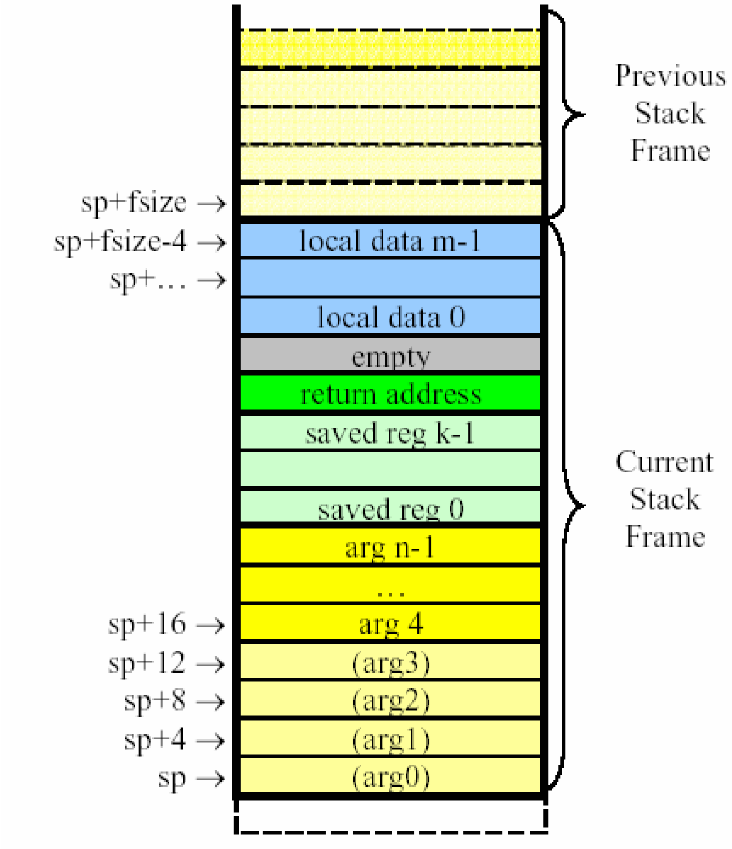
The organization of the stack frame is important for two reasons:

1. It forms a contract between the caller and the callee that defines how arguments are passed between the caller and callee, how the result of a function is passed from the callee to the caller, and defines how registers are shared between the caller and the callee.
2. It defines the organization of storage local to the callee within its stack frame.

In general, the stack frame for a subroutine may contain space to contain the following:

- Space to store arguments passed to subroutines that this subroutine calls.
- A place to store the values of saved registers ($s0 to $s7).
- A place to store the subroutine's return address ($ra).
- A place for local data storage.

The following figure illustrates the general organization of a stack frame.



We divide the stack frame into five regions:

1. The Argument Section of a stack frame contains the space to store the arguments that are passed to any subroutine that is called by the current subroutine (i.e., the subroutine whose stack frame is currently on top of the stack.) The first four words of this section are never used by the current

subroutine; they are reserved for use by any subroutine called by the current subroutine. (Recall that up to four arguments can be passed to a subroutine in the argument registers ($a0 to $a3)). If there are more than four arguments, the current subroutine stores them on the stack, at addresses sp+16, sp+20, sp+24, etc.

2. The Saved Registers Section of the stack frame contains space to save the values of any of the saved registers ($s0 to $s7) that the current subroutine wants to use. On entry into the current subroutine, the subroutine copies the values of any of the saved registers, $s0 to $s7, whose values it might change during the course of the execution of the subroutine into this section of the stack frame. Just before the subroutine returns, it copies these values from the Saved Registers Section back into the original saved registers. In between, the current subroutine is free to change the value of the saved registers at will. However, the caller of the current subroutine will see the same values in these registers after the subroutine call as it saw before the subroutine call.
3. The Return Address Section is used to store the value of the return address register, $ra. This value is copied onto the stack at the start of execution of the current subroutine and copied back into the $ra register just before the current subroutine returns.
4. The Pad is inserted into the stack frame to make sure that total size of the stack frame is always a multiple of 8. It is inserted here to ensure that the local data storage area starts on a double word boundary.
5. The Local Data Storage Section is used for local variable storage. The current subroutine must reserve enough words of storage in this area for all of its local data, including space to store the value of any temporary registers ($t0 to $t9) that it needs to preserve across subroutine calls. The local data storage area must also be padded so that its size is always a multiple of 8 words.

A couple of additional rules that do not obviously follow from the above:
The value of the stack pointer is required to be multiple of 8 at all times. This ensures that a 64-bit data object can be pushed on the stack without generating an address alignment error at run-time. This implies the size of every stack frame must be a multiple of 8; technically, this requirement applies even to leaf subroutines as discussed below.

The values of the argument registers $a0-$a3 are not required to be preserved across subroutine calls. Thus, a subroutine is allowed to change the values of any of the argument registers without saving/restoring them.
The first four words of the Argument Section of a stack frame are known as *argument slots* -- memory locations reserved to store the four arguments $a0-$a3. It is important to remember that a subroutine does *not* store anything in the first four argument slots, the actual arguments are passed in $a0 to $a3. However, the called subroutine may choose to copy the values of $a0 to $a3 into the argument slots if it wants to save these values. If it does so, then it can then treat all its arguments as a 1-dimensional array in memory.

There are several very important things to note about the argument slots. The argument slots are allocated by the caller but are used by the callee! All four slots are *required*, even if the caller knows it is passing fewer than four arguments. Thus, on entry a subroutine may legally store *all* of the argument registers into the argument slots if desired. The caller must allocate space on its stack frame for the maximum number of arguments for *any* subroutine that it calls (or the minimum of four arguments, if all the subroutines that it calls have fewer than four arguments.)

Leaf vs. Nonleaf Subroutines

Not every subroutine will need every section described above in its stack frame. The general rule is that if the subroutine does not need a section, then it may *omit* that section from its call frame. To help make this clear we will distinguish 3 different classes of subroutines:

- **Simple Leaf** subroutines do not call any other subroutines, do not use any memory space on the stack (either for local variables or to save the values of saved registers). Such subroutines do not require a stack frame, consequently never need to change $sp.
- **Leaf with Data** are leaf subroutines (i.e. do not call any other subroutines) that require stack space, either for local variables or to save the values of saved registers. Such subroutines push a stack frame (the size of which should be a multiple of 8 as discussed above). However, $ra is not saved in the stack frame.
- **Nonleaf** subroutines are those that call other subroutines. The stack frame of a nonleaf subroutine will probably have most if not all the sections. Below are examples for each of these cases.

A Simple Leaf Function:

Consider the simple function:

```
int g( int x, int y )
{
     return (x + y);
}
```

This function does not call any other function, so it does not need to save $ra. Also, it does not require any temporary storage. Thus, it can be written with no stack manipulation at all.
Here it is:

```
g:   add $v0, $a0, $a1        # result is sum of arguments
     jr $ra                   # return
```

Because it has no local data, this function does no stack manipulation at all; its stack frame is of zero size.

A Leaf Function With Data:
Now let's make g a little more complicated:

```
int g( int x, int y )
{
     int a[32];
     ... (calculate using x, y, a);
     return a[0];
}
```

This function does not call any other function, so it does not need to save $ra, but, it does require space for the array a. Thus, it must push a stack frame.
Here is the code:

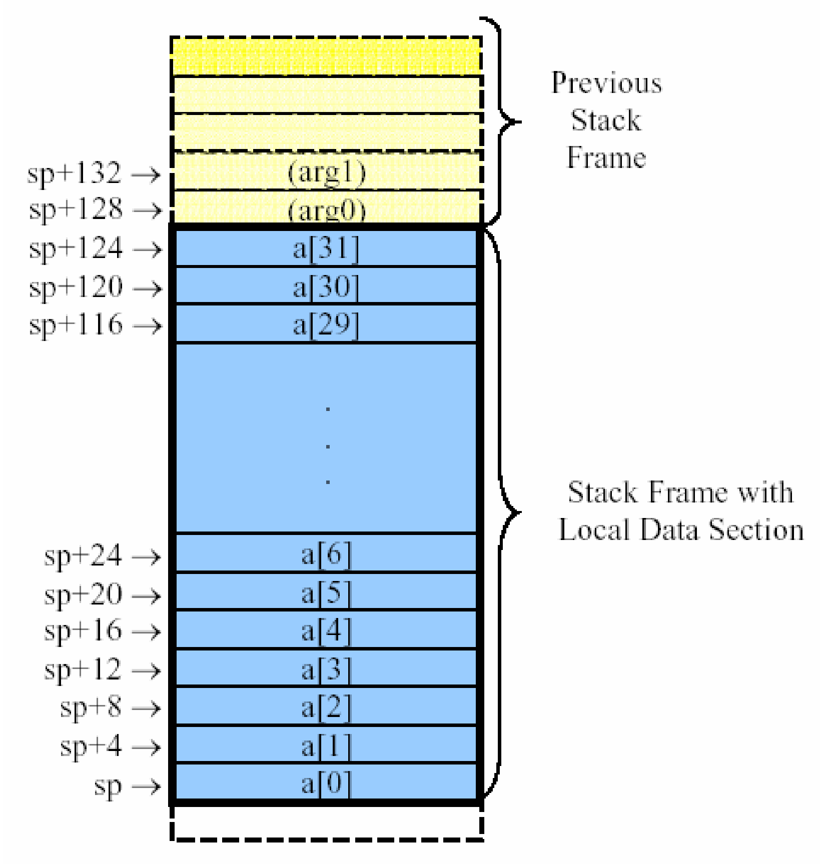
```
g:   # start of prologue
     addiu $sp,$sp,(-128) # push stack frame
     # end of prologue

     . . . # calculate using $a0, $a1 and a
     # array a is stored at addresses
     # 0($sp) to 124($sp)

     lw $v0, 0($sp)      # result is a[0]

     # start of epilogue
     addiu $sp,$sp,128 # pop stack frame
     # end of epilogue

     jr $ra # return
```

Because this is a leaf function, there is no need to save/restore $ra and no need to leave space for argument slots. This stack frame only needs space for local data storage. Its size is 32 words or 128 bytes. Also, we always require that the local data storage block be double word aligned, but in this case since the value of $sp is always double word aligned, no padding is necessary. The array a is stored at addresses 0($sp) to 124($sp).

Also we have gathered the instructions necessary to set up the stack frame into a *prologue* and the instructions necessary to dismantle the stack frame into an *epilogue* to keep them separate from the code for the body of the subroutine.

Note that, in this example, the code for the calculations (…) is *not* allowed to change the values of any of the registers $s0 to $s7.

The stack frame for this example follows.

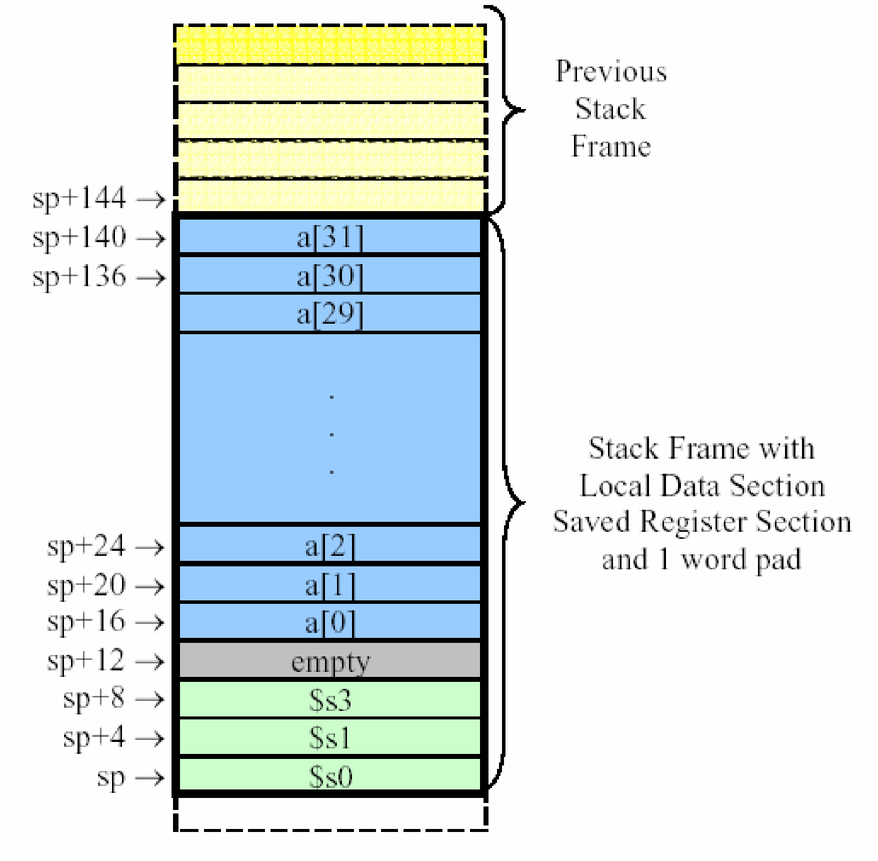


A Leaf Function With Data and Saved Registers:
Suppose that the code for the calculations (…) *does* change the value of some of the saved registers, $s0, $s1, and $s3. We would have to change code for g as follows.

```
g:   # start of prologue
     addiu $sp, $sp, (-144)        # push stack frame
     sw $s0, 0($sp)                # save value of $s0
     sw $s1, 4($sp)                # save value of $s1
     sw $s3, 8($sp)                # save value of $s3
     # end of prologue

     # start of body
     . . . # calculate using $a0, $a1 and a
     # array a is stored at addresses
     # 16($sp) to 140($sp)

     lw $v0, 16($sp)               # result is a[0]
     # end of body


     # start of epilogue
     lw $s0, 0($sp)                # restore value of $s0
     lw $s1, 4($sp)                # restore value of $s1
```

```
        lw $s3, 8($sp)                  # restore value of $s3
        addiu $sp,$sp,144               # pop stack frame
        # end of epilogue

        jr $ra                          # return
```

In this case, we had to add a Saved Register Section and Pad to the stack frame. The Saved Register Section consists of three words, at addresses 0(sp), 4(sp), and 8(sp), that are used to save the value of $s0, $s1, and $s3. The Pad, at address 12(sp) is used to pad the stack frame so that its size is a multiple of 8 and start to the Local Data Section at an address that is also a multiple of 8. The total size of this stack frame is 36 (3+1+32) words or 144 (12+4+128) bytes. Instructions to save the saved registers have been added to the prologue and instructions to restore the saved registers have been added to the epilogue. Note that the addresses used to store the array a increased by 16 as a result of adding this new section, so the array is now stored at addresses 16($sp) to 140($sp).
The stack frame for this example follows:



A Nonleaf Subroutine
Now let's make g even more complicated:

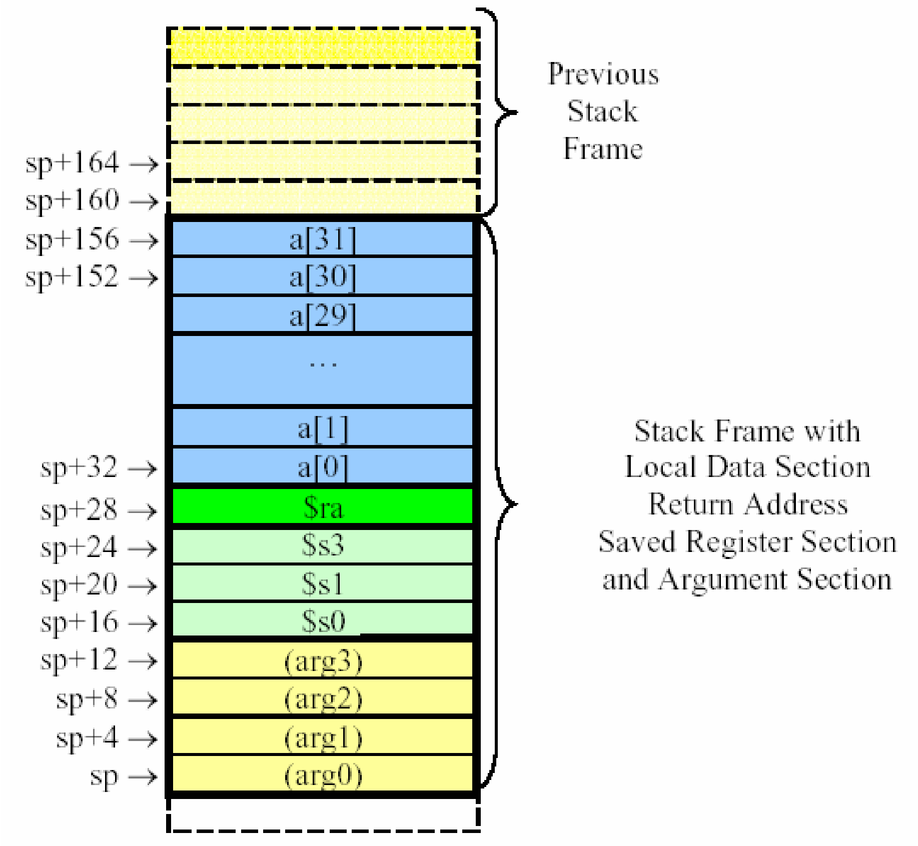
```
int g( int x, int y )
{
     int a[32];
```

```
    ... (calculate using x, y, a);

    a[1] = f(y,x,a[2]);
    a[0] = f(x,y,a[1]);
    return a[0];
}
```

Now g makes two calls to a function f. We would have to change the code for g as follows.

```
g:
        # start of prologue
        addiu $sp, $sp, (-160)          # push stack frame
        sw $ra, 28($sp)                 # save the return address
        sw $s0, 16($sp)                 # save value of $s0
        sw $s1, 20($sp)                 # save value of $s1
        sw $s3, 24($sp)                 # save value of $s3
        # end of prologue

        # start of body
        . . . # calculate using $a0, $a1 and a
        # array a is stored at addresses
        # 32($sp) to 156($sp)
        # save $a0 and $a1 in caller's stack frame
        sw $a0, 160(sp)                 # save $a0 (variable x)
        sw $a1, 164(sp)                 # save $a1 (variable y)

        # first call to function f
        lw $a0, 164(sp)                 # arg0 is variable y
        lw $a1, 160(sp)                 # arg1 is variable x
        lw $a2, 40(sp)                  # arg2 is a[2]
        jal f                           # call f
        sw $v0, 36(sp)                  # store value of f into a[1]

        # second call to function f
        lw $a0, 160(sp)                 # arg0 is variable x
        lw $a1, 164(sp)                 # arg1 is variable y
        lw $a2, 36(sp)                  # arg2 is a[1]
        jal f                           # call f
        sw $v0, 32(sp)                  # store value of f into a[0]

        # load return value of g into $v0
        lw $v0, 32($sp)                 # result is a[0]
        # end of body
        # start of epilogue
        lw $s0,16($sp)                  # restore value of $s0
        lw $s1,20($sp)                  # restore value of $s1
        lw $s3,24($sp)                  # restore value of $s3
        lw $ra,28($sp)                  # restore the return address
        addiu $sp,$sp,160               # pop stack frame
        # end of epilogue

        jr $ra # return
```

In this example, we have added the Argument Section and the Return Address Section to the stack frame and omitted the Pad. Since f, the function that g calls, has 3 arguments, we add the minimum size Argument Section of 4 words. The total size of this stack frame is 40 (4+3+1+32) words or 160 (16+12+4+128) bytes. Once again the addresses used to store the array a increased as a result of adding the new sections, so the array is now stored at addresses 32($sp) to 156($sp).
The stack frame for this example follows:



Summary
The following summarizes the ECE314 MIPS calling convention. A stack frame has 0 to 5 sections depending on the requirements of the subroutine.

A stack frame has a **Argument Section,**
If the subroutine calls other subroutines (i.e., is a non-leaf subroutine).
Don't forget that

1. The Argument Section is large enough to hold the largest number of arguments of any subroutine called by the current subroutine. (One word is reserved per argument.)
2. The Argument Section has a minimum size of 4 words.
3. The argument registers $a0, $a1, $a2, and $a3 are used to pass the first four arguments to a subroutine. These arguments are not copied into the Argument Section by the calling subroutine, although space is reserved for them. Additional arguments are copied to the Argument Section (at offsets 16, 20, 24, etc.)
4. A subroutine that is called by the current subroutine, may, but is not required to, copy the contents of the argument registers into the Argument Section of the current subroutine (at offsets 0, 4, 8 and 12.)
5. The Argument Section is found, at the low end of the stack frame.

A stack frame has a **Saved Registers Section,**

- If the subroutine want to use (i.e., change the value) of any of the saved registers, $s0 to $s7.
- Don’t forget that

1. The Saved Registers Section contains 1 word for each register that it is required to save on entry and restore on exit.
2. The subroutine prologue must copy the value of each register whose value it must save into the Saved Registers Section on entry into the subroutine (e.g., in the subroutine prologue) and must copy the value of each of these registers back into the correct register before it returns (e.g., in the subroutine epilogue.)
3. The subroutine is not required to save/restore the value of a saved register if it does not change the value of the register in the subroutine body.
4. The Saved Registers Section is found just above the Argument Section.

A stack frame has a **Return Address Section,**

- If the subroutine calls other subroutines (i.e., is a non-leaf subroutine).
- Don’t forget that

1. The Return Address Section contains exactly 1 word.
2. The value of the return address register, $ra, is copied to the Return Address Section on entry into the subroutine (e.g., in the subroutine prologue) and copied from the Return Address Section before exit from the subroutine (e.g., in the subroutine epilogue.)
3. The Return Address Section is found just above the Saved Registers Section.

