CS10 – Computer Architecture and Organization

My Notes

Leonard Mohr

Student at Foothill College

leonardmohr@gmail.com

Contents

Section 1

Computer Abstractions and Technology

Subsection 1.1

Introduction

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

Figure 1. We can describe storage in binary or decimal notation. We are much more familiar with the decimal term. Note also the size difference between the two: binary term gets progressivly larger.

1.1.1 Program Performance

One of the main goals for both the hardware designer and the software designer is to improve performance. This can be achieved in different ways (just think about how quick sort is much faster than bubble sort, and the apple M1 processor is much faster than the intel processors they replaced).

Hardware or software component	How this component affects performance	Where is this topic covered?
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed	Other books!
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 4, 5, and 6
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapters 4, 5, and 6

Figure 2. Here we can see the various ways program performance can be improved.

Check Yourself

1. As mentioned earlier, both the software and hardware affect the performance of a program. Can you think of examples where each of the following is the right place to look for a performance bottleneck?

   a) The algorithm chosen

   > If a program that sorts a really long list of names is taking a long time, you might want to look at the algorithm being used.

   b) The programming language or compiler

> If your program could is not language dependent, you could look at using a compiled language (like C), as opposed to an interpreted language like Python.

c) The operating system

> If one program runs well, but two at a time don't, then maybe the operating system isn't distributing it's resources efficiently.

d) The processor

> If the computer in general is using a lot of energy, you might want to look at the processor. Similarily, if you are doing compute heavy tasks such as vdeo editing, you might need a processor upgrade.

e) The I/O system and devices

> If it takes a long time to write to a hard drive, you might look at upgrading to a SSD.

Subsection 1.2

Eight Great Ideas in Computer Architecture

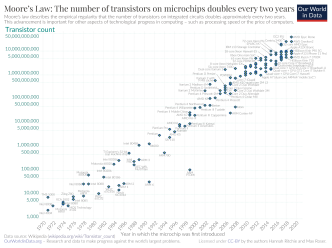


Figure 3. Moore's Law in action.

1. **Moore's Law**
   Moore's Law resulted from a 1965 prediction that integrated circuit (IC) resources would double every 18-24 months.

2. **Use Abstraction to Simplify Design**
   To increase productivity, by design both computer architects and programmers try use abstractions to hide the lower-level details and provide a simpler to work with. For example, the operating system abstracts away the complexity of the memory system so that programs are provided with a much simplified view of the memory, and the details are handled by the operating system.

3. **Make the Common Case Fast**
   Better performance gains can be made if you optimize for what the prgram is going to do most often.

4. **Performance via Parallelism**
   We can make a program a lot faster by performing multiple tasks at once. This is especially true with the advent of multi-core processors.

5. **Performance via Pipelining**
   If you are moving a lot of bricks from one place to another (using just manpower) it would be a lot more efficient to set up a line of people, and pass the bricks down the line, than to just have everyone running back and forth. The same principle can be used in computers.

6. **Performance via Prediction**
   When a processor encounters an if statement, it might say that on average the result is true, and procede like it is, so that it keep going, and then the if qualifier can be processed at a later date.

7. **Hierarchy of Memories**
   By using different types of memory, from really small and really fast, to really large and slow, a memory hierarchy is created. Caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy.

8. **Dependability via Redundancy**
   Because we are sad when a computer dies, we want to prevent the death of the computer. To do this we make them dependable. This can be achieved by including redundant components that can both take over in the event of a failure as well as detect when a failure has occured.

Subsection 1.3
Below Your Program

The Operating System is a great example of abstraction. Some of its most important functions are:

- Handling basic input and output operations
- Allocating storage and memory
- Provided for protected sharing of the computer among multiple applications using it simultaneously.

Another example of abstraction is high-level programming languages like C. When computers first came about, programmers programmed in binary; they wrote their programs in 1's and 0's (the language of the computer). Since that was tedious, they invented the assembler which would convert assembly language into binary code. Next came the compiler which would convert higher order languages into assembly.

Another benefit of the programming languages is it allows one to chose the specific language that is best for the task. Also, programs don't have to be written for a specific processor, since the compiler and assembler can package it for different computers.

Subsection 1.4
Under the Covers

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. This organization is independent of hardware technology: you can place every piece of every computer, past and present, into one of these five categories.

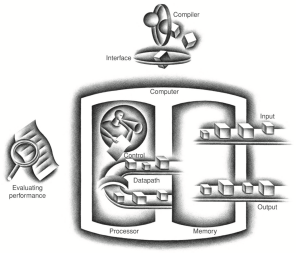


Figure 4. Here we see the flow of information in a computer. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

1.4.1 Parts of the Computer

- Integrated Circuit: Also called a chip. A device combining dozens to millions of transistors.
- Central Processing Unit (CPU): Also called the processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.
- Datapath: The component of the processor that performs arithmetic operations
- Control: The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.
- Memory: The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

- Dynamic Random Access Memory (DRAM): Memory built as an integrated circuit; it provides random access to any location Access times are 50 nanoseconds and cost per gigabyte in 2012 was \$5 to \$10.
- Cache Memory: Consists of a small, fast memory that acts as a buffer for the DRAM memory.
- Static Random Access Memory: SRAM is faster but less dense, and hence more expensive, than DRAM.
- Instruction Set Architector: The interface between the hardware and the lowest-level software. This includes all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.
- Application Binary Interface (ABI): The user portion of the instruction set plus the operating system interfaces used by application programmers. It defines a standard for binary portability across computers.

Subsection 1.6

Performance

When talking about computers, we often want to look at the performance of the computer. But how can we define performance?

Definition 1

Response Time
Also referred to as **execution time**. This is the total time required for the computer to complete a task, including disk access, memory aaccesses, I/O activities, operating system overhead, CPU execution time, and so on.

Definition 2

Throughput / Bandwidth
This measures the number of tasks computed per unit time.

For the time being, we will mostly be looking at response time. So that a higher number represents a machine with better performance we will say that

$$\text{Performance}_{\text{X}} = \frac{1}{\text{Execution time}_{\text{X}}}.$$

Also, we often want to say that computer "X is n times as fast as Y", to compute n we say:

$$\frac{\text{Performance}_{\text{X}}}{\text{Performance}_{\text{Y}}} = \frac{\text{Execution Time}_{Y}}{\text{Execution Time}_{X}} = n.$$

CPU execution time or simply **CPU time** is the actual time the CPU spends computing for a specific task.

1.6.1 Clock Rate and Clock Period

Designers refer to the length of a **clock period** both as the time for a complete clock cycle (e.g., 250 picoseconds) and as the *clock rate* (e.g., 4 gigahertz, or 4GHz), which is the inverse of the clock period.

For example a clock rate of

$$\begin{aligned} 4\text{GHz} &= 4,000,000,000 \frac{\text{cycles}}{\text{sec}} \\ &\equiv \frac{1 \text{ cycle}}{4,000,000,000 \text{ second}} \\ &= 0.00000000025 \frac{\text{cycle}}{\text{second}} \\ &= \frac{1}{250} \frac{\text{cycle}}{\text{picosecond}}. \end{aligned}$$

We can relate clock cycles and clock cycle time to CPU time:

$$\begin{matrix}\text{CPU execution time} \\ \text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles} \\ \text{for a program}\end{matrix} \times \text{Clock cycle time}$$

or alternatively,

$$\begin{matrix}\text{CPU execution time} \\ \text{for a program}\end{matrix} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate.}}$$

1.6.2 Instruction Performance

In addition to thinking about clock cycles and CPU time, we also need to think about number of instructions there are in a program and the time each instruction takes:

$$\text{CPU clock cycles} = \text{Instructions for a Program} \times \frac{\text{average clock cycles}}{\text{instruction}}$$

Clock Cycles per Instruction (CPI) is the average number of clock cycles per instruction for a program or program fragement.

1.6.3 The Classic CPU Performance Equation

Putting everything from above together we have

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

and

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}.$$

We also have

$$\text{Time} = \text{Seconds / Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle.}}$$

Processors these days can vary their clock rates. For example Intel Core i7 chips temporarily increase clock rate by 10% until the chip gets too warm. Thus we need to use the average clock rate for a program.

Figure 5.

Subsection 1.10

Fallacies and Pitfalls

1.10.1 Amdahl's Law

Definition 3 **Amdahl's Law**
A rule stating that the performance enhancement possible with a given improvement

is limited by the amount that the improved feature is used:

$$\begin{aligned}\text{Execution time after improvement} = &\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} \\ &+ \text{ Execution time unaffected}\end{aligned}$$

where Amount of improvement $= n$.

We thus see that there is only so much benefit that can be achieved by improving a part of the program that is rarely used. For this reason we want to make the common case fast!

1.10.2 MIPS

One should be wary about using only a subset of the performance equation as a performance metric; one can't determine performance just by looking at clock rate, instruction count, or CPI alone.

An alternative to time is **MIPS (million instructions per second)**:

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}.$$

There are a couple problems with MIPS:

- MIPS specifies the instruction execution rate but does not take into account the capabilities of these instructions. Therefore we can't compare computers with different instruction sets.
- MIPS varies between programs on the same computer; thus a computer cannot have a single MIPS rating.

$$\text{MIPS} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}.$$

1.10.3 Check Yourself

Consider the following performance measurements for a program:

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

Which computer is faster and which has the higher MIPS rating?

First looking at computer A.

$$\begin{aligned}\text{Time}(A) &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} \\ &= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}} \\ &= \frac{10 \times 10^9 \text{ instructions} \times \frac{1 \text{ cycle}}{\text{instruction}}}{4 \times 10^9 \frac{\text{cycles}}{\text{second}}} \\ &= 2.5 \text{ seconds}.\end{aligned}$$

$$\begin{aligned}\text{MIPS}(A) &= \frac{\text{Instruction Count}}{\text{Execution Time } \times 10^6} \\ &= \frac{10 \times 10^9 \text{ instructions}}{2.5 \text{ seconds } \times 10^6} \\ &= \frac{4 \times 10^3 \text{ million instructions}}{\text{second}}.\end{aligned}$$

Now looking at computer B.

$$\begin{aligned}\text{Time}(B) &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} \\ &= \frac{\text{Instruction Count } \times \text{ CPI}}{\text{Clock Rate}} \\ &= \frac{8 \times 10^9 \text{ instructions} \times \frac{1.1 \text{ cycles}}{\text{instruction}}}{4 \times 10^9 \frac{\text{cycles}}{\text{second}}} \\ &= 2.2 \text{ seconds}.\end{aligned}$$

$$\begin{aligned}\text{MIPS}(B) &= \frac{\text{Instruction Count}}{\text{Execution Time } \times 10^6} \\ &= \frac{8 \times 10^9 \text{ instructions}}{2.2 \text{ seconds } \times 10^6} \\ &\approx \frac{3.6 \times 10^3 \text{ million instructions}}{\text{second}}.\end{aligned}$$

We thus see that Computer A has a higher MIPS score but runs slower.

Section 2

Instructions: Language of the Computer

Subsection 2.2

Operations of the Computer Hardware

2.2.1 MIPS Operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

2.2.2 MIPS Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Note how just about all operations have exactly three operands. This conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number.

Subsection 2.3

Operands of the Computer Hardware

2.3.1 Memory Operands

Let's say we have the following C statement

```
1  g = h + A[8];
```

What will be the associated MIPS code if g and h are in registers $\$s1$ and $\$s2$, and that the base address of the array is in $\$s3$.

```
1  lw    $t0, 32($s3)     # Temporary reg $t0 gets A[8]
2  add   $t0, $s2, $t0    # Temporary reg $t0 gets h + A[8]
3  sw    $t0, 48($s3)     # A[12] <—— $t0
```

Note that MIPS uses byte addressing, and so to get to the 8th index, you need to add $8 * 4$ since the size of each array index is 4 bytes. Also note that words must start with addresses that are multiples of 4.