A stack frame has a **Pad,**

- If the number of bytes in the Argument Section, Saved Registers Section, and the Return Address Section is not a multiple of 8.
- Don’t forget that

1. The Pad contains exactly 1 word.

2. Nothing is stored in the Pad.
3. The Pad is found just above the Return Address Section.

A stack frame has a **Local Data Storage Section,**

- If the subroutine has local variables or must save the value of a register that is not preserved across subroutine calls.
- Don't forget that
  1. The size of the Local Data Storage Section is determined by the local data storage requirements of the subroutine, but must always be a multiple of 8 bytes.
  2. The internal organization is entirely determined by the subroutine. This part of the stack is private to the subroutine and is never accessed by any other subroutine, either a caller or callee of the current subroutine.
  3. The Local Data Storage Section is found just above the Pad. This places it at the high end of the stack frame.

The total size of the stack frame for a subroutine is the sum of the sizes of the individual sections (using 0 for the size of sections that it doesn't need.) The subroutine should push its stack frame onto the stack on entry into the subroutine by subtracting the size of its stack frame from the stack pointer, $sp. The subroutine should pop its stack frame off from the stack just before exit from the subroutine by adding the size of its stack frame to the stack pointer, $sp. Make sure that you add the same amount at the end as you subtracted at the beginning!

3.5 Input and Output in Assembly Language

3.5.1 Input and Output - System calls

Controlling the hardware responsible for input and output is a difficult and specialized job, and beyond the scope of this course. All computers have an operating system that provides many services for input/output anyway. The SPIM simulator provides 10 basic services.

The call code always goes in $v0, and the system is called with syscall.

Service	Call Code	Arguments	Result
Print integer	1	$a0 = integer	Integer printed in console window
Print float	2	$f12 = float	Float printed in console window
Print double	3	$f12 = double	Double printed in console window
Print string	4	$a0 = address of string	String printed in console window
Read	5	none	$v0 holds integer that was

integer			entered
Read float	6	none	$f0 holds float that was entered
Read double	7	none	$f0 holds double that was entered
Read string	8	$a0 = address to store $a1 = length limit	Characters are stored
Sbrk	9	$a0 = amount	$v0 holds address
Exit	10	none	Ends the program

The print_string service expects the address to start a null-terminated character string. The directive **.asciiz** creates a null-terminated character string.

The read_int, read_float and read_double services read an entire line of input up to and including the newline character.

The read_string service has the same semantics as the UNIX library routine fgets. It reads up to n-1 characters into a buffer and terminates the string with a null character. If fewer than n-1 characters are in the current line, it reads up to and including the newline and terminates the string with a null character.

The sbrk service returns the address to a block of memory containing n additional bytes. This would be used for dynamic memory allocation.

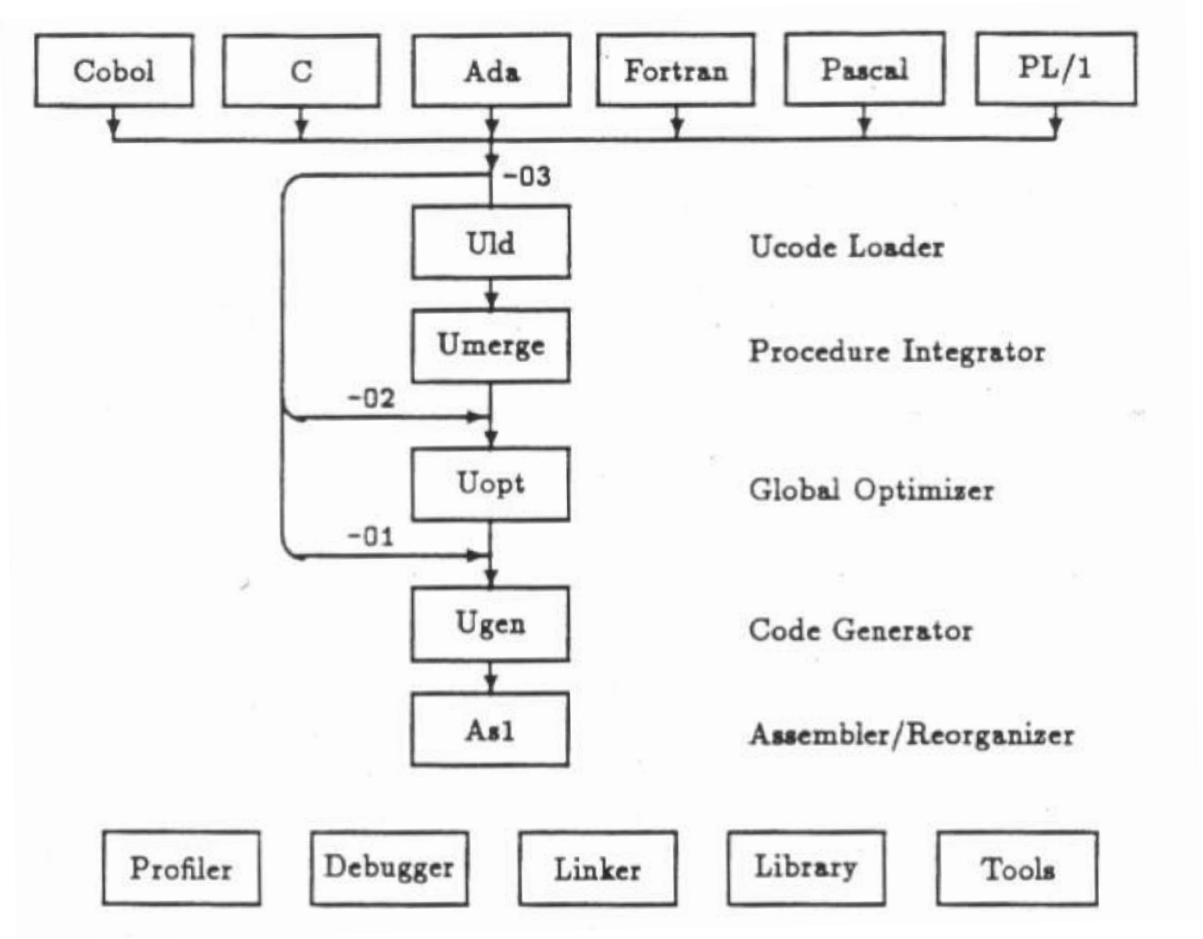
The exit service stops a program from running.

```
#    PROGRAM TO ADD TWO NUMBERS
#    Text segment
#    (all programs start with the next 3 lines)
        .text           # directive identifying the start of
                   # instructions
        .globl  __start
__start:
# -------------  print prompt on "console" ------------------
        la      $a0, prompt  # address of prompt goes in
        li      $v0, 4       # service code for print string
        syscall

# -------------  read in the integer ------------------------
        li      $v0, 5          # service code
        syscall
        sw      $v0, Num1       # store what was entered

# ------------- read another -------------------------------
        li      $v0, 5          # service code
        syscall
        sw      $v0, Num2       # store what was entered
```

```
# ------ Perform the addition, $a0 := Num1 + Num2 -----------
        lw      $t0, Num1
        add     $a0, $t0, $v0

# ------ print the sum, it is in $a0 ------------------------
        li      $v0, 1          # print integer service call
        syscall

# ------ print a final string identifying the result, and
# ending with a new line ------------------------------------
        la      $a0, final
        li      $v0, 4
        syscall
        li      $v0, 10         # exit program service
        syscall

#------------------------------------------------------------
#       Data segment
#------------------------------------------------------------
        .data
Num1:   .word   0
Num2:   .word   0
prompt: .ascii  "Please type 2 integers, end each with the "
        .asciiz "Enter key:\n"
final:  .asciiz " is the sum.\n"

#------ end of file  ADDNUMS.ASM ----------------------------
```

4 Language and the machine

4.1 The compilation process



4.1.1 The steps of compilation

All the languages are translated to the intermediate representation, that is called the Unicode. Because every language has it’s own instructions they use, so it must be translated in a code that is understandable for the code generator.

Uld links separate compilation units into a single file before presenting them to the to the code generator.

Umerge phase is to reorder the procedures in the Ucode file according to a post order. The inline expansion function of Umerge selectively replaces calls by the bodies of the called procedures.

Uopt is going to help in the register allocation from the Ucode.

Ugen translates the Ucode to a MIPS assembler code.

Asl is going to translate the assembler code into machine code and produces the MIPS machine language object file

4.2 The optimization choices

The MIPS compiler has four optimization levels, specified with the options -00, -01, -02 and -03.

4.2.1 Choices

The default optimization is -01, this is minimal optimization and fast compilation. Under this option, the code generator and assembler perform local optimizations within basic blocks. The code generator performs branch and label optimization and the assembler performs architecture dependent pipeline scheduling. These optimizations are performed by default because they do not add substantially to the compilation time.

The -00 option specifies no optimization. This option follow the same path as -01 but disables transformations normally performed by the code generator and assembler. This -00 option has a sporadic use, and normally used to compare codes.

With the -02 option, the Uopt phase is added to the compilation to perform global optimization and register allocation within each procedure. the compilation time increases substantially because global data-flow analysis and graph-coloring algorithms are involved.

The last optimization is the -03 option, this is option supports an entire load module. When this option is specified, the Uld phase is added to merge separate compilation units into a single file at the Ucode level. This enables multi module programs to achieve the same degree of optimization as single module programs. The Umerge phase is to reorder the procedures in the Ucode file according to a post-order. After the Umerge phase, the resulting Ucode object is then sent to the normal back end optimization and compilation stream starting with Uopt. Under the -03 option, Uopt also performs interprocedural register allocation.

4.3 The intermediate language

Ucode is a stack oriented pseudo machine language. MIPS’s Ucode was derived from the version used at Stanford, with modifications to support new programming languages and MIPS specific optimizations. Using Ucode as the intermediate language, the system can provide uniform compilation support to the high level languages. Ucode is also the medium for extensive global optimizations, register allocation. The intermediate code contains sufficient details about the underlying processor organization so that full benefit can be derived from the optimizations done by the Ucode optimizer.

Ucode has the following features:

- A stack used for all expression evaluation.
- A read-only storage area where instructions and string constants are kept.
- Static storage areas where global variables, static variables, external variables and common variables are kept.
- A set of registers where data items van be kept for fast access. Registers are also used for passing parameters and returning function results, in accordance with the MIPS linkage conventions.
- Stack memory organized into stack frames for processing procedure invocations. A stack frame is pushed on the memory stack whenever a procedure is activated.
- A heap for dynamic allocation of data objects at program execution time. Objects in the heap are accessible only through pointers.

4.4 The assembly process

An assembler is a program that translates each instruction to its binary machine code equivalent.
Translation takes 2 steps:

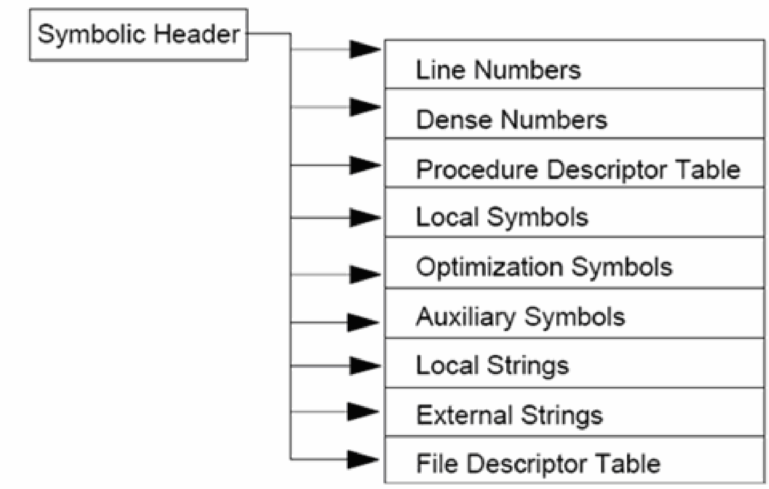
1. Associate memory locations with labels.
2. Translate each statement to the machine code.

So it works with a 2-pass assembler.

4.4.1 Associate memory locations with labels

Record the name and position of each label in the symbol table. To determine the position, the assembler must determine how many words each instruction or data declaration occupies.
The symbol table is often a hash table.

4.4.2 Symbol table



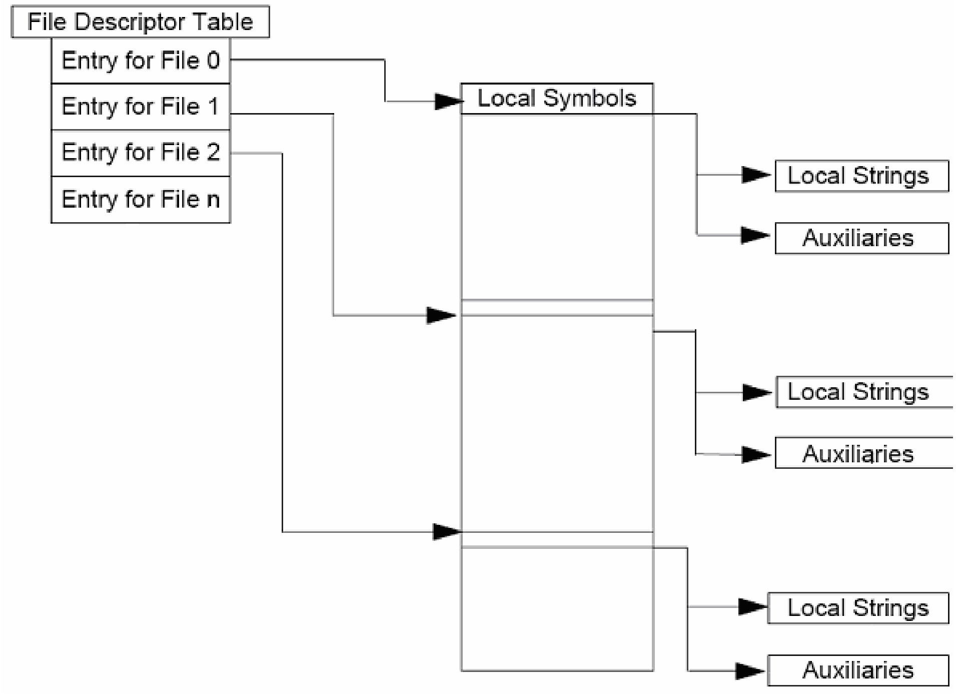
Symbolic Header: The Symbolic Header contains the sizes and locations of the sub tables that make up the Symbol Table.

Line Numbers: The assembler creates the Line Number table. It creates an entry for every instruction. Internally, the information is stored in an encoded form.

Dense Numbers: The Dense Number table is an array of pairs. An index into this table is called a dense number. Each pair consists of a file table index (ifd) and an index (isym) into Local Symbols. The table facilitates symbol look-up for the assembler, optimizer, and code generator by allowing direct table access rather than hashing.

Procedure Descriptor Table: The Procedure Descriptor table contains register and frame information, and offsets into other tables that provide detailed information on the procedure. The front-end creates the table and links it to the Local Symbols table. The assembler enters information on registers and frames. The debugger uses the entries in determining the line numbers for procedures and frame information for stack traces.

Local Symbols: The Local Symbols table contains descriptions of program variables, types, and structures, which the debugger uses to locate and interpret runtime values. The table gives the symbol type, storage class, and offsets into other tables that further define the symbol. A unique Local Symbols table exists for every source and include file; the compiler locates the table through an offset from the file descriptor entry that exists for every file. The entries in Local Symbols can reference related information in the Local Strings and Auxiliary Symbols sub tables.



Optimization Symbols: To be defined at a future date.

Auxiliary Symbols: The Auxiliary Symbols tables contain data type information specific to one language. Each entry is linked to an entry in Local Symbols. The entry in Local Symbols can have multiple, contiguous entries. The format of an auxiliary entry depends on the symbol type and storage class.

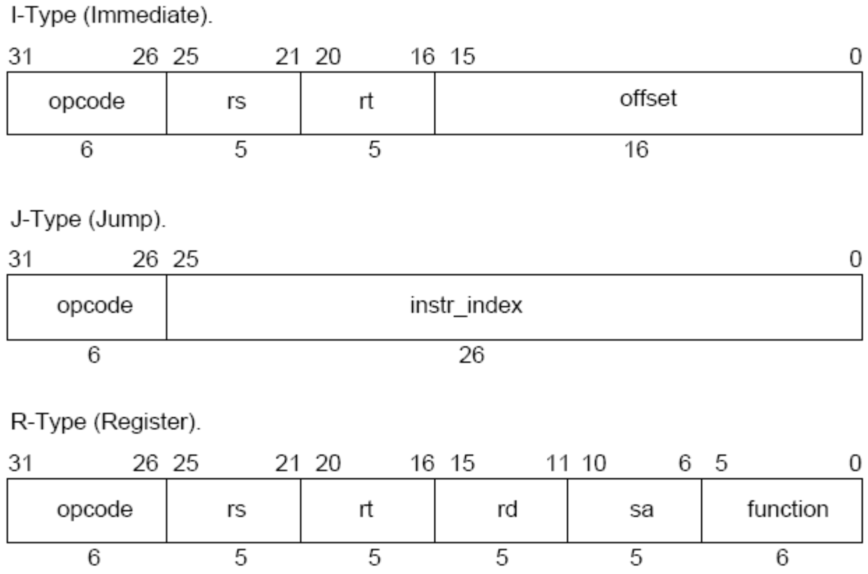
Local Strings: The Local Strings sub tables contain the names of local symbols.

External Strings: The External Strings table contains the names of external symbols.

File Descriptor: The File Descriptor table contains one entry each for each source file and each of its include files. The entry is composed of pointers to a group of sub tables related to the file.

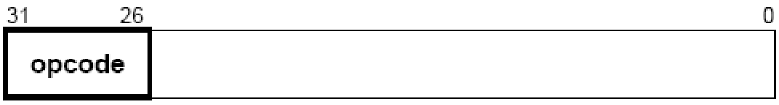
4.4.3 Translation to machine code

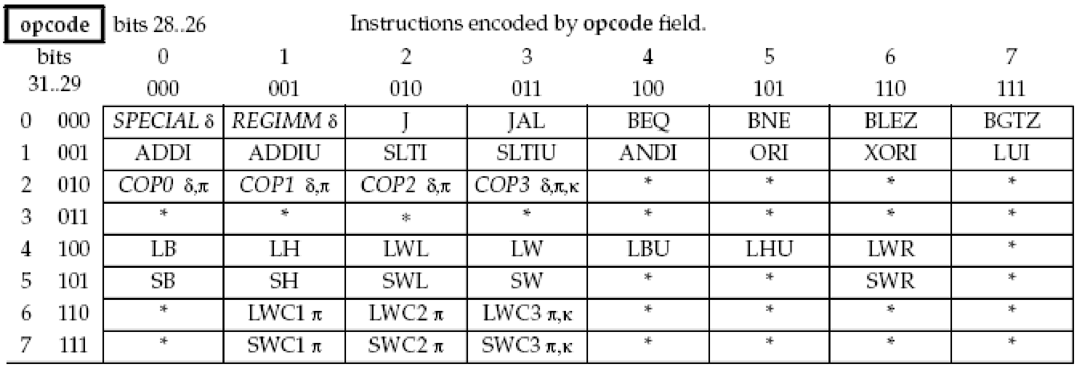
Machine code format:



Opcode	6-bit primary operation code
Rd	5-bit destination register specifier
Rs	5-bit source register specifier
Immediate	16-bit signed immediate used for: logical operands, arithmetic signed operands, load/store address byte offsets, PC-relative branch signed instruction displacement
Instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address.
Sa	5-bit shift amount
Function	6-bit function field used to specify functions within the primary operation code value SPECIAL.