Alignment Restriction is the requirement that data be aligned in memory on natural boundaries.

2.3.2 Constant or Immediate Operands

Let's say for example that we want to add 5 to some register for whatever reason, instead of loading that from a memory location into a temporary register, and then adding the temporary register to the desired register we can use the instruction add immediate:

```
1  addi    $s3, $s3, 4        # $s3 = $s3 + 4
```

By including this constant opperation the processor can operations much faster and using less energy. Because more than half of MIPS arithmetic instructions have a constant as an operand when running the SPEC CPU2006 benchmarks this is an example of making the common case fast.

Overflow occurs when after performing an arithmatic operation on two numbers results in a number that can't be stored in the number of bits available to register (32 in case of MIPS).

Subsection 2.4

Signed and Unsigned Numbers

2.4.1 Unsigned Numbers

In any number base, the value of the ith digit d is

$$d \times \text{ Base}^i.$$

Thus for example, in binary

$$\begin{aligned} 101 &= (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) \\ &= 4 + 0 + 1 \\ &= 5. \end{aligned}$$

2.4.2 Twos Complement

The idea of twos complement is that the most significant bit indicates the sign of the number, 1 if negative and 0 if positive (zero being treated as a positive number). But instead of merely reprsenting the sign, it represents the negative equivalent value of that number. So for example

$$\begin{aligned} 1 \text{ represents } &-1 \\ 10 \text{ represents } &-2 \\ 100 \text{ represents } &-4 \end{aligned}$$

and the other bits are their normal positive values:

$$\begin{aligned} 11 \text{ represents } &-1 \\ 111 \text{ represents } &-1 \\ 101 \text{ represents } &-3 \\ 110 \text{ represents } &-2. \end{aligned}$$

To achieve two's complement representation you

1. Start with the equivalent positive number
2. Invert all bits (change every 0 to 1, and every 1 to 0)

3. Add 1 to the inverted number, ignoring overflow.

The reason this works is because $x + \bar{x} = -1$ and therefore $\bar{x} + 1 = -x$, where $\bar{x}$ is x inverted. You can use this process to convert from negative to positive and vice versa:

- Twos complement representation of -5

$$\begin{aligned} 0101 &= 5 \\ 1010 &= \bar{5} && \textit{Inverse of 5} \\ 1011 &= -5 && \textit{-5 represented in twos complement} \end{aligned}$$

- Twos complement representation of 5

$$\begin{aligned} 1011 &= -5 \\ 0100 &= \overline{-5} \\ 0101 &= 5. \end{aligned}$$

In two's complement, if you want to use more bits to represent the same number, you just use **sign extension** (repeat the most significant bit):

$$\begin{aligned} 0111 \equiv 7 &\equiv 0000\ 0111 \\ 1011 \equiv -5 &\equiv 1111\ 1011. \end{aligned}$$

Subsection 2.5

Representing Instructions in the Computer

Now that we know how to tell the computer what to do (assembly language), what instructions are the computer actuall following (machine language)? Well, first of all, computers just read a stream of 1's and 0's. Depending upon the context, this stream could represent a string, a number, or an instruction.

2.5.1 Hexadecimal

Although we now know how to write numbers in binary, it can be a bit tedious, so we use hexadecimial (base 16). Because both binary and hexadecimal are powers of two, they play nicely together.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

Figure 6. To convert from hexadecimal to binary just replace hexadecimal digit by the corresponding four binary digits and vice versa. If the length of the binary number is not a multiple o 4, go from right to left.

2.5.2 MIPS Fields

Because computers just read a string of 1's and 0's, how does the computer know where each instruction begins and ends? When converting from assembly $\longrightarrow$ binary, MIPS instructions are formatted into 32 bits.

Register Instructions (R-type)

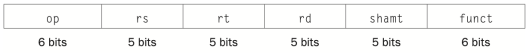


Figure 7. The various MIPS fields for R-type (register) instructions.

- op: Basic operation of the instruction, traditionally called the opcode. The value of this lets computer know meaning of, and size of the following fields.
- rs: The first register source operand
- rt: The second register source operand
- rd: The register destination operand. It gets the result of the operation.
- shamt: Shift amount. See 2.6 for more details.
- funct: Function. This field, often called the function code, selects the specific variant of the operation in the op field.

Immediate Instructions (I-type)

You might have noticed that not all instructions fit into the above format. For example Let's say you wanted to add a constant to a register and store it in another? Well, at the moment we would only have 5-bits available to hold a number, that's not a lot. The max constant value would be $2^5 - 1 = 31$ (assuming unsigned number), that's not very big. Instead we have a different instruction format:

Note that instructions with this format type are not just add immidiate type instructions, they are also used for load word instructions (among others). The following instruction

```
1  lw    $t0, 1200($t1)   # $t0 <-- A[300]
2  add   $t0, $s2, $t0    # $t0 <-- h + A[300]
3  sw    $t0, 1200($t1)   # A[300] <-- $t0
```

is expressed as follows:

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

and in binary:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

From this, you can see that each register has an associated reference number:

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Here are the opcodes for the instructions we have seen so far:

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

2.5.3 Check Yourself

What MIPS instruction does this represent?

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

From the ocode of 0 and function code of 34 we know this is a subtract function call.
We are dealing with registers $8, 9, 10$ which are `$t0`, `$t1`, and `$t2` respectively.
Because the MIPS register arguments are orderd `rd`, `rs`, `rt` we have get the MIPS code

```
1  sub     $t2, $t0, $t1
```

Subsection 2.7

Instructions for Making Decisions

2.7.1 Branch Instructions

MIPS has two decision making instructions:

- `beq register1, register2, L1`
  Go to the statement labeled `L1` if the value in `register1` equals the value in `register2`;
- `bne register1, register2, L1`
  Go to statement labeled `L1` if value in `register1` does not equal value in `register2`.