Opcode





opcode bits 28..26 Instructions encoded by **opcode** field.

bits 31..29		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ,π	*COP1* δ,π	*COP2* δ,π	*COP3* δ,π,κ	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	101	SB	SH	SWL	SW	*	*	SWR	*
6	110	*	LWC1 π	LWC2 π	LWC3 π,κ	*	*	*	*
7	111	*	SWC1 π	SWC2 π	SWC3 π,κ	*	*	*	*

4.4.4 A Little example

Machine code generation from a simple instruction:

```
addi $8, $20, 15
```

- Addi is the opcode
- $8 is rt
- $20 is rs
- 15 is the immediate

Opcode is 6 bits, addi is defined to be 001000
Rs is 5 bits, encoding of 20, 10100
Rt is 5 bits, encoding of 8, 01000

The 32-bit instruction for addi $8, $20, 15 is:

```
     001000 10100 01000 0000000000001111
```

4.5 Linking and loading

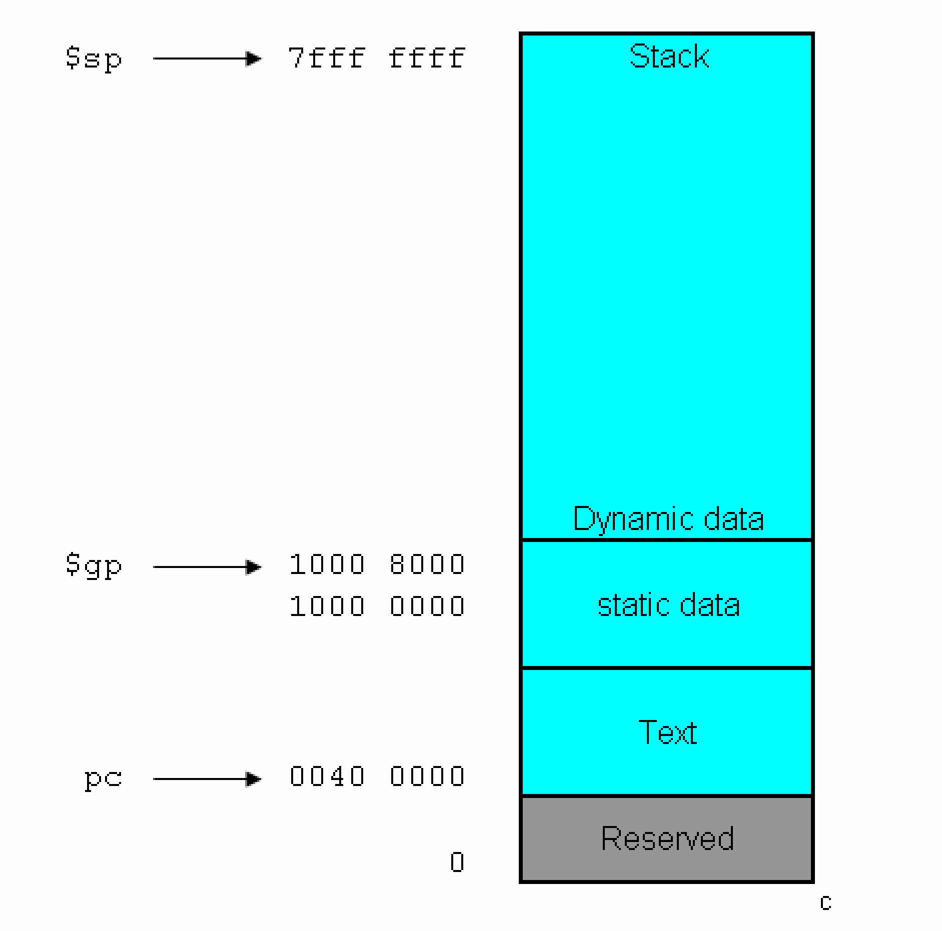
4.5.1 Linker

What we have presented so far suggests that a single change to one line of one procedure requires compiling and assembling the whole program. Complete retranslation is a terrible waste of computing resources. This repetition is particularly wasteful for standard library routines because programmers would be compiling and assembling routines that by definition almost never change. An alternative is to compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure. This alternative requires a new systems program, called a link editor or linker, that takes all the independently assembled machine language programs and “stitches them together.

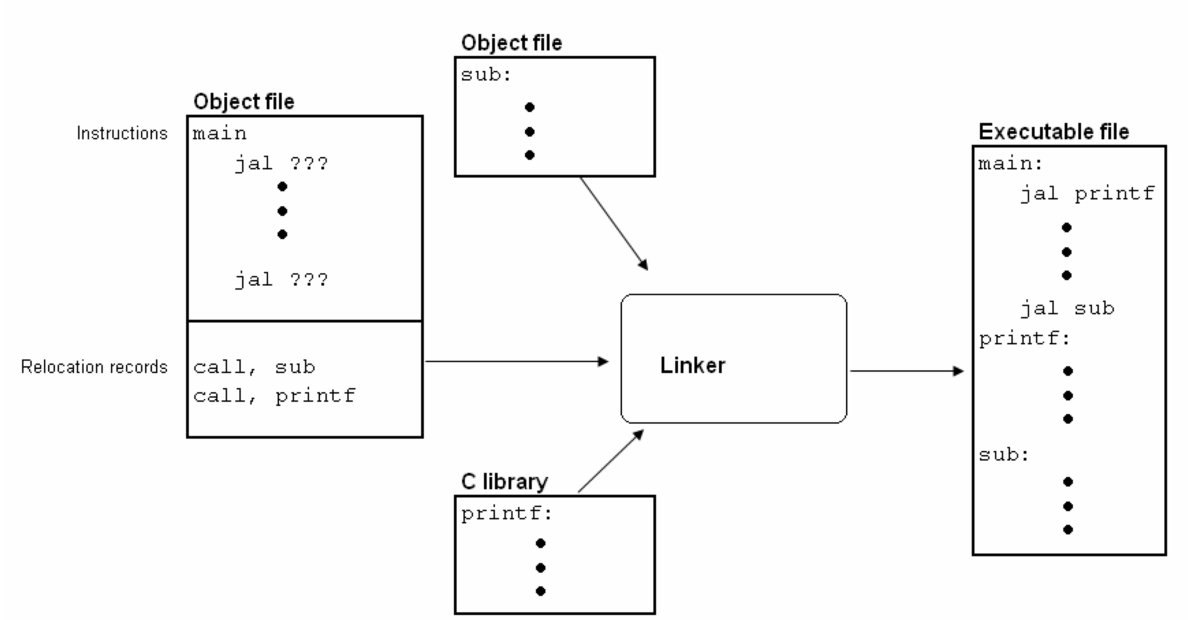
There are three steps for the linker:
Place code and data modules symbolically in memory.
Determine the addresses of data and instruction labels.
Patch both the internal and external references.

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions, jump instructions, and data addresses, so the job of this program is much like that of an editor. It finds the old addresses and replaces them with the new addresses. Editing is the origin of the name “link editor”, or linker for short. The reason a linker makes sense is that it is much faster to patch code than it is to recompile and reassemble.

If all external references are resolved, the linker next determines the memory locations each module will occupy. The figure below shows the MIPS convention for allocation of program and data to memory. Since the files were assembled in isolation, the assembler could not know where a module's instructions and data will be placed relative to other modules. When the linker places a module in memory, all absolute references, that is, memory addresses that are not relative to a register, must be relocated to reflect its true location.



The linker produces an executable file that can be run on a computer. Typically, this file has the same format as an object file, except that it contains no unresolved references, relocation information, symbol table, or debugging information. It is possible to have partially linked files, such as library routines, which still have unresolved addresses and hence result in object files.



4.5.2 Loader

A program that links without an error can be run. Before being run, the program resides in a file on secondary storage, such as a disk. On Unix systems, the operating system kernel brings a program into memory and starts it running. To start a program, the operating system performs the following steps:
Reads the executable file header to determine size of the text and data segments.
Creates an address space large enough for the text and data
Creates an address space large enough for the text and data.
Copies the parameters (if any) to the main program onto the stack.
Initializes the machine registers and sets the stack pointer to the first free location
Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an `exit` system call.

4.6 A programming example

C code

```
#include <stdio.h>
int main (int argc, char *argv[])
{
     int I;
     int sum = 0;

     for (I=0; I<=100; I++) sum += I * I;
     printf (“The sum 0..100=%d\n”,sum);
}
```

Compile this into a Ucode language with the compiler

```
.text
.align    2
.globl    main
.ent main 2
main:
     subu $sp, 32
     sw   $31, 20($sp)
     sd   $4, 32($sp)
     sw   $0, 24($sp)
     sw   $0, 28($sp)

loop:
     lw   $14, 28($sp)
     mul  $15, $14, $14
     lw   $24, 24($sp)
     addu $25, $24, $15
     sw   $8, 28($sp)
     ble  $8, 100, loop
     la   $4, str
     lw   $5, 24($sp)
     jal  printf
     move $2, $0
     lw   $31, 20($sp)
     addu $sp, 32
     j    $31
     .end main

     .data
     .align    0
str:
     .asciiz “The sum 0..100=%d\n”
```

Then you need to translate that in assembly language with the labels.

```
addiu     sp, sp,-32
sw        ra, 20(sp)
sw        a0, 32(sp)
sw        a1, 36(sp)
sw        zero, 24(sp)
sw        zero, 28(sp)
lw        t6, 28(sp)
lw        t8, 24(sp)
multu          t6, t6
addiu          t0, t6, 1
slti      at, t0, 101
sw        t0, 28(sp)
```

```
mflo      t7
addu      t9, t8, t7
bne       at, zero, -9
sw        t9, 24(sp)
lui       a0,4096
lw        a1, 24(sp)
jal       1048812
addiu          a0, a0, 1072
lw        ra, 20(sp)
addiu          sp, sp, 32
jr        ra
move      v0, zero
```

This you can translate in the machine language.

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

4.7 Macros

Macros are a pattern-matching and replacement facility that provide a simple mechanism to name a frequently used sequence of instructions. Instead of repeatedly typing the same instructions every time they are used, a programmer invokes the macro and the assembler replaces the macro call with the corresponding sequence of instructions. Macros, like subroutines, permit a programmer to create and name a new abstraction for a common operation. Unlike subroutines, however, macros do not cause a subroutine call and return when the program runs since a macro call is replaced by the macro’s body when the program is assembled. After this replacement, the resulting assembly is indistinguishable from the equivalent program written without macros.

4.7.1 Example

As an example, suppose that a programmer needs to print many numbers. The library routine printf accepts a format strings and one or more values to print as its arguments. A programmer could print the integer in register $7 with the following instructions:

```
          .data
Int_str:  .asciiz “%d”
          .text
          la   $a0, int_str   # Load string address into first
                              # argument
          mov  $al, $7        # Load value into second arg
          jal  printf         # Call the printf routine
```

The .data directive tells the assembler to store the string in he program’s data segment, and the .text directive tells the assembler to store the instructions in its text segment.

However, printing many numbers in this fashion is tedious and produces a verbose program that is difficult to understand. An alternative is to introduce a macro, print_int, to print an integer:

```
          .data
Int_str:  .asciiz “%d”
          .text
          .macro print_int($arg)
          la   $a0, int_str   # Load string address into first
                              # argument
          mov  $al, $arg      # Load value into second arg
          jal  printf         # Call the printf routine
          .end_macro
Print_int($7)
```

The macro has a formal parameter, $arg, that names the argument to the macro. When the macro is expanded, the argument from a call is substituted for the formal parameter throughout the macro’s body. Then the assembler replaces the call with the macro’s newly expanded body. In the first call on print_int, the argument is $7, so the macro expands to the code:

```
la   $a0, int_str
mov  $a1, $7
jal  printf
```

In a second call on print_int, say print_int($t0), the argument is $t0, so the macro expands to:

```
la   $a0, int_str
mov  $a1, $t0
jal  printf
```

4.7.2 Vector processor

The vector processor will operate as a coprocessor unit of the MIPS instruction set architecture. It will extend the ISA to execute memory and the computer instructions in vector mode. The instructions are integer arithmetic, logical, shift and floating point instructions. The vector processor has some features.

- 32 bit and 64 bit operation
- Control registers
- Load/store instructions
- Single chip implementation

4.7.3 Vector registers

In the MIPS architecture are two types of vector registers.

- General purpose vector register
- Vector control register

General purpose vector registers

The general purpose vector register can hold three types of data formats.

- Integer values
- Single precision
- Double precision

There are 32 vector registers so every register can hold a different data. But when the register is accessed they have all the same data formats.

The data can be copied to an different type, with this you can go from a integer to a double precision. The instructions to do that are **vtint** and **vtfp.**
You can do the same when you want to copy data from a register to a vector register. The instructions for this are **mfc2** and **mtc2.**

Vector control registers

There are 32 control registers, these registers are going to control parts of the instructions.

VLR: vector length register specifies the number of operations to execute a vector instruction.

- When the data from the VLR is zero then there is nothing to be executed.
- If there is a vector instruction want to be executed when the VLR data is greater than the maximum vector length, then the vector instruction data is raised.

- When you want to read or write to the VLR it will not affect the vector instructions in progress.

VCOND: the vector condition registers hold the results of the most recently executed vextor instructions. There is one bit corresponding to each matching result with the least significant bit as the representation when the vector element is zero.

- When you want to read the VCOND everything will be locked and it will return his value.
- If you want to write to VCOND it will also lock everything and you can write, it will not be affected by vector instructions that may still be executing.

VCSR: this is the vector status control register it is used to control and monitoring the floating point data.

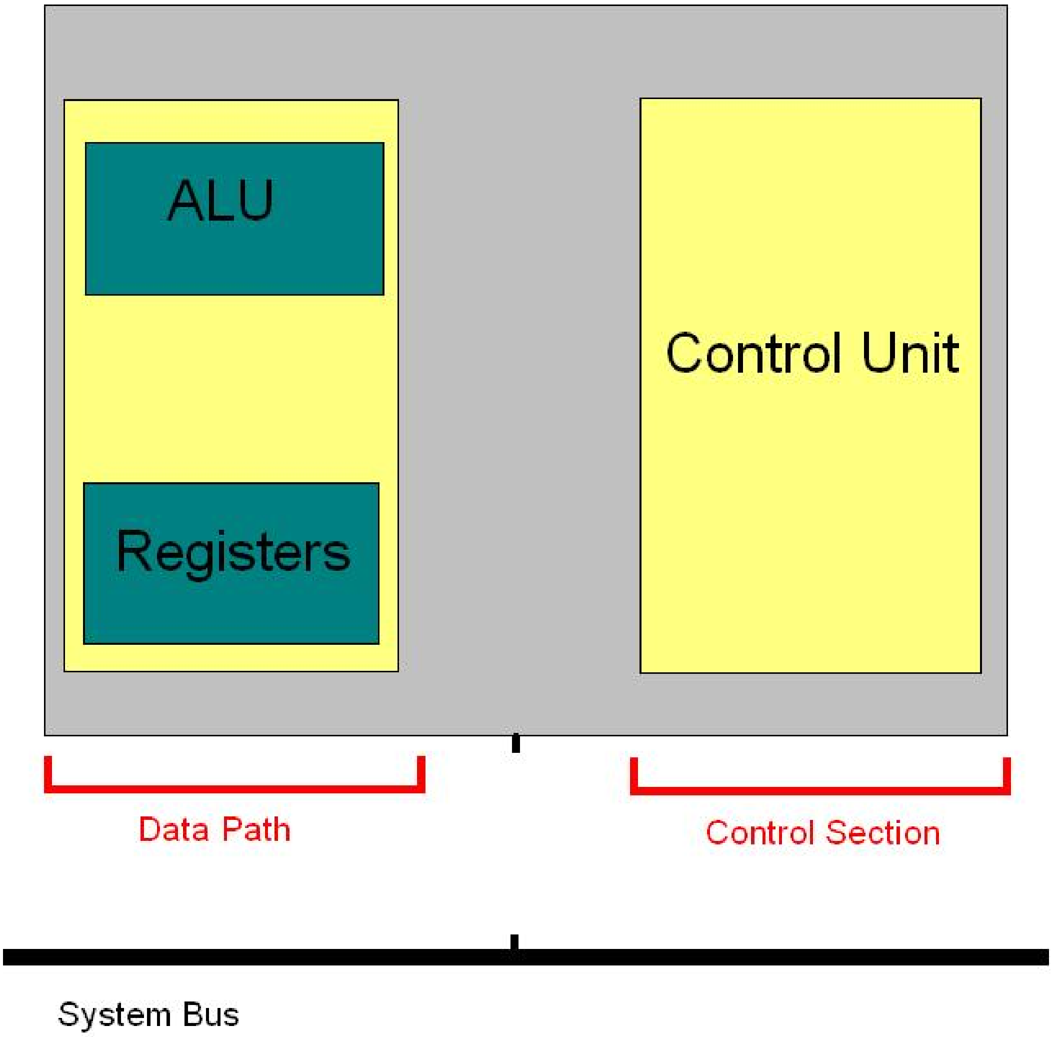
5 Datapath and control

5.1 The Basics of the Microarchitecture

The functionality of the microarchitecture centers around the fetch-execute cycle. The steps done by this cycle are:

- Fetch next instruction from the memory
- Decode the Opcode
- Read the operands
- Execute the instruction and store the results
- Start all over

It's the microarchitecture that makes these steps happen.
There are two main sections in the microarchitecture, the Data Section or Data Path and the Control Section. The data section contains registers and an ALU as shown on the picture.

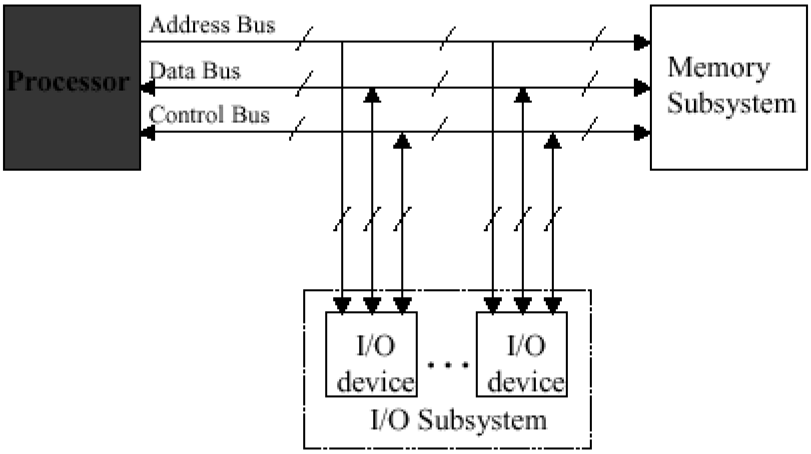


5.2 A Microarchitecture for the MIPS

5.2.1 The Central Processor

Basics

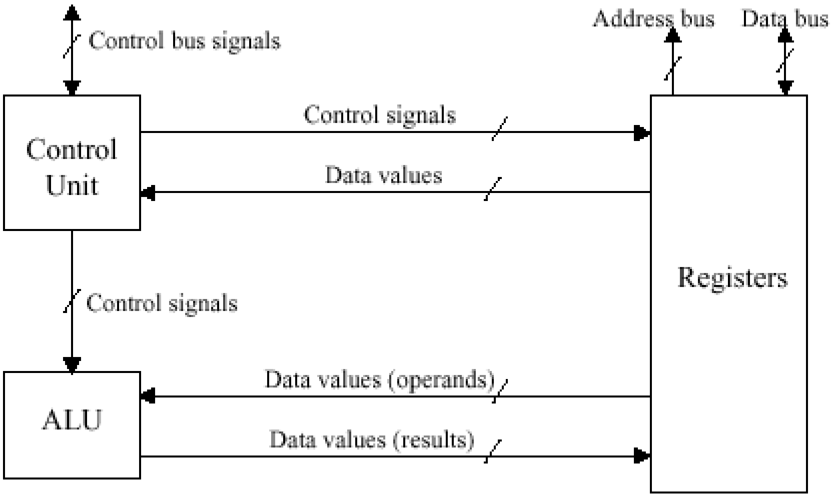
In the figure below, the typical organization of a modern von Neumann processor is illustrated. Note that the CPU, memory subsystem, and I/O subsystem are connected by address, data, and control buses. The fact that these are parallel buses is denoted by the slash through each line that signifies a bus.



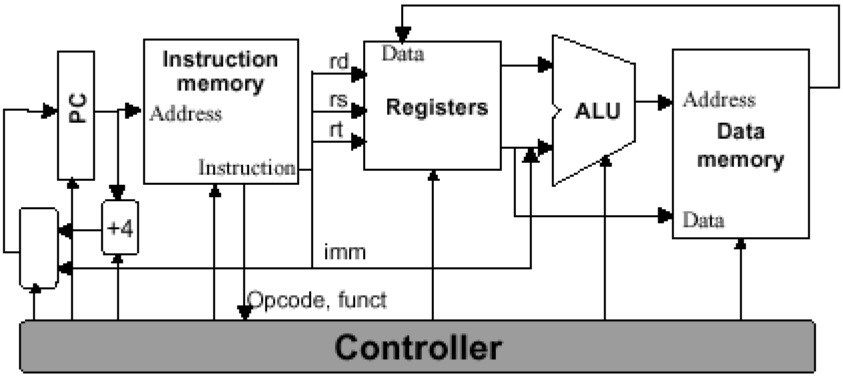
This machine contains the following components:

- *Processor (CPU)* is the active part of the computer, which does all the work of data manipulation and decision making.
- *Datapath* is the hardware that performs all the required operations, for example, ALU, registers, and internal buses.
- *Control* is the hardware that tells the datapath what to do, in terms of switching, operation selection, data movement between ALU components, etc.

The processor represented by the shaded block in the previous figure is organized as shown in the one following. Observe that the ALU performs I/O on data stored in the register file, while the Control Unit sends (receives) control signals (resp. data) in conjunction with the register file.



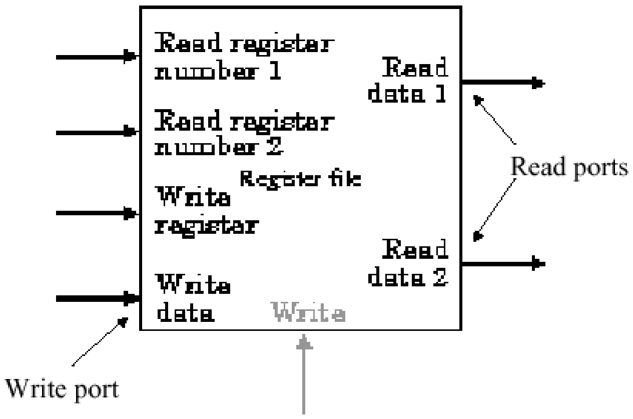
In MIPS, the ISA determines many aspects of the processor implementation. For example, implementational strategies and goals affect clock rate and cycles per instruction (CPI). These implementational constraints cause parameters of the components in the next figure to be modified throughout the design process.

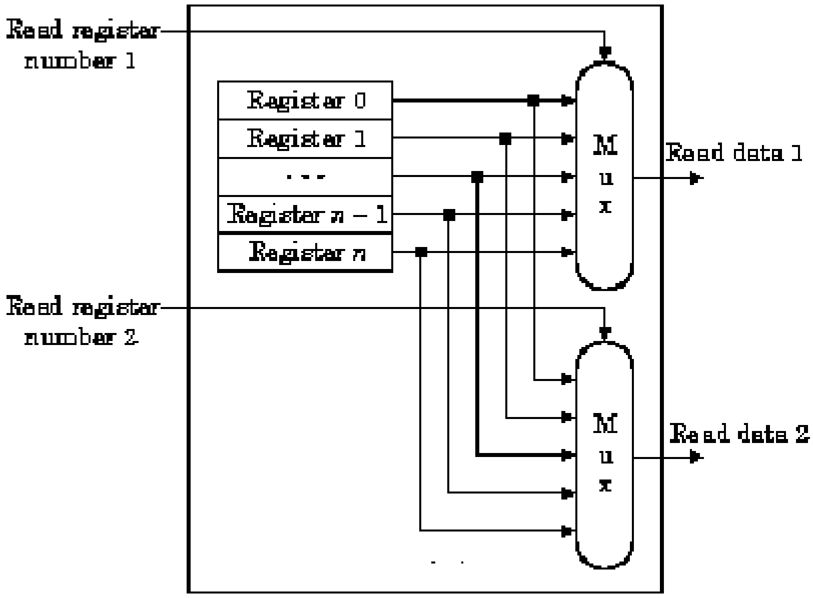


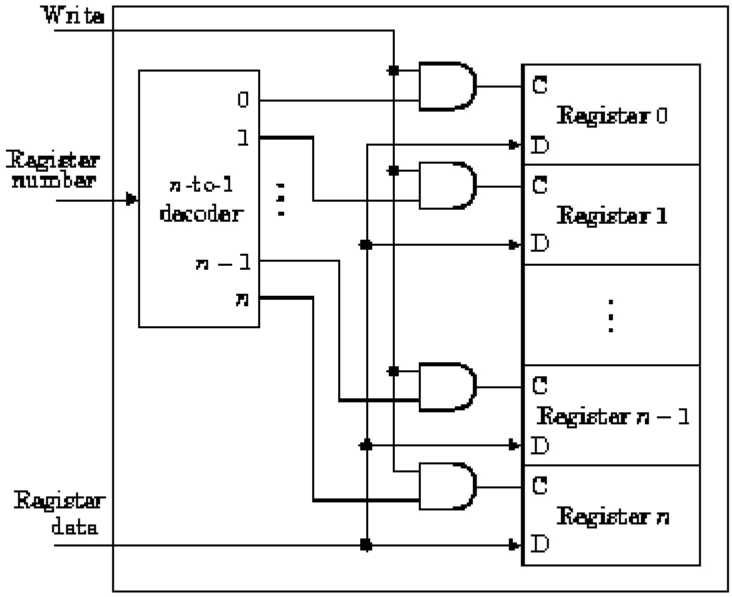
Such implementational concerns are reflected in the use of logic elements and clocking strategies. For example, with combinational elements such as adders, multiplexers, or shifters, outputs depend only on current inputs. However, sequential elements such as memory and registers contain state information, and their output thus depends on their inputs (data values and clock) as well as on the stored state. The clock determines the order of events within a gate, and defines when signals can be converted to data to be read or written to processor components (e.g., registers or memory).

Register File

The register file (RF) is a hardware device that has two read ports and one write port (corresponding to the two inputs and one output of the ALU). The RF and the ALU together contains the two elements required to compute MIPS R-format ALU instructions. The RF is comprised of a set of registers that can be read or written by supplying a register number to be accessed, as well (in the case of write operations) as a write authorization bit. A block diagram of the RF is shown below.


Read register number 1
Read register number 2
Register file
Write register
Write data
Write
Read data 1
Read data 2
Read ports
Write port


Read register number 1
Register 0
Register 1
. . .
Register n – 1
Register n
Read register number 2
M u x
Read data 1
M u x
Read data 2


Write
Register number
n-to-1 decoder
0
1
n – 1
n
Register data
C
Register 0
D
C
Register 1
D
C
Register n – 1
D
C
Register n
D

Since reading of a register-stored value does not change the state of the register, no "safety mechanism" is needed to prevent inadvertent overwriting of stored data, and we need only supply the register number to obtain the data stored in that register. However, when writing to a register, we need a register number, an authorization bit, for safety (because the previous contents of the register selected for writing are overwritten by the write operation), and a clock pulse that controls writing of data into the register.

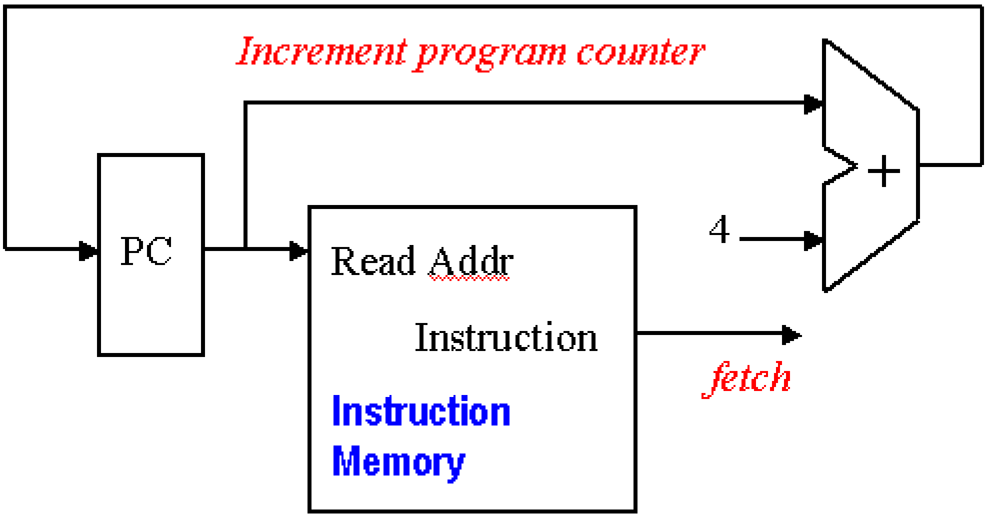
In this discussion and throughout this section, we will assume that the register file is structured as shown in the first scheme of the figure above. We further assume that each register is constructed from a linear array of D flip-flops, where each flip-flop has a clock (C) and data (D) input. The read ports can be implemented using two multiplexers, each having $\log_2 N$ control lines, where N is the number of bits in each register of the RF. Note that data in the second scheme from all N = 32 registers flows out to the output muxes, and the data stream from the register to be read is selected using the mux's five control lines

The last scheme shows an implementation of the RF write port. Here, the *write enable* signal is a clock pulse that activates the edge-triggered D flip-flops which comprise each register (shown as a rectangle with clock (C) and data (D) inputs). The register number is input to an N-to-2^N decoder, and acts as the control signal to switch the data stream input into the Register Data input. The actual data switching is done by *and*-ing the data stream with the decoder output: only the *and* gate that has a unitary (one-valued) decoder output will pass the data into the selected register (because 1 *and* *x* = *x*).

5.2.2 Datapath Design and Implementation

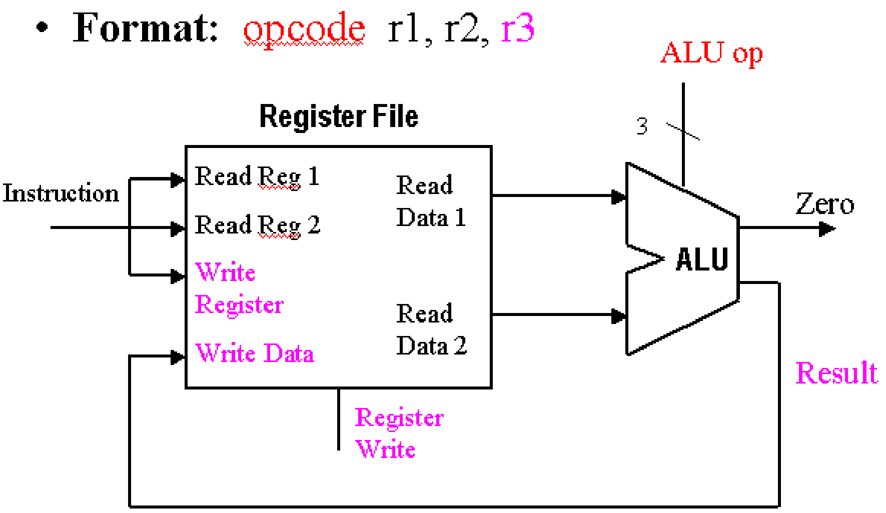
The datapath is the power of a processor, since it implements the fetch-decode-execute cycle. The general discipline for datapath design is:
to determine the instruction classes and formats in the ISA
to design datapath components and interconnections for each instruction class or format to compose the datapath segments designed in step 2 to yield a composite datapath.

Simple datapath components include *memory* (stores the current instruction), *PC* or program counter (stores the address of current instruction), and *ALU* (executes current instruction). The interconnection of these simple components to form a basic datapath is illustrated below. Note that the register file is written to by the output of the ALU.



R-format Datapath

Implementation of the datapath for R-format instructions is fairly straightforward. The register file and the ALU are all that is required. The ALU accepts its input from the DataRead ports of the register file, and the register file is written to by the ALUresult output of the ALU, in combination with the RegWrite signal.

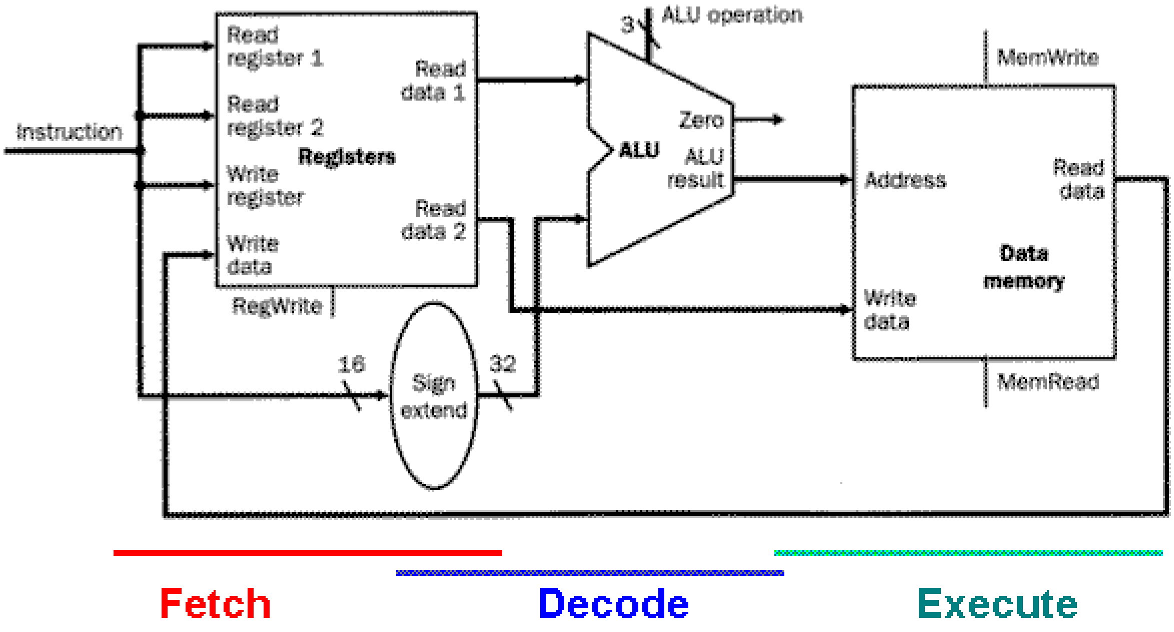


Load/Store Datapath

The load/store datapath uses instructions such as `lw $t1, offset($t2)`, where *offset* denotes a memory address offset applied to the base address in register `$t2`. The `lw` instruction reads from memory and writes into register `$t1`. The `sw` instruction reads from register `$t1` and writes into memory. In order to compute the memory address, the MIPS ISA specification says that we have to sign-extend the 16-bit offset to a 32-bit signed value.

The load/store datapath is illustrated in next figure, and performs the following actions in the order given:

Register Access takes input from the register file, to implement the instruction, data, or address *fetch* step of the fetch-decode-execute cycle.
Memory Address Calculation decodes the base address and offset, combining them to produce the actual memory address. This step uses the sign extender and ALU.
Read/Write from Memory takes data or instructions from the data memory, and implements the first part of the *execute* step of the fetch/decode/execute cycle.
Write into Register File puts data or instructions into the data memory, implementing the second part of the *execute* step of the fetch/decode/execute cycle.



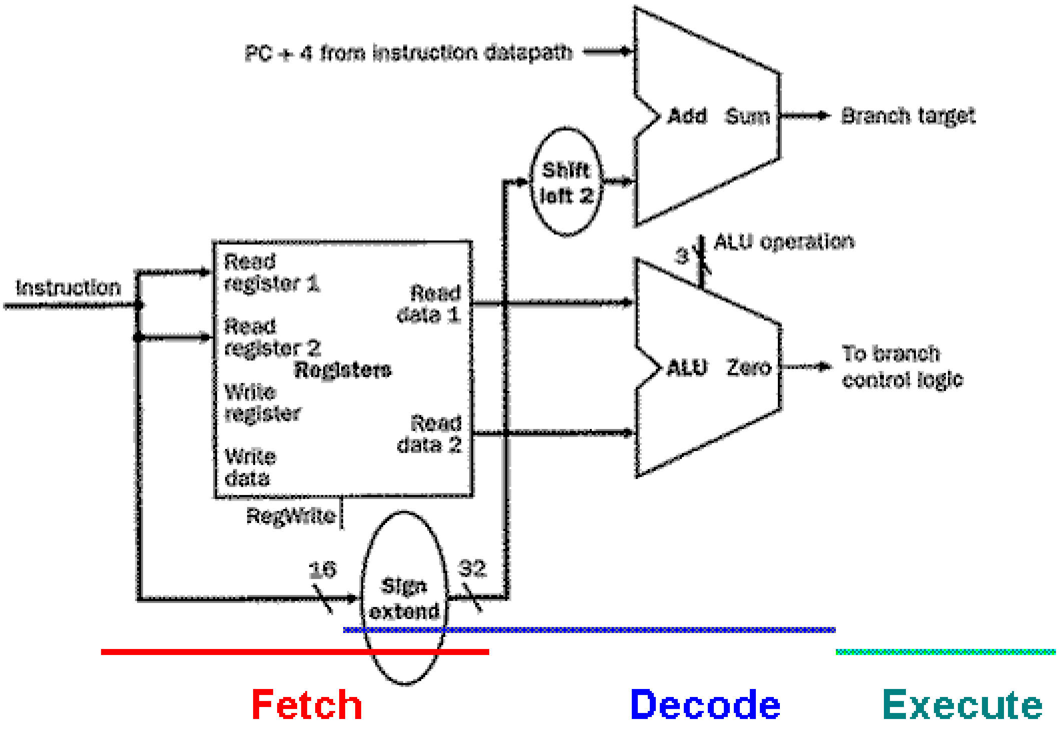
The load/store datapath takes operand #1 (the base address) from the register file, and sign-extends the offset, which is obtained from the instruction input to the register file. The sign-extended offset and the base address are combined by the ALU to yield the memory address, which is input to the Address port of the data memory. The MemRead signal is then activated, and the output data obtained from the ReadData port of the data memory is then written back to the Register File using its WriteData port, with RegWrite asserted.

Branch/Jump Datapath

The branch datapath (jump is an unconditional branch) uses instructions such as `beq $t1, $t2, offset`, where *offset* is a 16-bit offset for computing the branch target address via PC-relative addressing. The `beq` instruction reads from registers `$t1` and `$t2`, then compares the data obtained from these registers to see if they are equal. If equal, the branch is taken. Otherwise, the branch is not taken.

By *taking the branch*, the ISA specification means that the ALU adds a sign-extended offset to the program counter (PC). The offset is shifted left 2 bits to allow for word alignment (since 2^2 = 4, and words are comprised of 4 bytes). Thus, to jump to the target address, the lower 26 bits of the PC are replaced with the lower 26 bits of the instruction shifted left 2 bits.

The branch instruction datapath is illustrated in next figure, and performs the following actions in the order given:
Register Access takes input from the register file, to implement the *instruction fetch* or *data fetch* step of the fetch-decode-execute cycle.
Calculate Branch Target - Concurrent with ALU #1's evaluation of the branch condition, ALU #2 calculates the branch target address, to be ready for the branch if it is taken. This completes the *decode* step of the fetch-decode-execute cycle.
Evaluate Branch Condition and Jump to BTA or PC+4 uses ALU #1 in the sheme bellow, to determine whether or not the branch should be taken. Jump to BTA or PC+4 uses control logic hardware to transfer control to the instruction referenced by the branch target address. This effectively changes the PC to the branch target address, and completes the *execute* step of the fetch-decode-execute cycle.



The branch datapath takes operand #1 (the offset) from the instruction input to the register file, then sign-extends the offset. The sign-extended offset and the program counter (incremented by 4 bytes to reference the next instruction after the branch instruction) are combined by ALU #1 to yield the branch target address. The operands for the branch condition to evaluate are concurrently obtained from the register file via the ReadData ports, and are input to ALU #2, which outputs a one or zero value to the branch control logic.

MIPS has the special feature of a *delayed branch*, that is, instruction I_b which follows the branch is always fetched, decoded, and prepared for execution. If the branch condition is false, a normal branch occurs. If the branch condition is true, then I_b is executed. One wonders why this extra work is performed - the answer is that delayed branch improves the efficiency of pipeline execution, as we shall see in Section 5.

Also, the use of branch-not-taken (where I_b is executed) is sometimes the common case.

5.2.3 Single-Cycle and Multicycle Datapaths

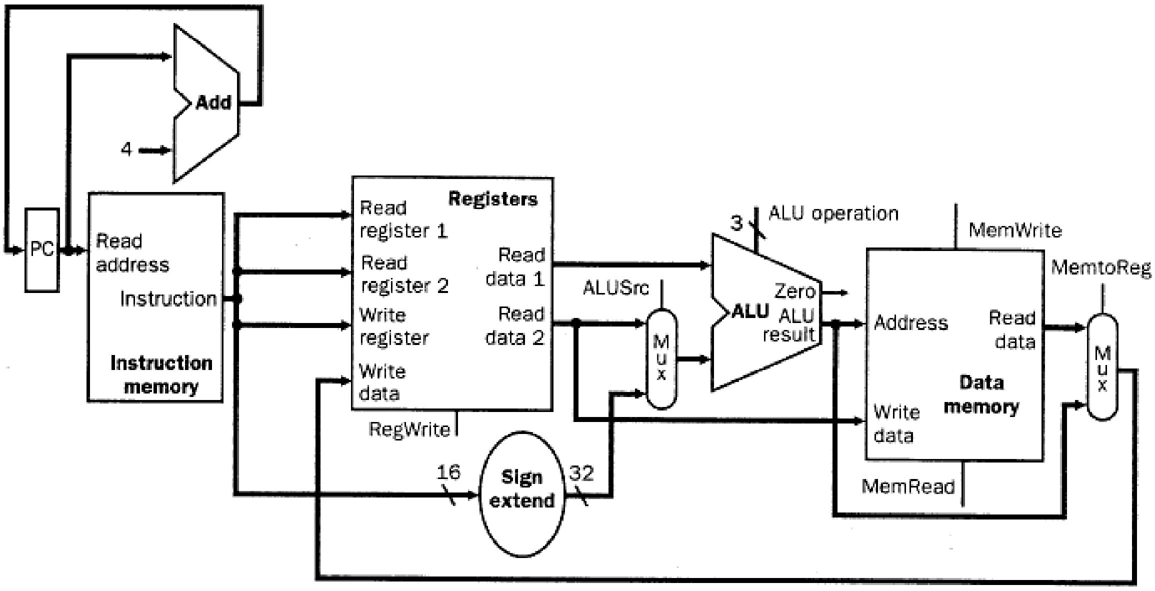
A single-cycle datapath executes in one cycle all instructions that the datapath is designed to implement. This clearly impacts CPI in a beneficial way, namely, CPI = 1 cycle for all instructions. In this section, we first examine the design discipline for implementing such a datapath. Then, we discover how the performance of a single-cycle datapath can be improved using a multi-cycle implementation.

Single Datapaths

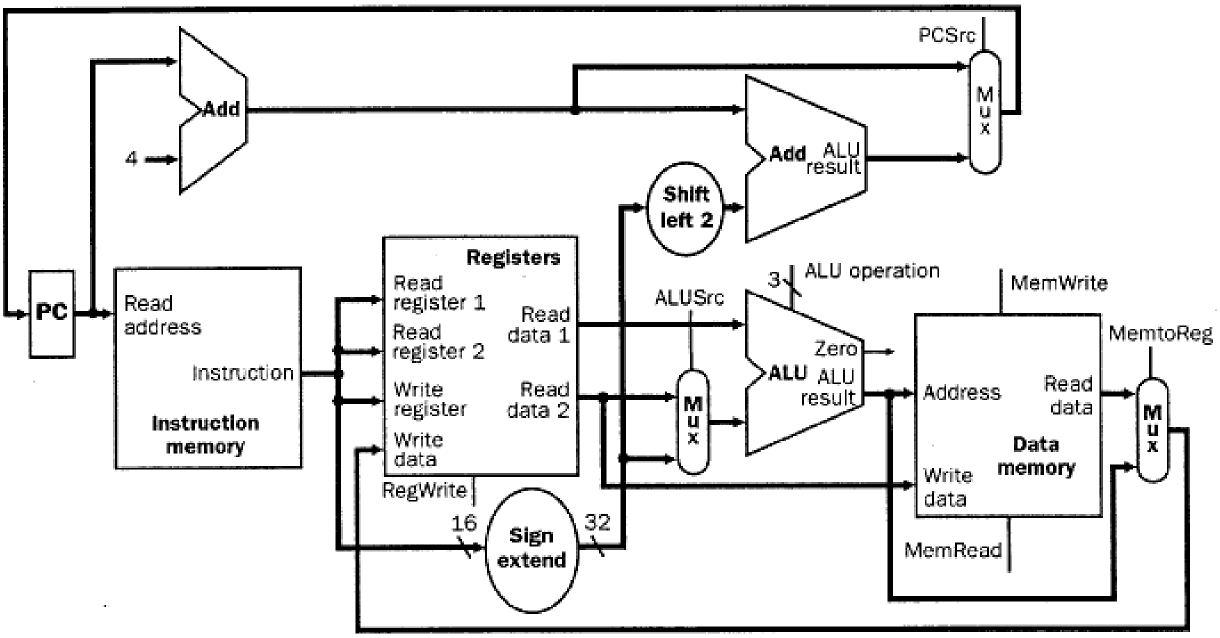
Let us begin by constructing a datapath with control structures. The simplest way to connect the datapath components is to have them all execute an instruction concurrently, in one cycle. As a result, no datapath component can be used more than once per cycle, which implies duplication of components. To make this type of design more efficient without sacrificing speed, we can share a datapath component by allowing the component to have multiple inputs and outputs selected by a multiplexer.