For example, the C code

```
1  if (i == j)
2      f = g + h;
3  else
4      f = g-h;
```

where `f` through `j` correspond to `$s0` through `$s4`, corresponds to the MIPS code

```
1       bne     $s3, $s4, Else
2       add     $s0, $s1, $s2       # f = g + h
3       j       Exit                # go to Exit
4
```

```
5  Else:
6      sub     $s0, $s1, $s2      # f = g-h
7
8  Exit:
```

Notice how we did things in reverse; instead of checking for equality (like in the C code), we checked for inequality.

The assembler relieves the compiler and the assembly language programmer from the tedium of calculated addresses for branches.

2.7.2 Loops

How could we use our branch statements to compile the following while Loop:

```
1  while (save[i] == k)
2      i += 1;
```

where `i` and `k` correspond to registers `$s3` and `$s5`. The address of `save[0]` is in `$s6`.

```
1  Loop:
2    sll     $t1, $s3, 2     # $t1 <-- i * 4
3    add     $t1, $t1, $s6   # $t1 <-- address of save[i]
4    lw      $t0, 0($t1)     # $t0 <-- save[i]
5    bne     $t0, $s5, Exit  # if (save[i] != k) Exit
6    addi    $s3, $s3, 1     # i += 1
7    j       Loop            # go to Loop
8
9  Exit:
```

This is an example of a **basic block**. This is a sequence without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning.

Because MIPS is byte addressed, to get the memory address of `save[i]` we need to multiply $i \cdot 4$ and add that to the base address. We then load that value to a temporary register, and test the condition.

2.7.3 Unsigned Comparison Tests

In addition to testing for equality, we might also want to test for less than, or greater than. To test if less than the instruction "set on less than"

```
1    sltu    $t0, $s3, $s4    # $t0 = 1 if $s3 < $s4; otherwise 0
```

This sets the destination register to 1 if less than, and 0 if not. An immediate version of this instruction also exists:

```
1    sltiu    $t0, $s2, 10    # $t0 = 1 if $s2 < 10, 0 otherwise
```

All relative conditions equal, not equal, less than, less than or equal, greater than, and greater than or equal, can all be created using `slt`, `slti`, `beq`, `bne`, and `$zero` register.

2.7.4 Signed Comparison Tests

We have to remember that with negative numbers, the most significant bit is turned on, and so using an unsigned comparison between -1 and 5: $1111_{\text{TWO}} < 0101_{\text{TWO}}$ would return false. Thus, for signed comparisons we have `slt` and `slti`:

```
slt     $t0, $s3, $s4     # $t0 = 1 if $s3 < $s4; otherwise 0
slti    $t0, $s2, 10     # $t0 = 1 if $s2 < 10, 0 otherwise
```

2.7.5 Bounds Checking Shortcut

Let's say we want to check if both

$$x < y \quad \text{and} \quad x \geq 0$$

where x is a and y are signed integers (and $y > 0$). Assume that `$s0` holds x and `$s1` holds y. Than we could do

```
# Check if x < y
   slt     $t0, $s0, $s1
   beq     $t0, $zero, False

# Check if x >= 0
   slt     $t0, $s0, $zero       # If x < 0
   bne     $t0, $zero, False

# Statement is true
   j       True
```

We can actually check for both $x < y$ and $x \geq 0$ in one go by treating x and y as unsigned integers. The idea here is that since a negative number will have its most significant bit turned on, it looks like a very large unsigned number. Therefore no matter how large y is, if x is a negative number it will appear as a larger unsigned number. Thus,

$$0 \leq x \leq y$$

will only be true if x is nonnegative and less than y, and when treated as unsigned numbers, testing if $x < y$ will also check if $x \geq 0$:

```
sltu $t0, $s0, $s1
beq  $t0, $zero, False
j    True
```

This shortcut is really helpful if we want to check if some index is out of bounds for a given array. For example, let's say we have an array `example` whose upper bounds y. Then we could use the code above to check if some index i is in bounds ($0 \leq x < y$).

Subsection 2.8

Supporting Procedures in Computer Hardware

Thus far, we have only really learned how to perform a sequence of instructions, but what if we want to perform that same sequence of instructions a lot of times, and maybe we want to act upon some variables. In C we have functions, and in assembly, we have **procedures**. To perform a procedure in MIPS, we follow the follwoing steps:

A **procedure** is a stored subroutine that performs a specific task based on the parameters with which it is provided.

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.

3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

The following registers are used for procedure calling

- `$a0-$a3`: four argument registers in which to pass parameters
- `$v0-$v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

The return address register is populated by the **jump-and-link-instruction** (jal):

`jal ProcedureAddress`. This instruction simultaneously jumps to ProcedureAddress, and populates the return address register (`$ra`) with the address of the following instruction. This way the computer knows what to perform once the procedure is finished.

The **return address** is a link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in regeister `$ra`.

The **caller** is the one that puts the parameter values in `$a0-$a3` and uses `jal X` to jump to procedure `X` (**callee**). Once the callee has finished, it places the results in `$v0` and `$v1`, and then returns control to the caller using `jr $ra`. This is an unconditional jump to the address specified in a register.

To make all of this work, we need to store the address of the current instruction being executed. We do this in a register called the **program counter** (PC). The `jal` instruction saves PC + 4 in register `$ra`.