The key to efficient single-cycle datapath design is to find simularities among instruction types. For example, the R-format MIPS instruction and the load/store datapath have similar register file and ALU connections. However, the following differences can also be observed:
The second ALU input is a register (R-format instruction) or a signed-extended lower 16 bits of the instruction (e.g., a load/store offset).
The value written to the register file is obtained from the ALU (R-format instruction) or memory (load/store instruction).

These two datapath designs can be combined to include separate instruction and data memory, as shown below. The combination requires an adder and an ALU to respectively increment the PC and execute the R-format instruction.



Adding the branch datapath to the datapath produces the augmented datapath shown on next page. The branch instruction uses the main ALU to compare its operands and the adder computes the branch target address. Another multiplexer is required to select either the next instruction address (PC + 4) or the branch target address to be the new value for the PC.



ALU Control.

Given the above datapath, we next add the control unit. Control accepts inputs (called *control signals*) and generates:

- a write signal for each state element

- the control signals for each multiplexer
- the ALU control signal.

The ALU has three control signals

ALU Control Input	Function
000	and
001	or
010	add
110	sub
111	slt

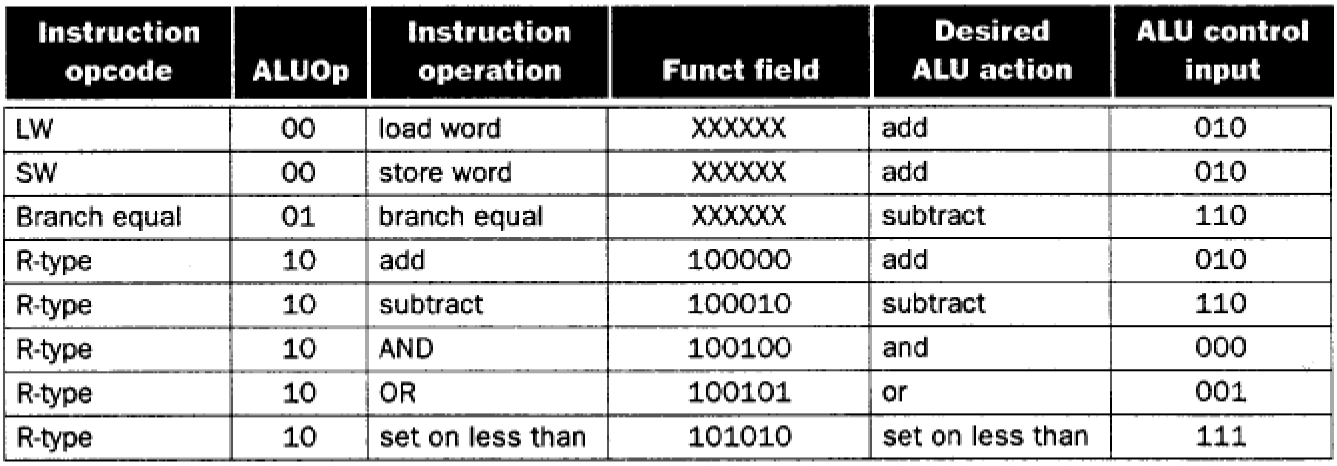
The ALU is used for all instruction classes, and *always* performs one of the five functions in the right-hand column of the table above. For branch instructions, the ALU performs a subtraction, whereas R-format instructions require one of the ALU functions. The ALU is controlled by two inputs:
the opcode from a MIPS instruction (six most significant bits
a two-bit control field (which Patterson and Hennesey call `ALUop`).
The ALUop signal denotes whether the operation should be one of the following:

ALUop Input	Operation
00	load/store
01	beq
10	determined by opcode

The output of the ALU control is one of the 3-bit control codes shown in the left-hand column of the first table. The second table shows how to set the ALU output based on the instruction opcode and the ALUop signals. Later, we will develop a circuit for generating the ALUop bits. We call this approach *multi-level decoding* -- main control generates ALUop bits, which are input to ALU control. The ALU control then generates the three-bit codes from the first table.

The advantage of a hierarchically partitioned or pipelined control scheme is realized in reduced hardware (several small control units are used instead of one large unit). This results in reduced hardware cost, and can in certain instances produce increased speed of control. Since the control unit is critical to datapath performance, this is an important step in the implementation process.

Recall that we need to map the two-bit ALUop field and the six-bit opcode to a three-bit ALU control code. Normally, this would require $2^{(2 + 6)}$ = 256 possible combinations, eventually expressed as entries in a truth table. However, only a few opcodes are to be implemented in the ALU designed herein. Also, the ALU is used only when ALUop = 10_2. Thus, we can use simple logic to implement the ALU control, as shown in terms of the following truth table.

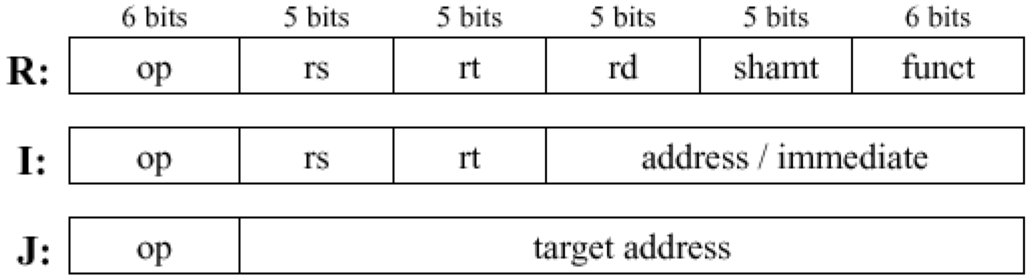


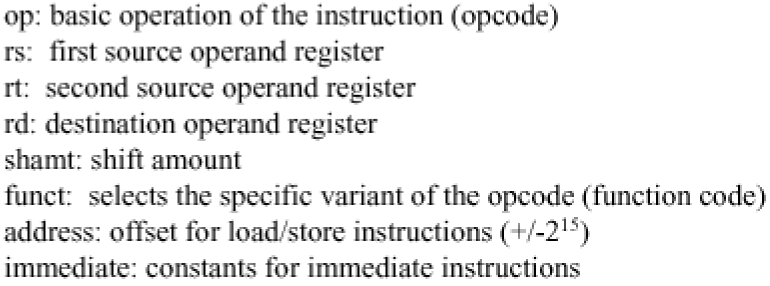
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

In this table, an "X" in the input column represents a "don't-care" value, which indicates that the output does not depend on the input at the i-th bit position.

Main Control Unit.

The first step in designing the main control unit is to identify the fields of each instruction and the required control lines to implement the datapath. Recalling the three MIPS instruction formats (R, I, and J), shown as follows:





op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions

Observe that the following always apply:

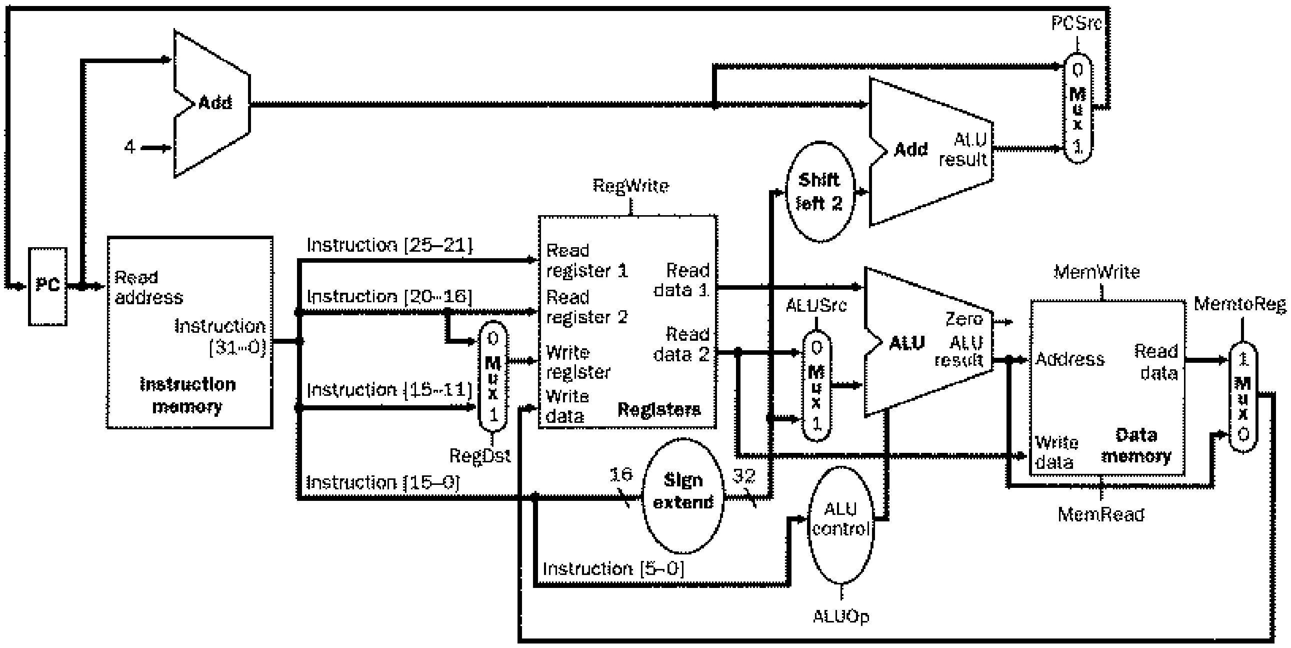
- **Bits 31-26:** *opcode* - always at this location
- **Bits 25-21 and 20-16:** *input register indices* - always at this location

Additionally, we have the following instruction-specific codes due to the regularity of the MIPS instruction format:

- **Bits 25-21:** *base register* for load/store instruction - always at this location

- **Bits 15-0:** *16-bit offset* for branch instruction - always at this location
- **Bits 15-11:** *destination register* for R-format instruction - always at this location
- **Bits 20-16:** *destination register* for load/store instruction - always at this location

Note that the different positions for the two destination registers implies a selector (i.e., a mux) to locate the appropriate field for each type of instruction. Given these contraints, we can add to the simple datapath thus far developed instruction labels and an extra multiplexer for the WriteReg input of the register file as shown in the figure on the next page

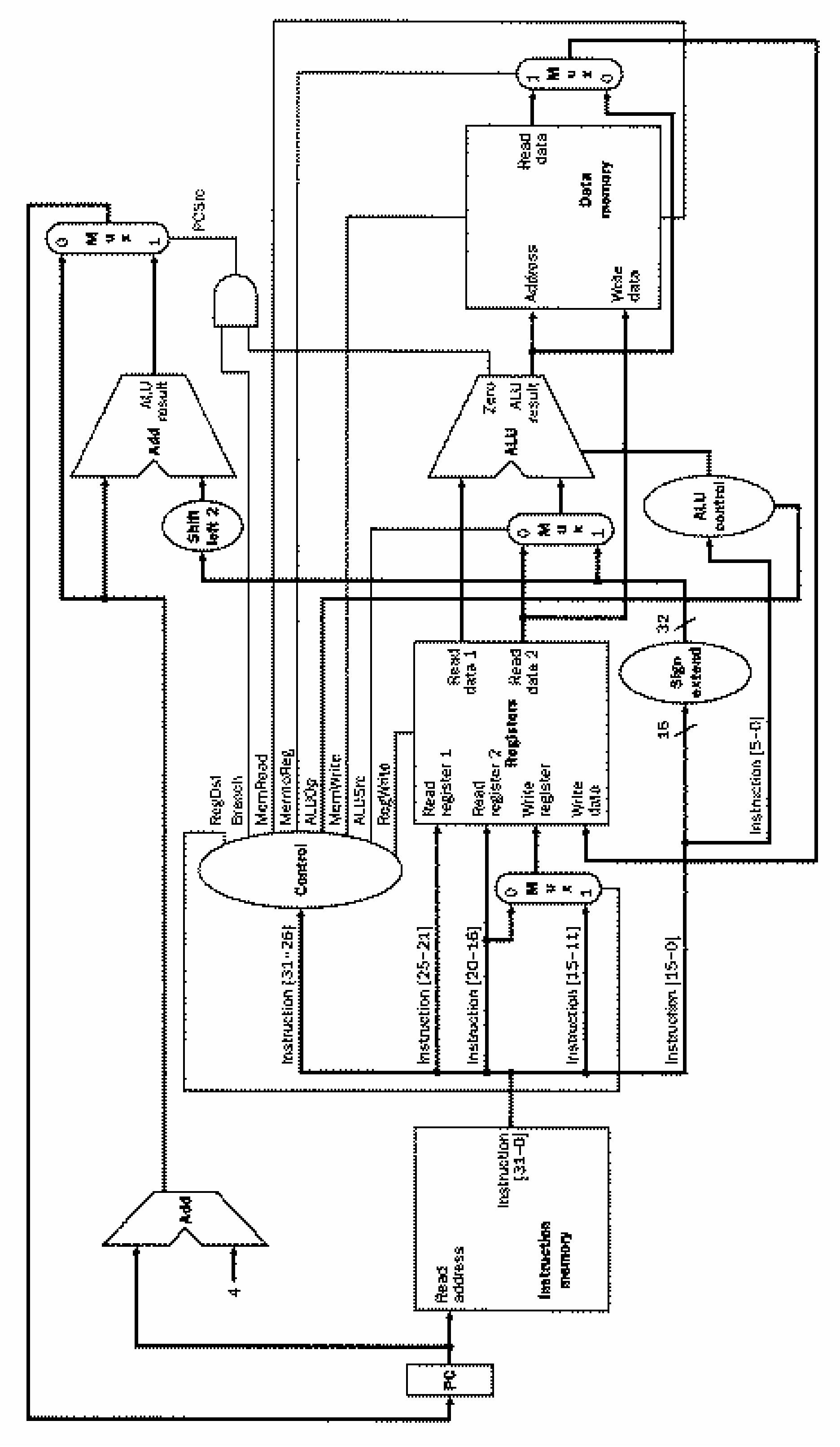


Here, we see the seven-bit control lines (six-bit opcode with one-bit WriteReg signal) together with the two-bit ALUop control signal, whose actions when asserted or deasserted are given as follows:

- RegDst
  - *Deasserted*: Register destination number for the Write register is taken from bits 20-16 (rt field) of the instruction
  - *Asserted*: Register destination number for the Write register is taken from bits 15-11 (rd field) of the instruction
- RegWrite
  - *Deasserted*: No action
  - *Asserted*: Register on the WriteRegister input is written with the value on the WriteData input
- ALUSrc

    - *Deasserted*: The second ALU operand is taken from the second register file output (ReadData 2)
    - *Asserted*: the second alu operand is the sign-extended, lower 16 bits of the instruction
- PCSrc
    - *Deasserted*: PC is overwritten by the output of the adder (PC + 4)
    - *Asserted*: PC overwritten by the branch target address
- MemRead
    - *Deasserted*: No action
    - *Asserted*: Data memory contents designated by address input are present at the ReadData output
- MemWrite
    - *Deasserted*: No action
    - *Asserted*: Data memory contents designated by address input are present at the WriteData input
- RegWrite
    - *Deasserted*: The value present at the WriteData input is output from the ALU
    - *Asserted*: The value present at the register WriteData input is taken from data memory

Given only the opcode, the control unit can thus set all the control signals except PCSrc, which is only set if the instruction is `beq` and the Zero output of the ALu used for comparison is *true*. PCSrc is generated by *and*-ing a *Branch* signal from the control unit with the Zero signal from the ALU. Thus, all control signals can be set based on the opcode bits. The resultant datapath and its signals are shown in detail on the next page.


PC
Read address
Instruction [31–0]
Instruction memory
4
Add
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
M u x
0
1
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign extend
32
Instruction [5–0]
ALU control
Shift left 2
Add
ALU result
PCSrc
M u x
0
1
M u x
0
1
ALU
Zero
ALU result
Address
Write data
Data memory
Read data
M u x
1
0

Datapath Operation

Recall that there are three MIPS instruction formats -- R, I, and J. Each instruction causes slightly different functionality to occur along the datapath, as follows.

R-format Instruction.

Execution of an R-format instruction (e.g., `add $t1, $t0, $t1`) using the developed datapath involves the following steps:
Fetch instruction from instruction memory and increment PC
Input registers (e.g., `$t0` and `$t1`) are read from the register file
ALU operates on data from register file using the *funct* field of the MIPS instruction (Bits 5-0) to help select the ALU operation
Result from ALU written into register file using bits 15-11 of instruction to select the destination register (e.g., `$t1`).

Note that this implementational sequence is actually combinational, becuase of the single-cycle assumption. Since the datapath operates within one clock cycle, the signals stabilize approximately in the order shown in Steps 1-4, above.

Load/Store Instruction.

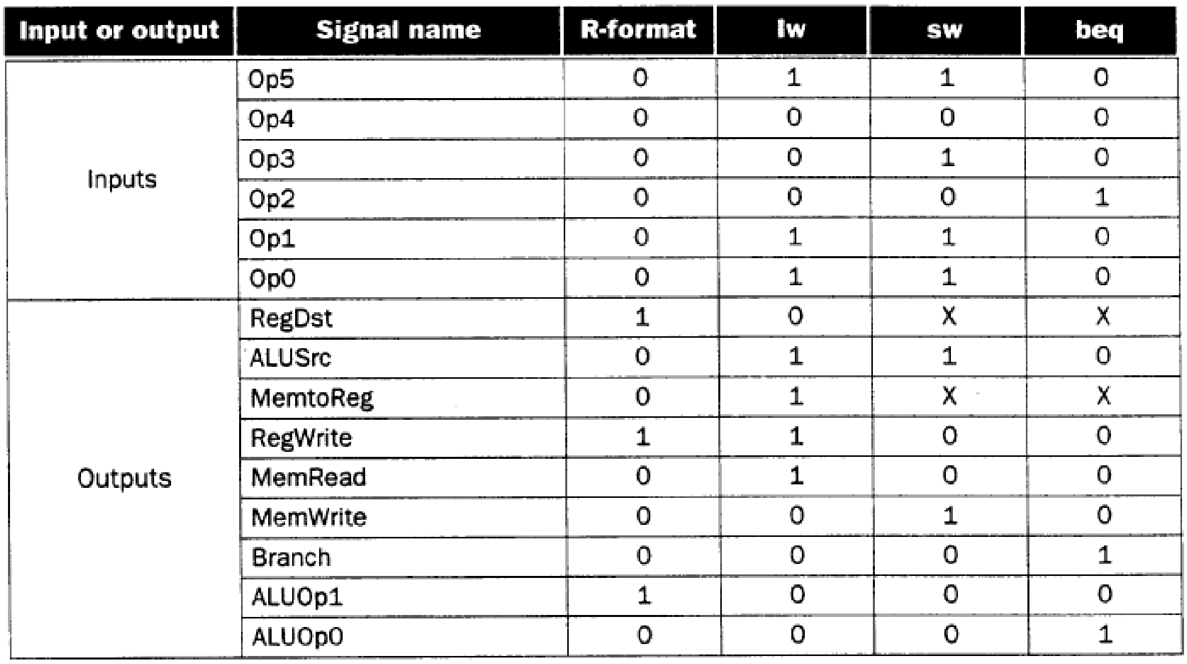
Execution of a load/store instruction (e.g., `lw $t1, offset($t2)`) using the datapath developed in Section 4.3.1 involves the following steps:
Fetch instruction from instruction memory and increment PC
Read register value (e.g., base address in `$t2`) from the register file
ALU adds the base address from register `$t2` to the sign-extended lower 16 bits of the instruction (i.e., `offset`)
Result from ALU is applied as an address to the data memory
Data retrieved from the memory unit is written into the register file, where the register index is given by `$t1` (Bits 20-16 of the instruction).

Branch Instruction.

Execution of a branch instruction (e.g., `beq $t1, $t2, offset`) using the datapath developed in Section 4.3.1 involves the following steps:
Fetch instruction from instruction memory and increment PC
Read registers (e.g., `$t1` and `$t2`) from the register file. The adder sums PC + 4 plus sign-extended lower 16 bits of `offset` shifted left by two bits, thereby producing the branch target address (BTA).
ALU subtracts contents of `$t1` minus contents of `$t2`. The Zero output of the ALU directs which result (PC+4 or BTA) to write as the new PC.

Final Control Design.

Now that we have determined the actions that the datapath must perform to compute the three types of MIPS instructions, we can use the information in the table bellow to describe the control logic in terms of a truth table. This truth table is optimized to yield the datapath control circuit.



Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Extended Control for New Instructions

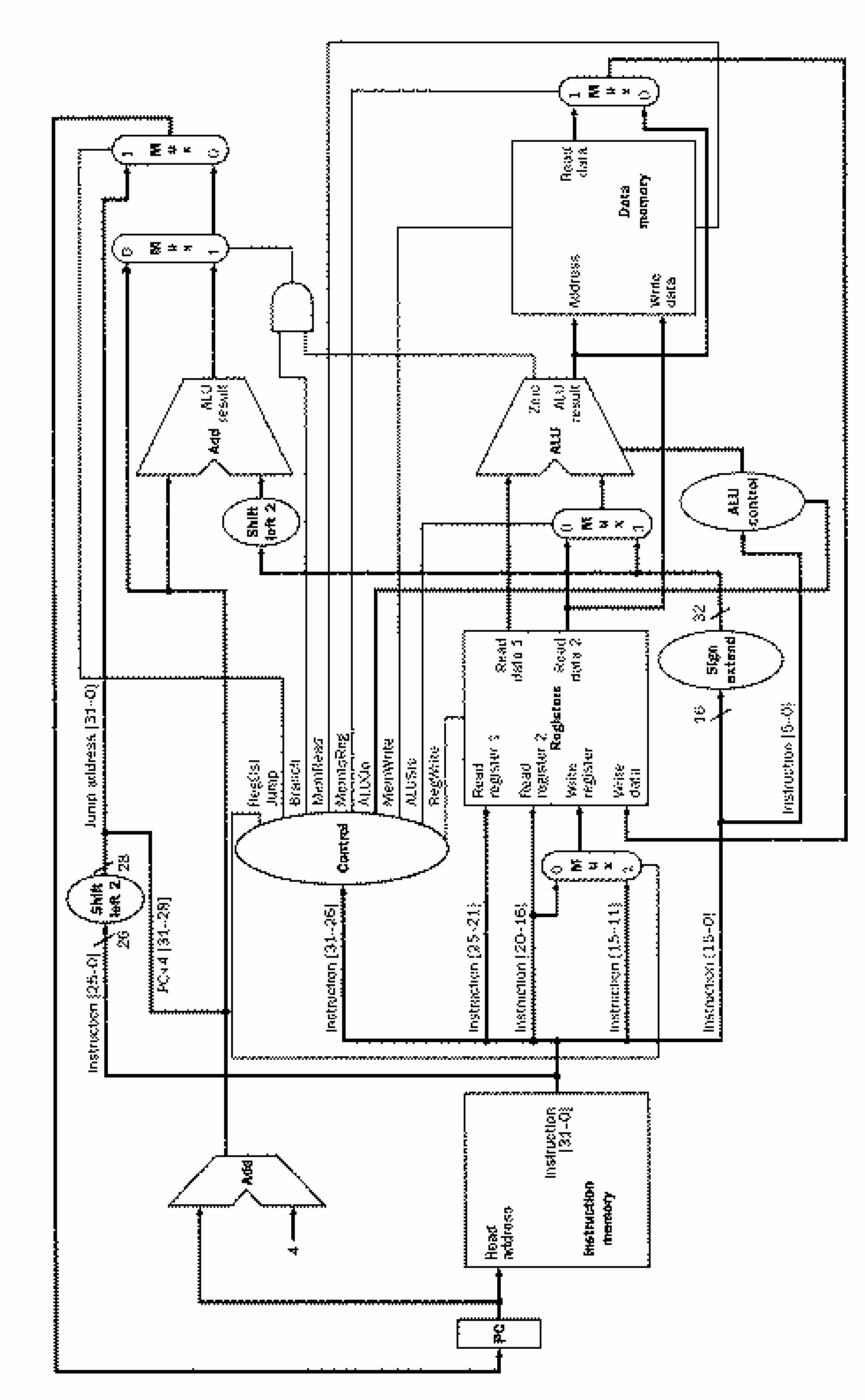
The jump instruction provides a useful example of how to extend the already created single-cycle datapath, to support new instructions. Jump resembles branch (a conditional form of the jump instruction), but computes the PC differently and is unconditional. Identical to the branch target address, the lowest two bits of the jump target address (JTA) are always zero, to preserve word alignment. The next 26 bits are taken from a 26-bit immediate field in the jump instruction (the remaining six bits are reserved for the opcode). The upper four bits of the JTA are taken from the upper four bits of the next instruction (PC + 4). Thus, the JTA computed by the jump instruction is formatted as follows:

- **Bits 31-28:** Upper four bits of (PC + 4)
- **Bits 27-02:** Immediate field of jump instruction
- **Bits 01-00:** Zero (00_2)

The jump is implemented in hardware by adding a control circuit to Figure 4.13, which is comprised of:

An additional multiplexer, to select the source for the new PC value. To cover all cases, this source is PC+4, the conditional BTA, or the JTA.
An additional control signal for the new multiplexer, asserted only for a jump instruction (opcode = 2).

The resulting augmented datapath is shown in the next figure.



Limitations of the Single-Cycle Datapath

The single-cycle datapath is not used in modern processors, because it is inefficient. The critical path (longest propagation sequence through the datapath) is five components for the load instruction. The cycle time t_c is limited by the settling time t_s of these components. For a circuit with no feedback loops, $t_c > 5t_s$. In practice, $t_c = 5kt_s$, with large proportionality constant k, due to feedback loops, delayed settling due to circuit noise, etc. It is possible to compute the required execution time for each instruction class from the critical path information. The result is that the Load instruction takes 5 units of time, while the Store and R-format instructions take 4 units of time. All the other types of instructions that the datapath is designed to execute run faster, requiring three units of time.

The problem of penalizing addition, subtraction, and comparison operations to accomodate loads and stores leads one to ask if multiple cycles of a much faster clock could be used for each part of the fetch-decode-execute cycle. In practice, this technique is employed in CPU design and implementation, as discussed in the following sections on multicycle datapath design. We already explained that datapath actions can be interleaved in time to yield a potentially fast implementation of the fetch-decode-execute cycle that is formalized in a technique called pipelining.

Multicycle Datapath Design

We designed a single-cycle datapath by grouping instructions into classes decomposing each instruction class into constituent operations deriving datapath components for each instruction class that implemented these operations.

Now we use the single-cycle datapath components to create a multi-cycle datapath, where each step in the fetch-decode-execute sequence takes one cycle. This approach has two advantages over the single-cycle datapath:

Each functional unit (e.g., Register File, Data Memory, ALU) can be used more than once in the course of executing an instruction, which saves hardware (and, thus, reduces cost)

Each instruction step takes one cycle, so different instructions have different execution times. In contrast, the single-cycle datapath that we designed previously required every instruction to take one cycle, so all the instructions move at the speed of the slowest.

We next consider the basic differences between single-cycle and multi-cycle datapaths.

Cursory Analysis.

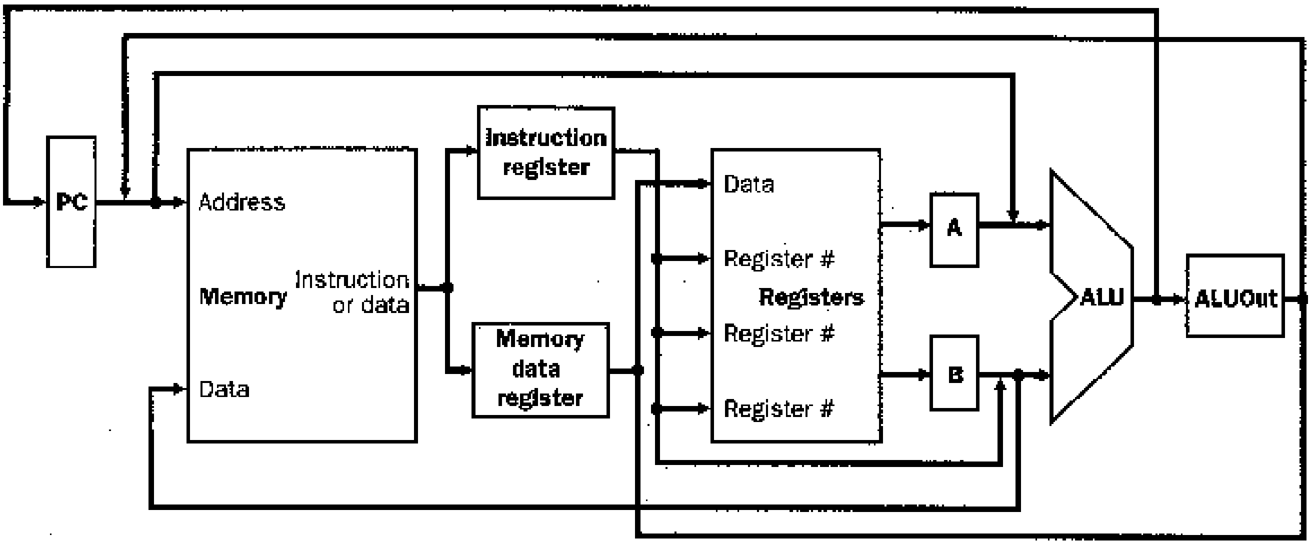
Observe the following differences between a single-cycle and multi-cycle datapath.

In the multicycle datapath, one memory unit stores both instructions and data, whereas the single-cycle datapath requires separate instruction and data memories.

The multicycle datapath uses on ALU, versus an ALU and two adders in the single-cycle datapath, because signals can be rerouted throuh the ALU in a multicycle implementation.

In the single-cycle implementation, the instruction executes in one cycle (by design) and the outputs of all functional units must stabilize within one cycle. In contrast, the multicycle implementation uses one or more registers to temporarily store (buffer) the ALU or functional unit outputs. This *buffering* action stores a value in a temporary register until it is needed or used in a subsequent clock cycle.

Below the scheme of a simple multicycle datapath.



Note that there are two types of *state elements* (e.g., memory, registers), which are:

Programmer-Visible (register file, PC, or memory), in which data is stored that is used by subsequent instructions (in a later clock cycle); and
Additional State Elements(buffer registers), in which data is stored that is used in a later clock cycle of the same instruction.

Thus, the additional (buffer) registers determine what functional units will fit into a given clock cycle and the data required for later cycles involved in executing the current instruction. In the simple implementation presented herein, we assume for purposes of illustration that each clock cycle can accomodate one and only one of the following operations:

- Memory access
- Register file access (two reads *or* one write)
- ALU operation (arithmetic or logical)

New Registers.
As a result of buffering, data produced by memory, register file, or ALU is saved for use in a subsequent cycle. The following temporary registers are important to the multicycle datapath implementation discussed in this section:

- *Instruction Register* (IR) saves the data output from the Text Segment of memory for a subsequent instruction read;
- *Memory Data Register* (MDR) saves memory output for a data read operation;

- *A and B Registers* (A,B) store ALU operand values read from the register file; and
- *ALU Output Register* (ALUout) contains the result produced by the ALU.

The IR and MDR are distinct registers because some operations require both instruction and data in the same clock cycle. Since all registers except the IR hold data only between two adjacent clock cycles, these registers do not need a write control signal. In contrast, the IR holds an instruction until it is executed (multiple clock cycles) and therefor requires a write control signal to protect the instruction from being overwritten before its execution has been completed.

New Muxes.
We also need to add new multiplexers and expand existing ones, to implement sharing of functional units. For example, we need to select between memory address as PC (for a load instruction) or ALUout (for load/store instructions). The muxes also route to one ALU the many inputs and outputs that were distributed among the several ALU's of the single-cycle datapath. Thus, we make the following additional changes to the single-cycle datapath:

Add a multiplexer to the first ALU input, to choose between (a) the A register as input (for R- and I-format instructions) , or (b) the PC as input (for branch instructions).
On the second ALU, the input is selected by a four-way mux (two control bits). The two additional inputs to the mux are (a) the immediate (constant) value 4 for incrementing the PC and (b) the sign-extended offset, shifted two bits to preserve alighment, which is used in computing the branch target address.

The details of these muxes are shown in Figure 4.16. By adding a few registers (buffers) and muxes (inexpensive widgets), we halve the number of memory units (expensive hardware) and eliminate two adders (more expensive hardware).

New Control Signals.
The datapath shown next is multicycle, since it uses multiple cycles per instruction. As a result, it will require different control signals than the single-cycle datapath, as follows:

Write Control Signals for the IR and programmer-visible state units
Read Control Signal for the memory; and
Control Lines for the muxes.

It is advantageous that the ALU control from the single-cycle datapath can be used as-is for the multicycle datapath ALU control. However, some modifications are required to support branches and jumps. We describe these changes as follows.

Branch and Jump Instruction Support.
To implement branch and jump instructions, one of three possible values is written to the PC:

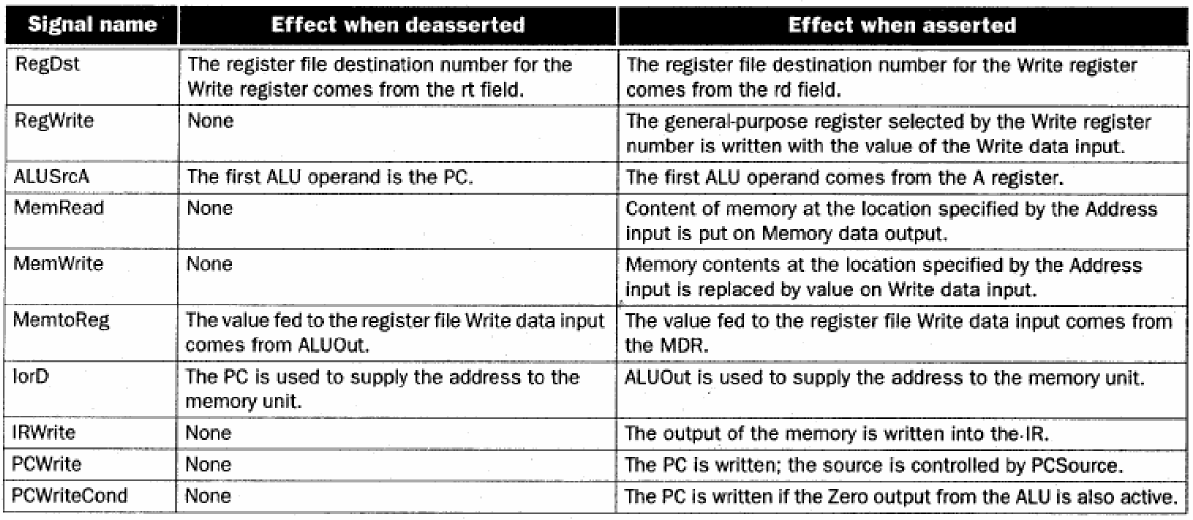
ALU output = PC + 4, to get the next instruction during the instruction fetch step (to do this, PC + 4 is written directly to the PC)
Register ALUout, which stores the computed branch target address.
Lower 26 bits (offset) of the IR, shifted left by two bits (to preserve alginment) and concatenated with the upper four bits of PC+4, to form the jump target address.

The PC is written unconditionally (jump instruction) or conditionally (branch), which implies two control signals - PCWrite and PCWriteCond. From these two signals and the Zero output of the ALU, we derive the PCWrite control signal, via the following logic equation:

PCWriteControl = (ALUZero *and* PCWriteCond) *or* PCWrite,

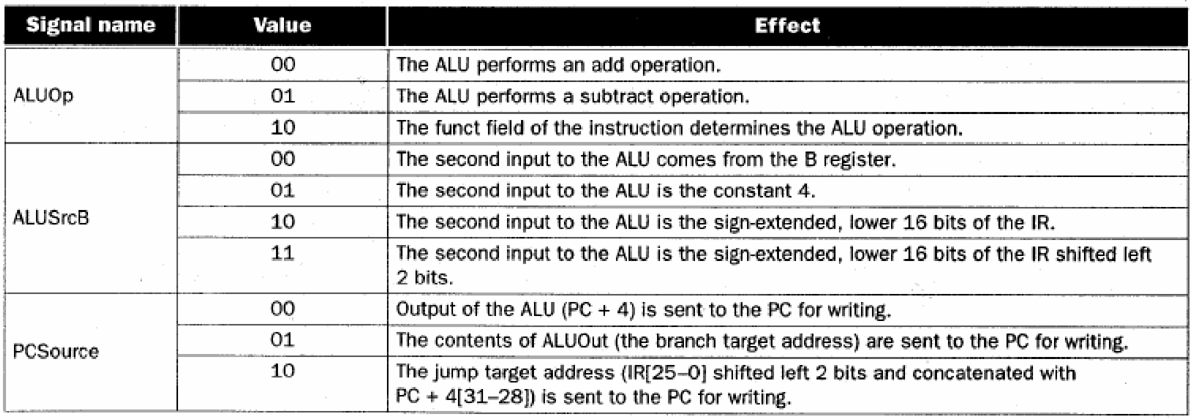
where ALUZero indicates if two operands of the `beq` nstruction are equal and the result of (ALUZero *and* PCWriteCond) determines whether the PC should be written during a conditional branch. We call the latter the *branch taken* condition. The figure on the next page shows the resultant multicycle datapath and control unit with new muxes and corresponding control signals. The table illustrates the control signals and their functions.

Actions of the 1-bit control signals

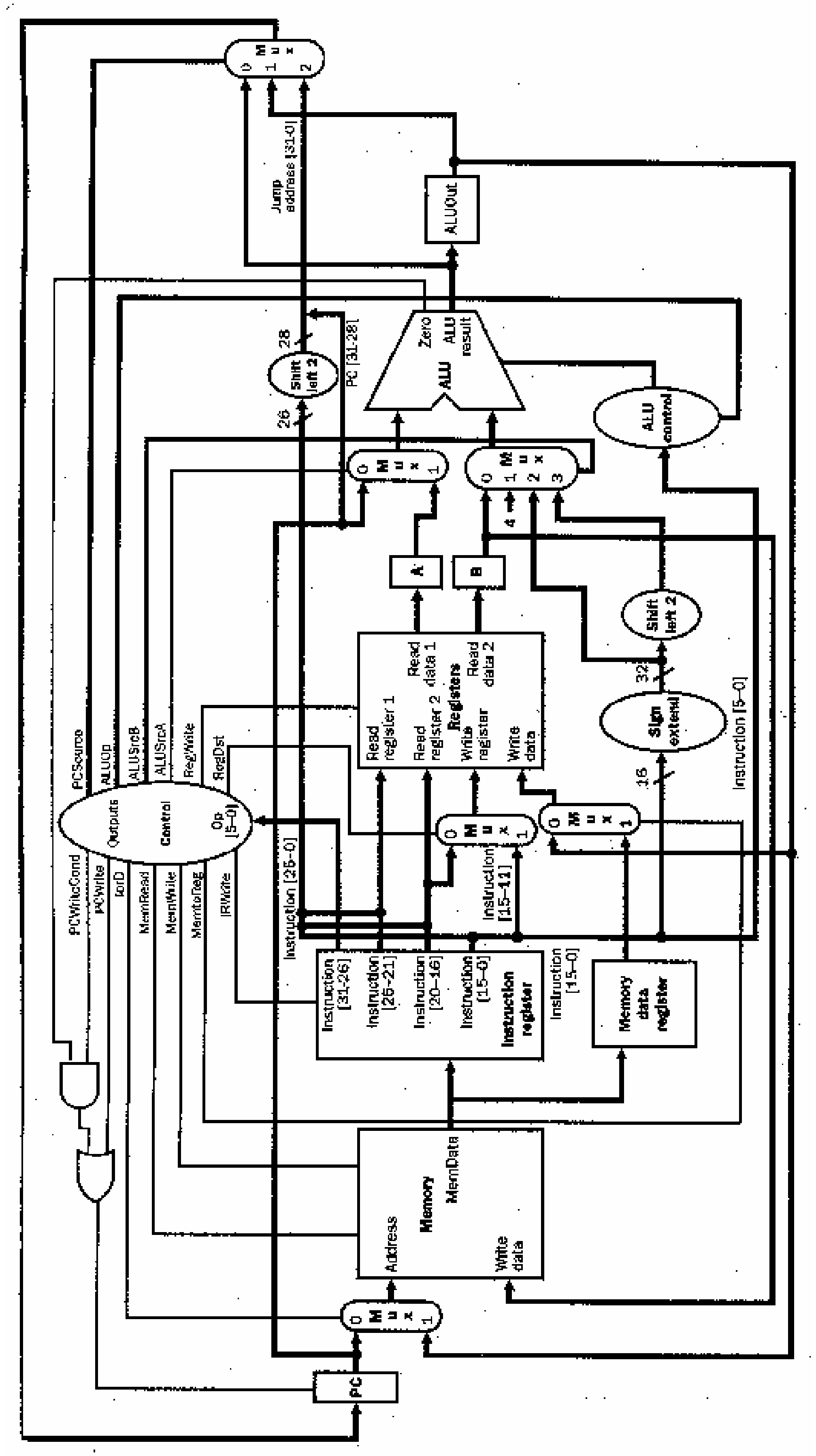


Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
lorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals



Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25–0] shifted left 2 bits and concatenated with PC + 4[31–28]) is sent to the PC for writing.


PC
Mux 0 1
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25-21]
Instruction [20-16]
Instruction [15-0]
Instruction register
Instruction [15-0]
Memory data register
Mux 0 1
Instruction [15-11]
Mux 0 1
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
Instruction [25-0]
Instruction [5-0]
16
Sign extend
32
Shift left 2
Mux 0 1
Mux 0 1 2 3
4
ALU control
ALU
Zero
ALU result
ALUOut
26
Shift left 2
28
PC [31-28]
Jump address [31-0]
Mux 0 1 2
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5-0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst

Multicycle Datapath and Instruction Execution

Given the datapath illustrated in the previous figure, we examine instruction execution in each cycle of the datapath. The implementational goal is balancing of the work performed per clock cycle, to minimize the average time per cycle across all instructions. For example, each step would contain one of the following:

- ALU operation
- Register file access (two reads or one write)
- Memory access (one read or one write)

Thus, the cycle time will be equal to the maximum time required for any of the preceding operations.

Since the datapath is designed to be edge-triggered and the outputs of ALU, register file, or memory are stored in dedicated registers (buffers), we can continue to read the value stored in a dedicated register. The new value, output from ALU, register file, or memory, is not available in the register until the next clock cycle.

In the multicycle datapath, all operations within a clock cycle occur in parallel, but successive steps within a given instruction operate sequentially. Several implementational issues present that do not confound this view, but should be discussed. One must distinguish between reading/writing the PC or one of the buffer registers, and reads/writes to the register file. Namely, I/O to the PC or buffers is part of one clock cycle, i.e., we get this essentially "for free" because of the clocking scheme and hardware design. In contrast, the register file has more complex hardware and requires a dedicated clock cycle for its circuitry to stabilize.

Instruction Fetch

In this first cycle that is common to all instructions, the datapath fetches an instruction from memory and computes the new PC (address of next instruction in the program sequence), as represented by the following pseudocode:

```
IR = Memory[PC]      # Put contents of Memory[PC] in
                     # Instr.Register
PC = PC + 4          # Increment the PC by 4 to preserve
                     # alignment
```

The PC is sent (via control circuitry) as an address to memory. The memory hardware performs a read operation and control hardware transfers the instruction at Memory[PC] into the IR, where it is stored until the next instruction is fetched. Then, the ALU increments the PC by four to preserve word alighment. The incremented (new) PC value is stored back into the PC register by setting PCSource = 00 and asserting PCWrite. Fortunately, incrementing the PC and performing the memory read are concurrent operations, since the new PC is not required (at the earliest) until the next clock cycle.

Instruction Decode and Data Fetch.
Included in the multicycle datapath design is the assumption that the actual opcode to be executed is not known prior to the instruction decode step. This is reasonable, since the new instruction is not yet available until completion of instruction fetch and has thus not been decoded.

As a result of not knowing what operation the ALU is to perform in the current instruction, the datapath must execute *only* actions that are:

- Applicable to *all* instructions and
- Not harmful to *any* instruction.

Therefore, given the *rs* and *rt* fields of the MIPS instruction format (per Figure 2.7), we can suppose (harmlessly) that the next instruction will be R-format. We can thus read the operands corresponding to *rs* and *rt* from the register file. If we don't need one or both of these operands, that is not harmful. Otherwise, the register file read operation will place them in buffer registers A and B, which is also not harmful.

Another action the datapath can perform is computation of the branch target address using the ALU, since this is the instruction *decode* step and the ALU is not yet needed for instruction execution. If the instruction that we are decoding in this step is not a branch, then no harm is done - the BTA is stored in ALUout and nothing further happens to it.

We can perform these preparatory actions because of the of MIPS instruction formats. The result is represented in pseudocode, as follows:

```
A = RegFile[IR[25:21]]     # First  operand = Bits 25-21 of
                          #instruction
B = RegFile[IR[20:16]]     # Second operand = Bits 25-21 of
                           # instruction
ALUout = PC + SignExtend(IR[15:0]) << 2 ;   # Compute BTA
```

where "x << n" denotes x shifted left by n bits.

Instruction Execute, Address Computation, or Branch Completion.
In this cycle, we know what the instruction is, since decoding was completed in the previous cycle. The instruction opcode determines the datapath operation, as in the single-cycle datapath. The ALU operates upon the operands prepared in the decode/data-fetch step, performing one of the following actions:

- *Memory Reference*: `ALUout = A + SignExtend(IR[15:0])`

  The ALU constructs the memory address from the base address (stored in A) and the offset (taken from the low 16 bits of the IR). Control signals are set as described on p. 387 opf the textbook.

- *R-format Instruction*: `ALUout = A op B`

  The ALU takes its inputs from buffer registers A and B and computes a result according to control signals specified by the instruction opcode, function field, and control signals `ALUop = 10`. The control signals are further described on p. 387 of the textbook.

- *Branch*: `if (A == B) then PC = ALUout`

  In branch instructions, the ALU performs the comparison between the contents of registers A and B. If A = B, then the Zero output of the ALU is asserted, the PC is updated (overwritten) with (1) the BTA computed in the preceding step (per Section 4.3.6.2), then (2) the ALUout value. If the branch is not taken, then the PC+4 value computed during instruction fetch (per Section 4.3.6.1) is used. This covers all possibilities by using for the BTA the value most recently written into the PC. Salient hardware control actions are discussed on p. 387 of the textbook.

- *Jump*: `PC = PC[31:28] || (IR[25:0] << 2)`

  Here, the PC is replaced by the jump target address, which does not need the ALU be computed, but can be formed in hardware as described on p. 387 of the textbook.

Memory Access or R-format Instruction Completion.
In this cycle, a load-store instruction accesses memory and an R-format instruction writes its result (which appears at ALUout at the end of the previous cycle), as follows:

```
MDR = Memory[ALUout]      # Load
Memory[ALUout] = B        # Store
```

where MDR denotes the memory data register.

For an R-format completion, where

```
Reg[IR[15:11]] = ALUout   # Write ALU result to register file
```

the data to be loaded was stored in the MDR in the previous cycle and is thus available for this cycle. The *rt* field of the MIPS instruction format (Bits 20-16) has the register number, which is applied to the input of the register file, together with RegDst = 0 and an asserted RegWrite signal.

From the preceding sequences as well as their discussion in the textbook, we are prepared to design a finite-state controller, as shown in the following section.

5.2.4 Finite State Control

In the single-cycle datapath control, we designed control hardware using a set of truth tables based on control signals activated for each instruction class. However, this approach must be modified for the multicycle datapath, which has the additional dimension of time due to the stepwise execution of instructions. Thus, the multicycle

datapath control is dependent on the current step involved in executing an instruction, as well as the next step.
There are two alternative techniques for implementing multicycle datapath control. First, a *finite-state machine* (FSM) or *finite state control* (FSC) predicts actions appropriate for datapath's next computational step. This prediction is based on the status and control information specific to the datapath's current step and actions to be performed in the next step. A second technique, called *microprogramming*, uses a programmatic representation to implement control.

Finite State Machine

An FSM consists of a set of states with directions that tell the FSM how to change states. The following features are important:

- Current state and inputs;
- Next-state function, also called the *transition function*, which converts inputs to a new state, and outputs of the FSM
- Outputs, which in the case of the multicycle datapath, are control signals that are asserted when the FSM is in a given state.
- We assume that all outputs not explicitly asserted are not asserted. Additionally, all multiplexer controls are explicitly specified if and only if they pertain to the current and next states.

Finite State Control

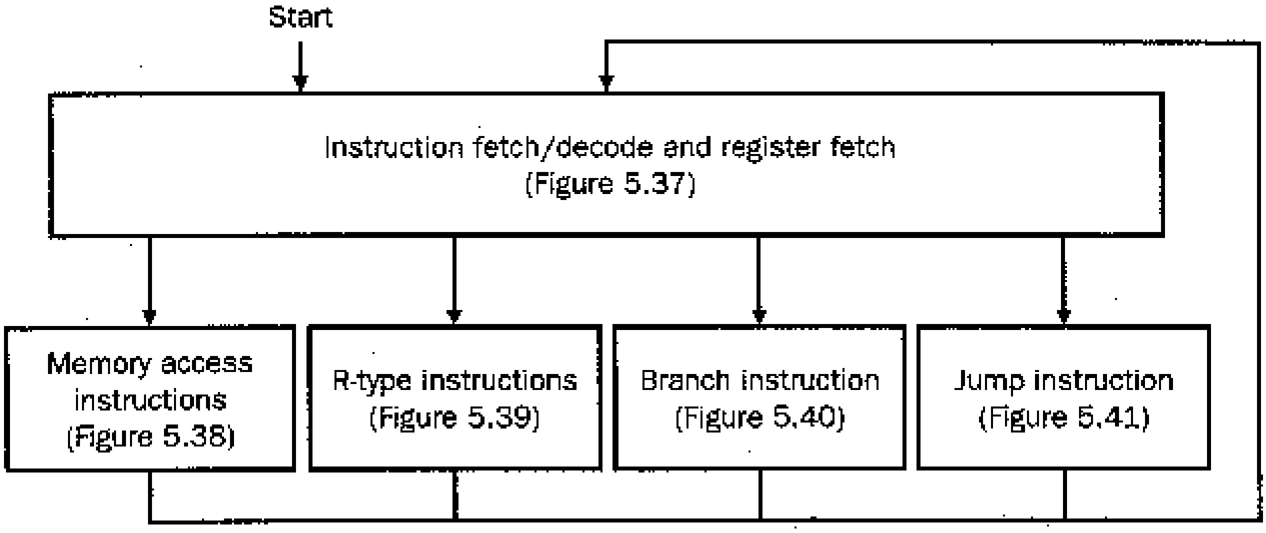
The FSC is designed for the multicycle datapath by considering the five steps of instruction execution given in, namely:

- Instruction fetch
- Instruction decode and data fetch
- ALU operation
- Memory access or R-format instruction completion
- Memory access completion

Each of these steps takes one cycle, by definition of the multicycle datapath. Also, each step stores its results in temporary (buffer) registers such as the IR, MDR, A, B, and ALUout. Each state in the FSM will thus:

- Occupy one cycle in time
- Store its results in a temporary (buffer) register.

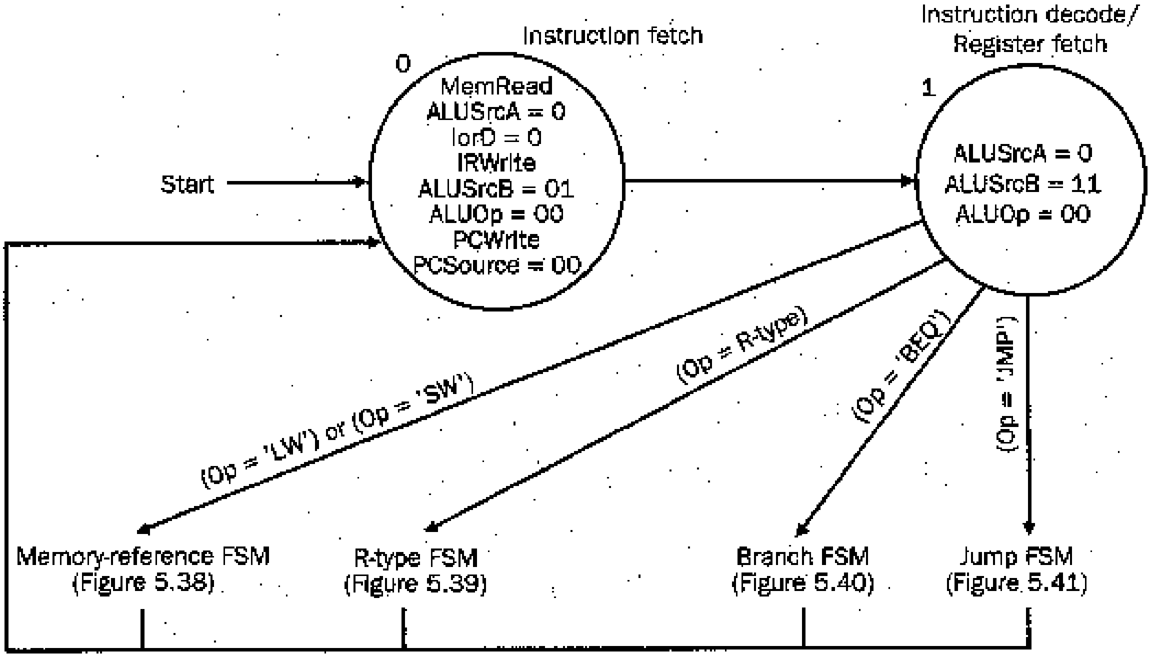
Also note that after completion of an instruction, the FSC returns to its initial state (Step 1) to fetch another instruction, as you can see in the scheme below.



Let us begin our discussion of the FSC by expanding steps 1 and 2, where State 0 (the initial state) corresponds to Step 1.

Instruction Fetch and Decode.

In the following scheme is the FSM representation for instruction fetch and decode shown. The control signals asserted in each state are shown within the circle that denotes a given state. The edges (lines or arrows) between states are labelled with the conditions that must be fulfilled for the illustrated transition between states to occur. Patterson and Hennessey call the process of branching to different states decoding, which depends on the instruction class after State 1 (i.e., Step 2, as listed above).

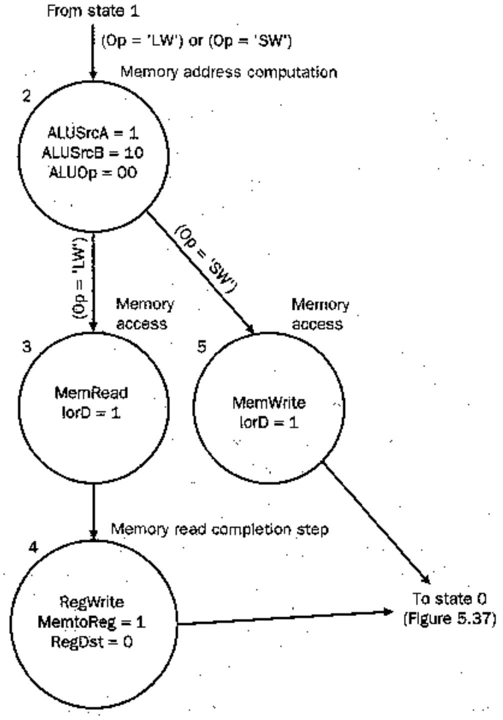


Memory Reference.

The memory reference portion of the FSC is shown in the figure bellow. Here, **State 2** computes the memory address by setting ALU input muxes to pass the A register (base address) and sign-extended lower 16 bits of the offset (shifted left two bits) to the ALU. After address computation, memory read/write requires two states:

- **State 3:** Performs memory access by asserting the MemRead signal, putting memory output into the MDR.
- **State 5:** Activated if `sw` (store word) instruction is used, and MemWrite is asserted.

In both states, the memory is forced to equal ALUout, by setting the control signal IorD = 1.



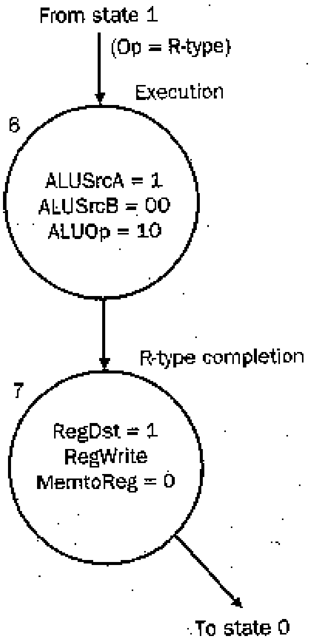
When **State 5** completes, control is transferred to **State 0**. Otherwise, **State 3** completes and the datapath must finish the load operation, which is accomplished by transferring control to **State 4**. There, MemtoReg = 1, RegDst = 0, and the MDR contents are written to the register file. The next state is **State 0**.

R-format Execution.

To implement R-format instructions, FSC uses two states, one for execution (**Step 3**) and another for R-format completion (**Step 4**). **State 6** asserts ALUSrcA and sets ALUSrcB = 00, which loads the ALU's A and B input registers from register file

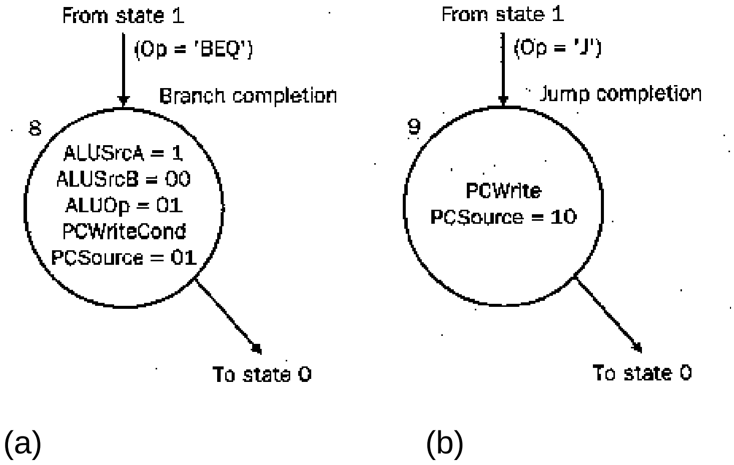
outputs. The ALUop = 10 setting causes the ALU control to use the instruction's *funct* field to set the ALU control signals to implement the designated ALU operation.
State 7 causes:

- the register file to write (assert RegWrite
- *rd* field of the instruction to have the number of the destination register (assert RegDst)
- ALUout selected as having the value that must be written back to the register file as the result of the ALU operation (by deasserting MemtoReg).



Branch Control.
Since branches complete during **Step 3**, only one new state is needed.
In **State 8**:

- Control signs that cause the ALU to compare the contents of its A and B input registers are set (i.e., ALUSrcA = 1, ALUSrcB = 00, ALUop = 01)
- the PC is written conditionally (by setting PCSrc = 01 and asserting PCWriteCond). Note that setting ALUop = 01 forces a subtraction, hence only the `beq` instruction can be implemented this way.



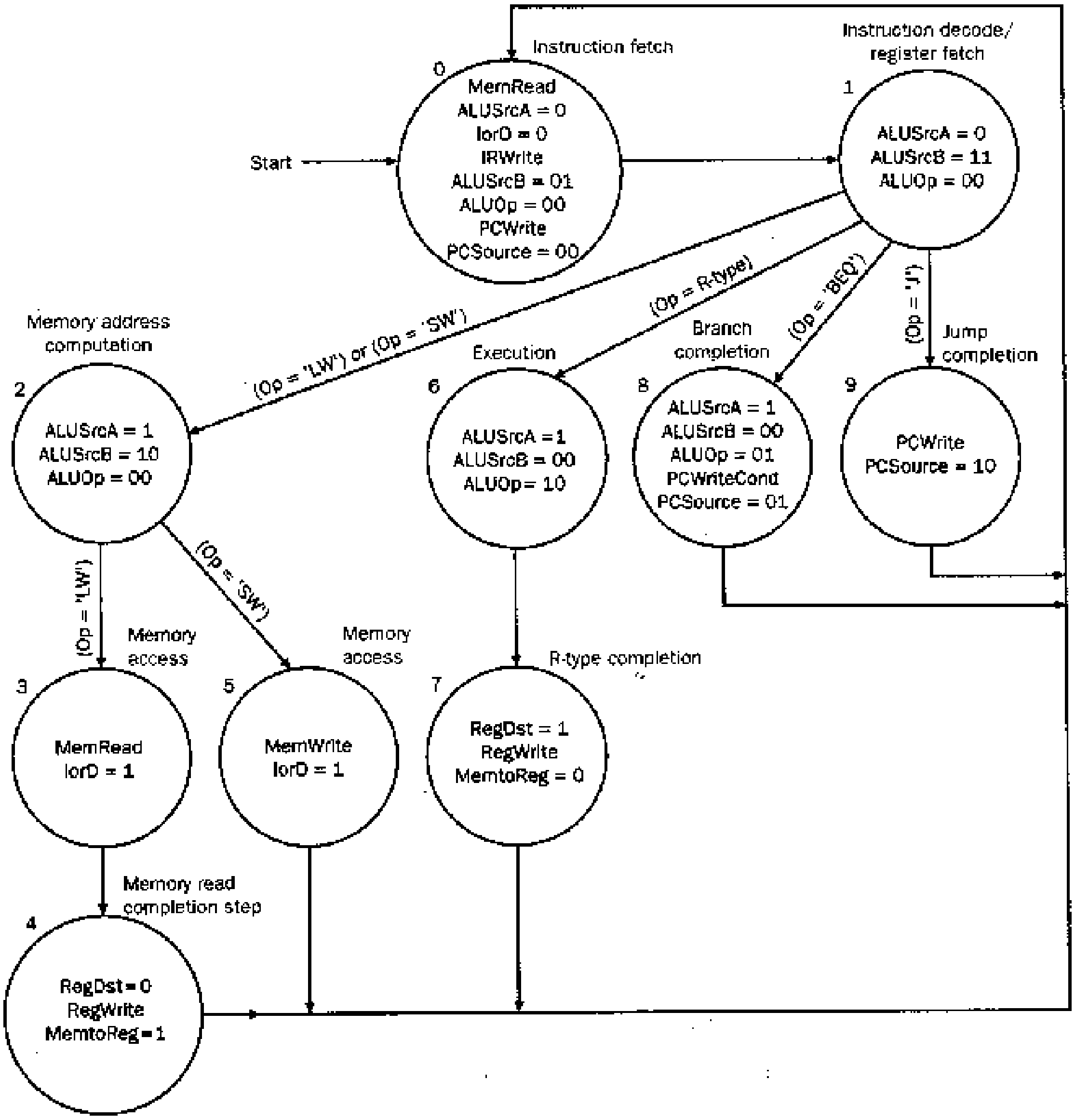
Representation of finite-state control for (a) branch and (b) jump instruction-specific states of the multicycle datapath.

Jump Instruction.

Similar to branch, the jump instruction requires only one state (#9) to complete execution. Here, the PC is written by asserting PCWrite. The value written to the PC is the lower 26 bits of the IR with the upper four bits of PC, and the lower two bits equal to 00_2. This is done by setting PCSrc = 10_2.