2.8.1 Nested Procedures

Let's say you have the following program:

```
1  #include <stdio.h>
2
3  int fact(int n)
4  {
5      if (n < 1)
6          return 1;
7      else
8          return (n * fact(n-1));
9  }
10
11 int main() {
12     int result = fact(3);
13     printf("%d\n", result);
14     return 0;
15 }
```

Because `fact(n)` can call itself, you have to be careful about saving the return address register (`$ra`), as well as any other registers that may be used (more info below). We can write the above program in MIPS as follows:

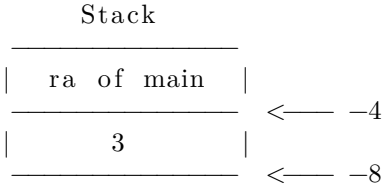
```
1  main:
2    addi    $a0, $zero, 3
3    jal     fact
4    add     $a0, $v0, $zero # store result for printing
5    addi    $v0, $zero, 1   # print int syscall
6    syscall
7    addi    $v0, $zero, $10 # exit syscall
8    syscall
9
10 fact:
11   addi    $sp, $sp, -8    # make space for two items on stack
12   sw      $ra, 4($sp)     # save return address
13   sw      $a0, 0($sp)     # save arg n
14   slti    $t0, $a0, 1     # test if n < 1
15   beq     $t0, $zero, L1  # if n >= 1, goto L1
16   lw      $ra, 4($sp)     # restore $ra
17   lw      $a0, 0($sp)     # restore arg n
18   addi    $v0, $zero, 1   # return 1
19   addi    $sp, $sp, 8     # pop two items from stack
20   jr      $ra             # return to caller
21
22 L1:
23   addi    $a0, $a0, -1    # n += -1
24   jal     fact            # calls fact and $ra <-- PC+4
25   lw      $a0, 0($sp)     # restore arg n
26   lw      $ra, 4($sp)     # restore $ra
27   addi    $sp, $sp, 8     # Pops two items from stack
28   mul     $v0, $a0, $v0   # return n*fact(n-1)
29   jr      $ra             # return to caller
```

Let's look at how the register values, and stack changes as we go through the program:

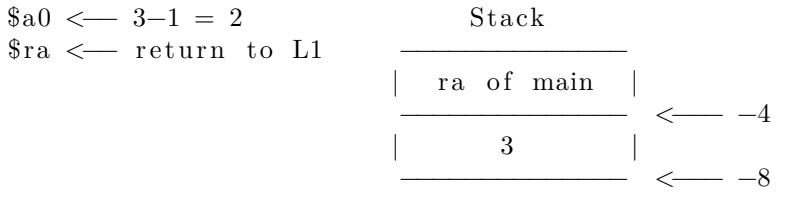
1. In `main` (before first call of fact):

   ```
   $a0 <---- 3
   $ra <---- return to main
   ```

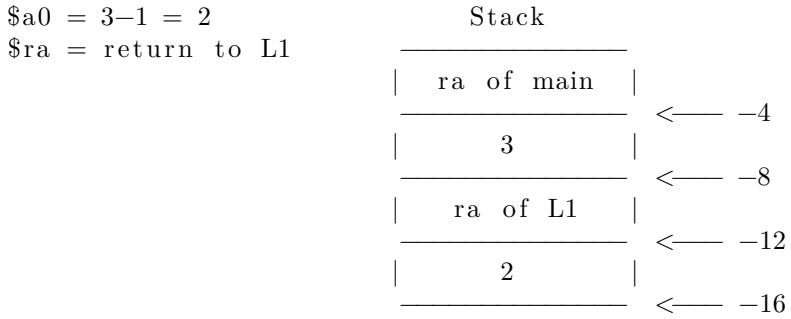
2. During first call of `fact(3)`



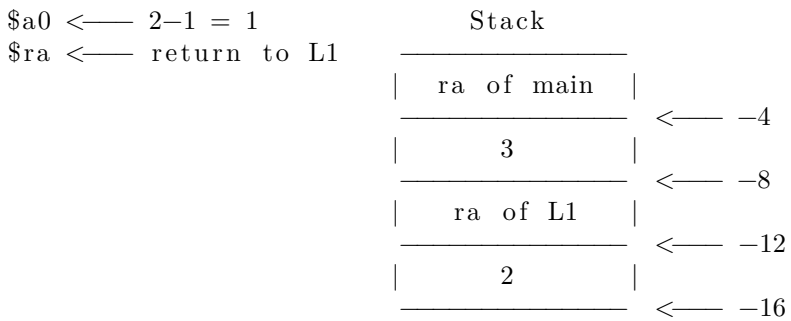
3. First break to `L1`:



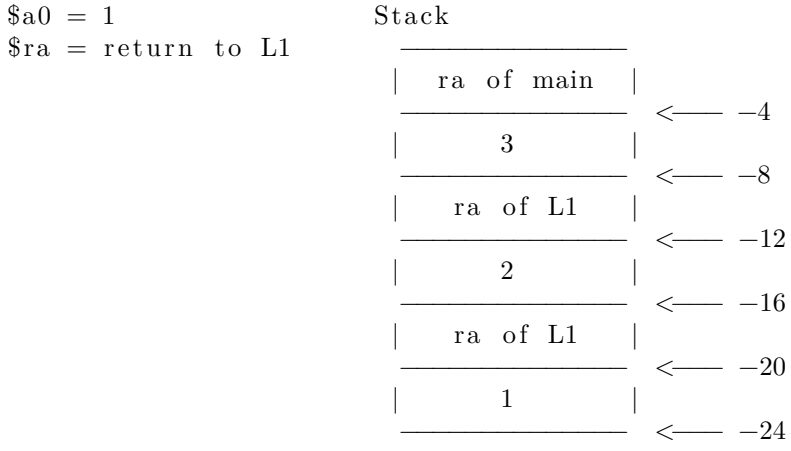
4. During second call to `fact(2)`



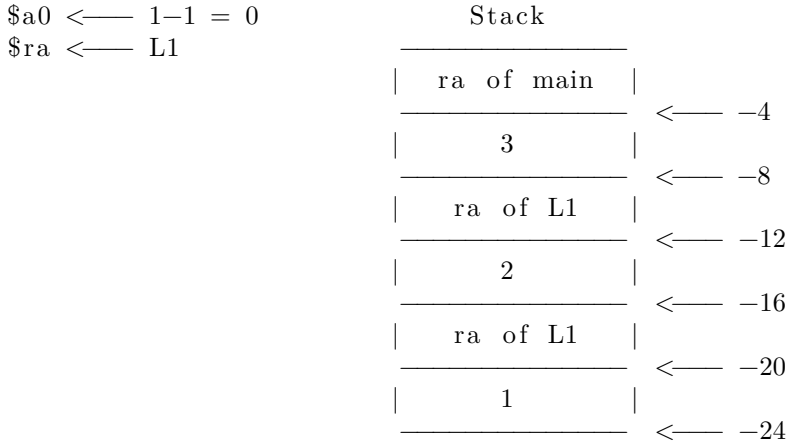
5. During second break to `L1`



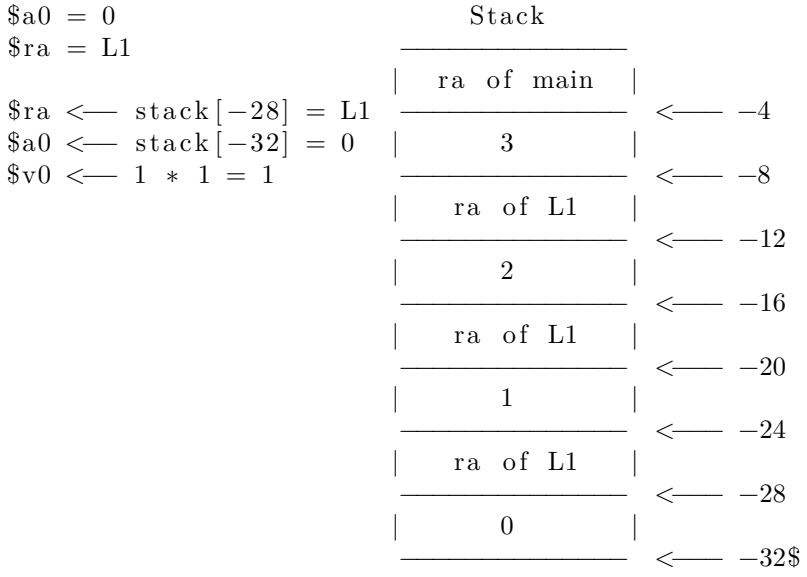
6. During third call to `fact(1)`



7. During third break to `L1`



8. During fourth call to `fact(0)`



9. Backtracking to 3rd `L1`:

```
$ra <— stack[−20] = L1        Stack
$a0 <— stack[−24] = 1     ——————————————
$v0 <— 1 * 1 = 1          |  ra of main  |
                          ——————————————  <—— −4
                          |      3       |
                          ——————————————  <—— −8
                          |   ra of L1   |
                          ——————————————  <—— −12
                          |      2       |
                          ——————————————  <—— −16
                          |   ra of L1   |
                          ——————————————  <—— −20
                          |      1       |
                          ——————————————  <—— −24$
```

10. Backtracking to 2nd `L1`:

```
$ra <— stack[−12] = L1        Stack
$a0 <— stack[−16] = 2     ——————————————
$v0 <— 1 * 2 = 2          |  ra of main  |
                          ——————————————  <—— −4
                          |      3       |
                          ——————————————  <—— −8
                          |   ra of L1   |
                          ——————————————  <—— −12
                          |      2       |
                          ——————————————  <—— −16$
```

11. Backtracking to 1st `L1`:

```
$ra <— stack[−4] = main       Stack
$a0 <— stack[−8] = 3      ——————————————
$v0 <— 2 * 3 = 6          |  ra of main  |
                          ——————————————  <—— −4
                          |      3       |
                          ——————————————  <—— −8$
```

12. Back in main

```
$a0 <— 6
$v0 <— 1
Print 1
Exit
```

If you examine things closely, you can actually forgoe the restoring of `$ra` and `$a0` inside of fact, since those remain unchanged.

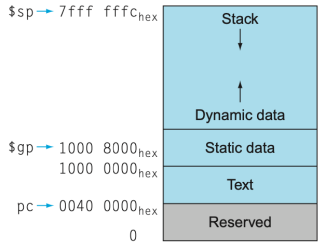


Figure 8. How the memory is split up.

2.8.2 Calling Conventions

Rules for all calls

- The first instruction in the function manipulates `$sp` to allocate F words of stack:

  ```
  addi $sp, $sp, -F*4
  ```

  where $F = A+L+P+$(1 for `$ra` if needed)+(1 for `$fp` if in use)+(2 for `$gp`)+ padding . Also, F is the frame size in words, and should be even.
  Corollary 1: `$sp` is not changed at any other point than the first and last instructions in the function.

- If you want to use any preserved register in any place during the function, you must store them into the stack when you **enter** the function (not when you use it). Let's assume we are saving `$s0`, `$s1`, and `$s2` registers, as well as `$ra` and `$sp`. Then we would write:

  ```
  sw $ra, (F-1)*4($sp)    # Not needed for leaf functions
  sw $fp, (F-2)*4($sp)
  sw $s2, (F-3)*4($sp)
  sw $s1, (F-4)*4($sp)
  sw $s0, (F-5)*4($sp)
  ```

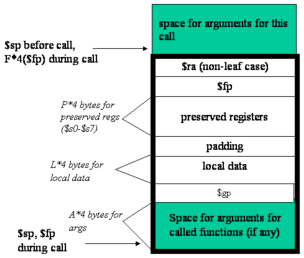


Figure 9. Stack organization during call

- After saving preserved registers, copy `$sp` into `$fp`, so that from now until you start returning, you can use offsets from `$fp` to access the locations. Save and restore local variables as needed after this point, using `$fp` and the spots already allocated for them on the stack.

- You may pass information **into** a function only through `$a0-$a3` and extra stack arguments. As the called function, you may not use the value held in any other register except `$sp`, `$fp`, `$gp` and `$ra` (i.e. don't try and use `$v0` or `$s0`). The 4 (fifth) argument to the current function is available at $(F+4)*4$(`$fp`). You may store argument 0 from `$a0` into its reserved spot at $F*4(\$fp)$, arg1 into $(F+1)*4$(`$fp`), etc., if you want.

- You may pass information **out of** a function through only `$v0-$v1`.

- You may only enter a function at its beginning. You may not `jal` into the middle of a function.

- You may have only one `jr $ra` in each function. It is the last instruction in the function, following the instructions that restore the registers saved.