5.2.5 FSC and Multicycle Datapath Performance

The composite FSC is shown in the picture underneath.



When computing the performance of the multicycle datapath, we use this FSM representation to determine the critical path (maximum number of states encountered) for each instruction type, with the following results:

- Load: 5 states
- Store: 4 states
- R-format ALU instructions: 4 states
- Branch: 3 states
- Jump: 3 states

Since each state corresponds to a clock cycle (according to the design assumption of the FSC controller in, we have the following expression for CPI of the multicycle datapath:

```
CPI = [#Loads · 5 + #Stores · 4 + #ALU-instr's · 4 + #Branches
· 3 + #Jumps · 3] / (Total Number of Instructions)
```

5.2.6 Implementation of Finite-State Control

The FSC can be implemented in hardware using a read-only memory (ROM) or programmable logic array (PLA). Combinatorial logic implements the transition function and a state register stores the current state of the machine. The inputs are the IR opcode bits, and the outputs are the various datapath control signals (e.g., PCSrc, ALUop, etc.)
We next consider how the preceding function can be implemented using the technique of *microprogramming*.

5.2.7 Microprogrammed Control

It is possible to develop a control system design by using abstractions from the programming language. This technique is called microprogramming, it will help the control system design improvement of the correctness and it will also be more servient. When you are using the microinstructions that set the value of the datapath control signals, then there is only one that can write microprograms that implement a processor's control system.

- Microinstruction Format: this will initialize in what structure and content the microinstructions fields must have.
- Sequencing Mechanism: this will determine which instruction will be used next.
- Exception Handling: this will determine what actions must be taken when there are errors.

Microinstruction Format
The microinstructions are abstractions of low-level control that are used to program control logic hardware. The microinstructions format should be hold easy, and discourage or prohibit inconsistency.

For the implementation of the microinstructions, there must be specified for each field a set of non overlapping values. When there are signals that will never correspond they can use the still share the same field.

Field Name	Field Function
ALU control	This will specify the operations that are preformed bij the ALU during the clock cycle, the result will be written into ALUout.
SRC1	Source for the first ALU operand
SRC2	Source for the second ALU operand
Register control	This will read or write data from the register file.
Memory	Specify read or write, and the source for a write. For a read, specify the destination register.
PCWrite control	Specify how the process counter is to be written.
Sequencing	Specify how to choose the next microinstruction for execution .

In the hardware microinstructions are mostly stored in ROM. The microinstructions are mostly referred by sequential addresses to simplify sequencing.
There are three different sequencing process methods:

- Incrementation: is for which address of the current microinstruction is incremented to get the address of the next microinstruction.
- Branching: this is the microinstruction that will initialize the execution of the next MIPS instruction.
- Control-directed choice: this will choose the next microinstruction based on the control input.

The microinstructions are going to be the input of the microassembler, which will checks for inconsistencies. When inconsistencies are detected they are flagged and must be corrected for hardware implementation.

5.2.8 Microprogramming the datapath control

When we want to design the microprogrammed control we are going to use the fetch decode execute sequence. The instructions can sometime have a blank field, this will be when:

- A field that controls a functional unit (ALU, register file, memory) or causes state information to be written, can have a blank field this is when there is no control signal that should be asserted.
- A field that will only specifies control of an input multiplexer for a functional unit, can have a blank field when the datapath does not have to care about what value the output of the multiplexer has.

Instruction Fetch and Decode, Data Fetch
When the instruction will be executed it will first fetches the instruction, it will decode it and compute the sequential process counter and the branch target process counter.

Two examples:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
-----	-------------	------	--------	-------------------	--------	---------	------------
Fetch	Add	PC	4	---	Read PC	ALU	Seq

The ---- are the blank fields in the microinstruction.

- ALU control, SRC1, and SRC2 are set to compute PC+4, which is written to ALUout. The memory field will read the instruction. The PCwrite control will cause the ALU output to be written into the process counter, while the sequencing field will tell you what the next microinstruction is.
- The label field will be used to transfer control in the next Sequencing field when execution of the next instruction begins.

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing

-----	-------------	------	--------	-------------------	--------	---------	------------
---	Add	PC	Extshft	Read	---	---	Dispatch 1

- ALU control, SRC1, and SRC2 are set to store the process counter and the sign-extended, shifted IR[15:0] into ALUout. Register control causes data referenced by the *rs* and *rt* fields to be placed in ALU input registers A and B. output (PC + 4) to be written into the PC, while the Sequencing field tells control to go to dispatch table 1 for the next microinstruction address.

Dispatch Tables

Case statement that uses the opcode field and dispatch table i to select one of the different labels. The different labels are **Mem1** for memory reference instructions, **Rformat1** for arithmetic and logical instructions, **beq1** for conditional branches and **Jump1** for unconditional branches.

Each of these labels will point to a different microinstruction sequence that will be seen as a kind of subprogram.

Memory Reference Instructions

There are three microinstructions suffice to implement memory access in the terms of a MIPS load instruction:

Memory address computation

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
-----	-------------	------	--------	-------------------	--------	---------	------------
Mem1	Add	A	Extend	---	---	---	Dispatch 2

Memory read

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
-----	-------------	------	--------	-------------------	--------	---------	------------
LW2	---	---	---	---	Read ALU	---	Seq

Register file write

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
-----	-------------	------	--------	-------------------	--------	---------	------------
---	---	---	---	Write MDR	---	---	Fetch

Implementing a Microprogram

When you want to implement a microprogram it is useful when you think that it is a textual representation of the finite-state machine. A microprogram could then be implemented similar to the final state control, using a programmable logic array to encode the sequencing function and main control.

Sometimes it will be useful to store the control function in a ROM and then you will implement the sequencing function in some other way. The sequencer will use an incrementer to choose the next control instruction. The microcode storage will determine the values of the datapath control lines and the technique of how to select the next state. From the dispatch tables you can get the address of the next microinstruction to be executed.

This technique is preferred because it substitutes a simple counter for more complex address control logic, which is especially efficient when the microinstructions have a little branching. When you use ROM the microcode can be stored in its own memory and will be addressed by the microprogram counter. The same will happen to regular program instructions being addressed by an instruction sequencer.

The is the actually how microprogramming got started, by making the ROM and a fast counter. This will represent a great advance when you are using slower main memory for microprogram storage.

These days you have the cache and this will make a separate microprogram memory, it will be easier to store the microprogram in the main memory and the put the parts that you need in the cache. This is very fast and you are do not need extra hardware.

5.2.9 Exception Handling

There are two types of exceptions:

- An exception is an anomalous event from within the processor, such as arithmetic overflow.
- An interrupt is an event that causes an unexpected change in control flow. Interrupts are assumed to originate outside the processor, for example, an I/O request.

Basic Exception Handling Mechanism
When there is an exception detected, the processor’s control circuit must be able to save the address in the exception counter of the instruction that has caused the exception. When that is done it will transfer the control to the operating system at a prespecified address, this will be done by an exception handler.

The exception got a routine that either helps the program to recover from the exception or when there is an error message it will attempt to terminate the program.

When the program execution is going to continue after the exception is detected and handled, the exception counter register helps to determine where it need to restart the program.
The operating system will have two methods to handle a exception.

1. The machine can have an exception counter which contains the codes that respectively represent the cause of the exception and it also will have the address of the exception causing instruction.
2. This will use vectored interrupts, where the address to which control is transferred and it will be determine the what it has to do by the cause of that

exception. When vectored interrupts are not employed, the control will be transferred to only one address, regardless of what cause it is. The cause will be used to determine what action the exception handling routine should take

5.2.10 Hardware Support

MIPS is using the second method, the non vectored exceptions. To support this the datapath must be capable to do this. Therefore we need two registers

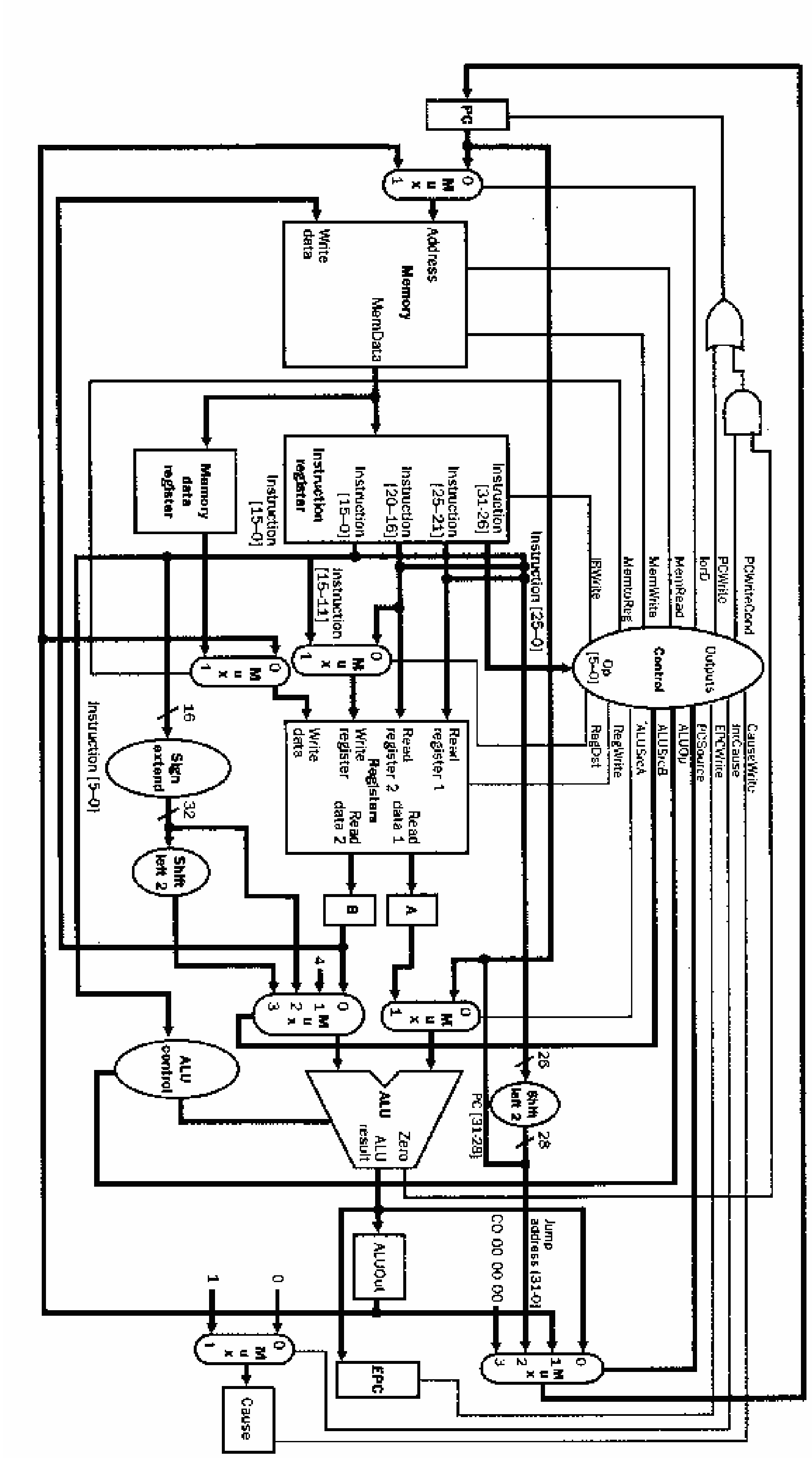
- Exception counter: 32-bit register holds the address of the exception-causing instruction.
- *Cause*: 32-bit register contains a binary code that describes the cause or type of exception.

There are two additional control signals needed **EPCWrite** and **CauseWrite**, they will write the right information to the exception and cause registers. When you want to write the process counter into the exception counter register, it will give an error. This is because the process counter is incremented at the instruction fetch instead of instruction execution when the exception actually accurs.

But when there is an exception detected the ALU will subtract 4 from the process counter and the ALUout register and this must be written to the exception counter register.

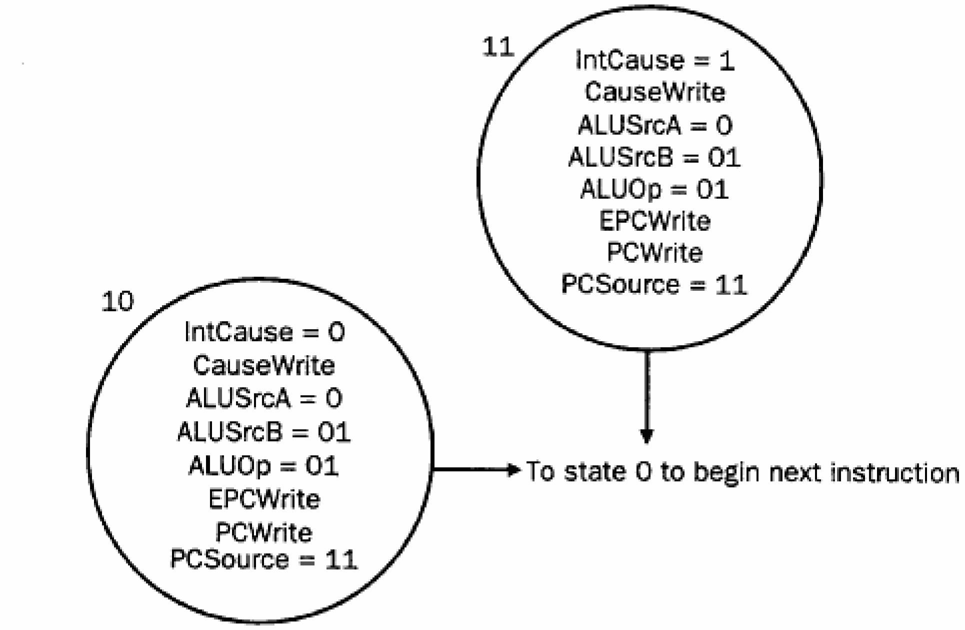
So there are hardware support for the datapath needed to implement the exception handling.

On the next page you can see the representation of the datapath with a establishment for exception handling.


PC
0
M
u
x
1
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [15–0]
Memory data register
Instruction [15–11]
Instruction [25–0]
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5–0]
CauseWrite
IntCause
EPCWrite
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
A
B
16
Sign extend
32
Shift left 2
Instruction [5–0]
4
0
1
2
3
ALU control
ALU
Zero
ALU result
26
Shift left 2
28
PC [31-28]
Jump address [31-0]
CO 00 00 00
ALUOut
EPC
Cause
0
1

In the finite state diagram, you can see that each is preceding two types of exceptions that can be handled, each is using one state.
For every exception type there are state actions:

1. It will set the cause register to get the exception type.
2. It will compute and save the process counter – 4 into the exception counter register to make it available to return the address.
3. It will write the address to the process counter so the control can be transferred to the exception handler.

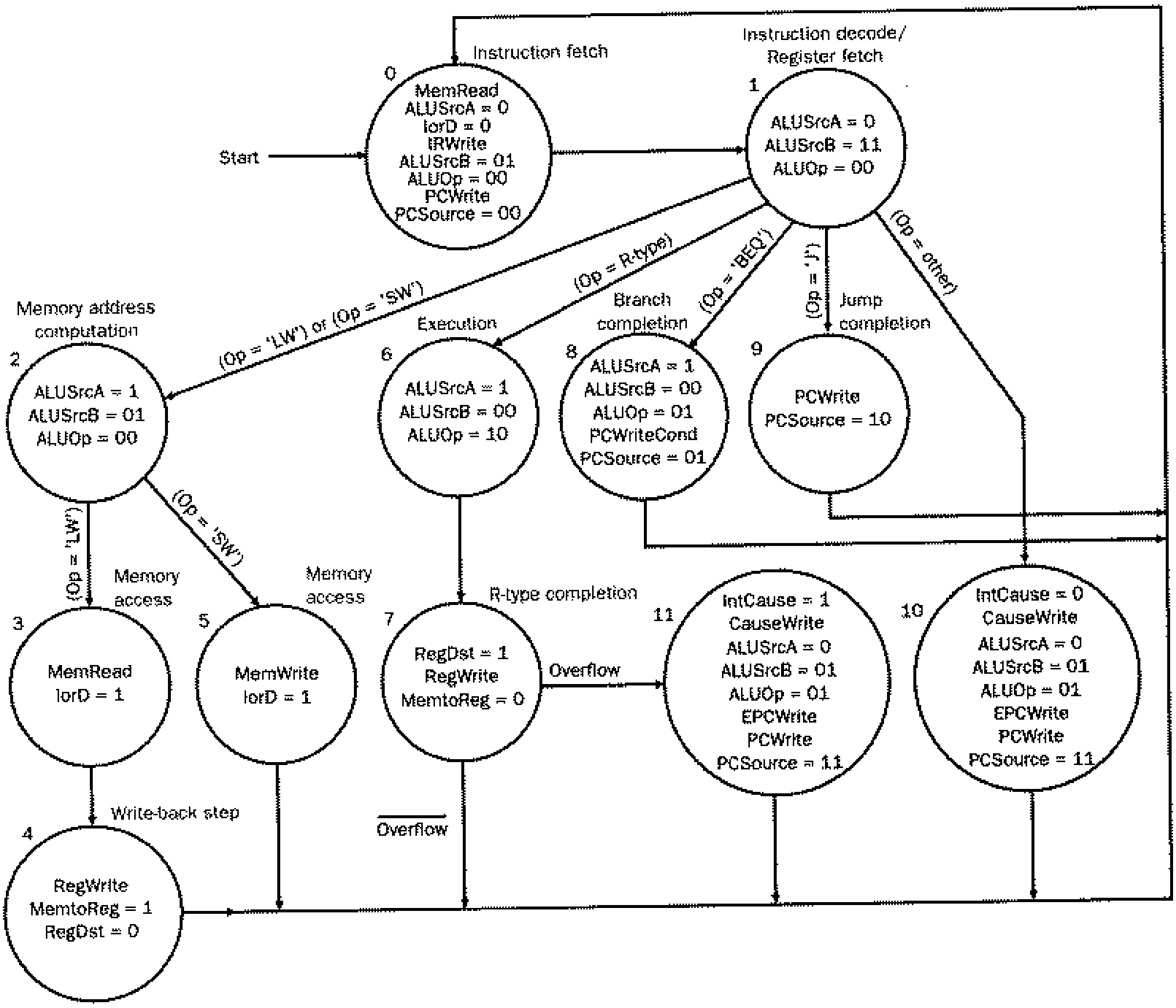


Exception Detection

Two examples of exception types:

- Undefined Instruction: this is because the finite state must be modified to define the next state value.
- Arithmetic Overflow: an ALU can be designed to have an overflow detection with a signal output that is called overflow. You get an output when overflow is detected.

You can see what will happen when there is overflow in the next figure at circle 7. This is also the representation from the finite state control of the MIPS datapath, with exception handling.


Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
Instruction decode/
Register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
(Op = other)
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 01
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Overflow
11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
Write-back step
4
RegWrite
MemtoReg = 1
RegDst = 0
$\overline{\text{Overflow}}$

6 Bibliography

A Websites

www.mips.com/content/.../documents/R4000%20Microprocessor%20Users%20Manual.pdf
ftp.utcluj.ro/pub/users/nedevschi/ac/AC2006-2007/5_ALU%20Design_2006.pdf
www.cs.tcd.ie/Michael.Manzke/3ba5/3ba5_sixth_lecture.pdf
logos.cs.uic.edu/366/notes/index.html
www.e-lation.net/site/sony.html
en.wikipedia.org/wiki/John_L._Hennessy
www.cs.rochester.edu/u/sandhya/csc252/lectures/
www.seas.upenn.edu/~blundell/mipscompiler.pdf
chortle.ccsu.edu/AssemblyTutorial/TutorialContents.html
www.cepba.upc.es/docs/sgi_doc/SGI_Developer/books/MProAsLg_PG/sgi_html/ch08.html
www.cs.unibo.it/~solmi/teaching/arch_2002-2003/AssemblyLanguageProgDoc.pdf
www.inf.uni-konstanz.de/dbis/teaching/ws0304/computing-systems/download/rs-05.pdf
www.cs.umd.edu/class/spring2003/cmsc311/Notes/Mips/format.html
logos.cs.uic.edu/366/notes/MIPS%20Quick%20Tutorial.htm
www.go-ecs.com/mips/miptek1.htm
en.wikipedia.org/wiki/RISC
www.mips.com/content/Documentation/MIPSDocumentation/ProcessorCores/doclibrary
www.cs.tcd.ie/Jeremy.Jones/vivio%203.4/dlx/dlxtutorial.htm
cse.stanford.edu/class/sophomore-college/projects-00/risc/mips/index.html
shekel.jct.ac.il/~citron/ca/ca-lec5/sld011.htm
www.xs4all.nl/~vhouten/mipsel/r3000-isa.html
courses.csusm.edu/cs331xz/notes/
www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html
www.compapp.dcu.ie/~ray/CA225b.html
http://www.doc.ic.ac.uk/lab/secondyear/spim/spim.html
http://en.wikipedia.org/wiki/MIPS_architecture
http://www.compapp.dcu.ie/~ray/CA225b.html
http://www.cs.purdue.edu/homes/hosking/502/spim/raw.html
http://www.csl.cornell.edu/courses/ece314/MIPSXCallingXConventionsXSummary.pdf
http://logos.cs.uic.edu/366/notes/MIPS%20Quick%20Tutorial.htm

B Books

Computer Organization & Design
The hardware / software interface
David A. Patterson and John L. Hennessy

Principles of computer architecture
Miles J. Murdocca and Vincent P. Heuring