Extra rules for non-leaf functions:

- Look at the declarations for all procedures this function *might* call. Find the one with the largest number of arguments, A, and then make A at least 4. You must allocate $A \cdot 4$ bytes on the stack upon entry into your function.

- If passing more than 4 arguments, then place 4 arguments into `$a0-$a3`. Every argument past the 4^{th} argument is placed on the stack. For example suppose 6 arguments are in `$s0-$s5`:

  ```
  move $a0, $s0    # 0($fp) reserved on the stack
  move $a1, $s1    # 4($fp) reserved on the stack
  move $a2, $s2    # 8($fp) reserved on the stack
  move $a3, $s3    # 12($fp) reserved on the stack
  sw $s4, 16($fp)  # 5th arg in the stack. Always here!
  sw $s5, 20($fp)  # 5th arg in the stack. Always here!
  ```

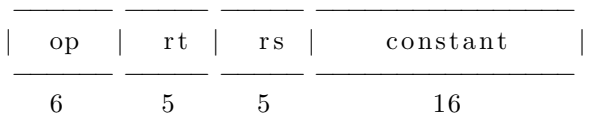
- You must preserve register `$ra` because a called-function `jal` destroys the value, and it is a preserved register.

Subsection 2.10

MIPS Addressing for 32-bit Immediates and Addresses

2.10.1 32-bit Constant into Register

If you have been following closely, you may have noticed that when adding a constant to a register there are only 16 bits available for the constant (as it is an I-type instruction):



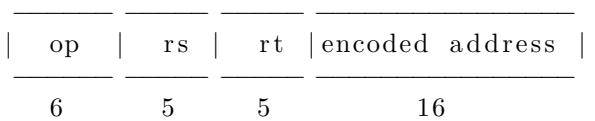
The question then remains, what do you do if you want to place a constant larger than 16 bits? You achieve this with the following two instructions:

1. `lui rt, immediate`
   which loads immediate value into the upper 16 bits of `rt` and zeros lower 16 bits.
2. `ori rt, rs, immediate`
   Bitwise or of `rs` and `immediate` placed into `rt` (`rt <- rs | imm`).

A word of caution. Using the instruction `addi` sign extends the msb of immediate.

2.10.2 Branch Addressing

Branch instructions take are I-type instructions:



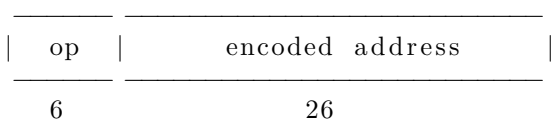
The target of the encoded addres is

$$\text{Target} = 4(\text{encoded address}) + \texttt{PC} + 4.$$

That is, branch instructions are word addressed (4(encoded address)), and relative to the next instruction ($\texttt{PC} + 4$).

2.10.3 Jump Addressing

Jump instructions are J-type instructions:



In this case the lower 28 bits of the address to be jumped to are

$$4 \cdot \text{encoded address}$$

and the upper four are that of the `PC`. If you need to jump to a really far away address (more than 64 million instructions away), then you use jump register instruction (`jr`).

Subsection 2.12

Translating and Starting a Program

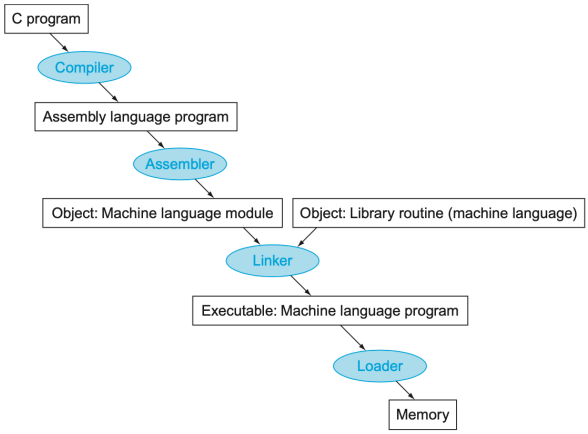


Figure 10. The four steps that go into loading your C program into memory.

2.12.1 Compiler

The compiler transforms the C program into an assembly language program.

2.12.2 Assembler

The main job of the assembler is to convert assembly language into binary machine code (object file). The assembler must determine the addresses associated with labels, which it keeps track of in a symbol table which contains pairs of symbols and addresses. The object for UNIX systems typically contains six distinct pieces:

- The *object file header* describes the size and position of the other pieces of the object file.
- The *text segment* contains the machine language code.
- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use both static data, which is allocated throughout the program, and dynamic data, which can grow or shrink as needed by the program.)
- *The relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The *symbol table* contains the remaining labels that are not defined, such as external references.
- The *debugging information* contains concise descriptions of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

2.12.3 Linker

The linker or link editor is a systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file. In this way, you can combine your program with other progras (such as header files or libraries) without having to recompile everything together.

2.12.4 Loader

The loader places an object program in main memory so that it is ready to execute. The loader follows the following steps in UNIX systems:

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory.
4. Copies the parameters (if any) to the main program onto the stack.
5. Initializes the machine registers and sets the stack pointer to the first free location.
6. 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.

Subsection 2.14

Arrays versus Pointers

Let's look at two ways we can set an array to all zeros: one using pointers, and one using arrays, so that we get an idea for how C pointers work.

2.14.1 Array

Let's say that we have the following C code:

```
1  clear1(int array[], int size)
2  {
3      int i;
4      for (i = 0; i < size; i +=1)
5          array[i] = 0;
6  }
```

Then we would have the following MIPS code:

```
1    addiu  $sp, $sp, $-16       # Space for four arguments
2    addi   $sp, $sp, $-32       # Space for array of 8 words
3    addi   $sp, $sp, $-4        # Space for $gp
4    addi   $sp  $sp, $-4        # space for $fp
5    addi   $sp  $sp  $-4        # Space for $ra
6
7  F = 4(arguments) + 8(local data) + 1(ra) + 1(fp) + 2(gp)
8    = 16
9  Stack
10  ---------------
11 |      ra       |
12  --------------- <---- +60
13 |      fp       |
14  --------------- <---- +56
15 |  array[7]     |
16  --------------- <---- +52
17 |  array[6]     |
```

```
18   ---------------- <---- +48
19  |   array[5]     |
20   ---------------- <---- +44
21  |   array[4]     |
22   ---------------- <---- +40
23  |   array[3]     |
24   ---------------- <---- +36
25  |   array[2]     |
26   ---------------- <---- +32
27  |   array[1]     |
28   ---------------- <---- +28
29  |   array[0]     |
30   ---------------- <---- +24
31  |                |
32   ---------------- <---- +20
33  |       gp       |
34   ---------------- <---- +16
35  | arg 3 spce     |
36   ---------------- <---- +12
37  | arg 2 spce     |
38   ---------------- <---- +8
39  | arg 1 spce     |
40   ---------------- <---- +4
41  | arg 0 spce     |
42   ---------------- <---- +0
43
44  main:
45    addiu   $sp, $sp, -64         # Allocate stack space
46    sw      $ra, 60($sp)          # Save return address
47    sw      $fp, 56($sp)          # Save frame pointer
48    sw      $gp, 16($sp)          # Save global pointer
49    add     $fp, $sp, $zero       # Estabilish frame pointer
50    addiu   $a0, $fp, 24          # arg0 <---- &array[0]
51    addiu   $a1, $zero, 8         # arg1 <---- size = 8
52    jal     clear1                # clear1(&array, 8)
53    j exitclear1:add t0,zero, zeroi <--0slti t3, a1, 1t3 <- 1 if size < 1bne t3,zero,
      exit  if (size < 1) exitloop:sll t1,t0, 2  t1 <--i * 4add t2, a0,t1  t2 <--
      memaddressofarray[i]swzero, 0(t2)array[i] <--0addit0, t0, 1i+ = 1slt t3, t0,a1
      t3 <--i, ifi < sizebnet3, zero, Loopgotoloopifiexit :
```

Note that this code only works if `size > 0`; ANSI C requires a test of size before the loop, but we'll skip that legality here.

Section 3

Arithmetic for Computers

Subsection 3.2

Addition and Subtraction

Addition in binary works just like addition in decimal, going bit by bit, and carrying over:

$$\begin{array}{rl} & 10\_\{10\} = 01010\_2 \\ + & 6\_\{10\} = 00110\_2 \end{array}$$

$16_{10} = 10000_{2}$

Subtraction can either be done via the subtract operation, or by adding a negative number (in two's complement):

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

or via addition using the two's complement representation of -6:

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

3.2.1 Overflow Twos Complement

Addition

- Cannot occur when adding operands with different signs. This is because the sum can be no larger than one of the numbers, and both numbers fit.
- Overflow can be detected when adding numbers of the same sign but the sign bit changes.

Subtraction

- Cannot occur when signs are the same ($a - b = a + (-b)$).
- Overflow occurs when we subtract a negative number from a positive number but get a negative number ($a - (-b) = -x$)

3.2.2 Unsigned Overflow

Depending upon the situtation, one may or may not want an exception to be thrown when overflow occurs. For this reason there are there are two types of integer opporations.

- `add`, `addi`, and `sub` cause exceptions on overflow.
- `addu`, `addiu`, and `subu` do not cause exceptions on overflow.

Note that allthough `addiu` is unsigned addition, the 16-bit immediate field is sign extended to 32-bits and so the immediate field is signed, even if the operation is "unsigned".

Note that C ignores overflows, and so MIPS compilers will always generate the unsigned versions.

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
                             # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                  # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs ≠; goto overflow
```

For unsigned addition (`$t0 = $t1 + $t2`), the test is

```
addu $t0, $t1, $t2     # $t0 = sum
nor $t3, $t1, $zero    # $t3 = NOT $t1
                       # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2     # (2^32 - $t1 - 1) < $t2
                       # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1<$t1+$t2) goto overflow
```

Subsection 3.3
Multiplication

Let's say we want to multiply 1011×0101, then we would perform the following steps:

$$\begin{array}{rl} 1011 & \textit{Multiplicand} \\ \times\ 0101 & \textit{Multiplier} \\ \hline 1011 & \\ 0000 & \\ 1011 & \\ +\ 0000 & \\ \hline 00110111 & \textit{Product} \end{array}$$

Notice that we just add copies of the multiplicand (shifted over by 1 each time), or zero (depending upon bit of multiplier). Thus, to perform multiplaction, we will need the following pieces of hardware:

- A multiplicand register. Because we are shifting over the multiplicand each time, it needs to be twice as big (so 64-bits).
- A multiplier register. Size of 32-bits.
- A product register. Needs to be 64 bits wide.
- An Arithmetic Logic Unit (ALU) to perform the addition of product and multiplicand registers.
- A control test that in each cycle tells:
  - the ALU to sum the multiplicand and product (if control says to)
  - the multiplicand to shift left,
  - the multiplier to shift right,
  - the product register to change or not depending upon the lsb if the multiplier register.

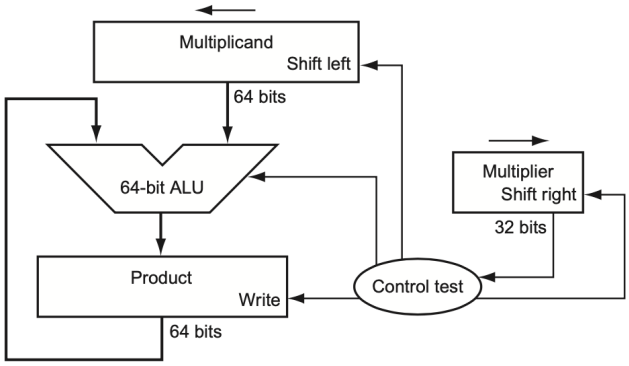


Figure 11. A simple version of multiplication hardware.

So what will the first cycle of hardware look like?

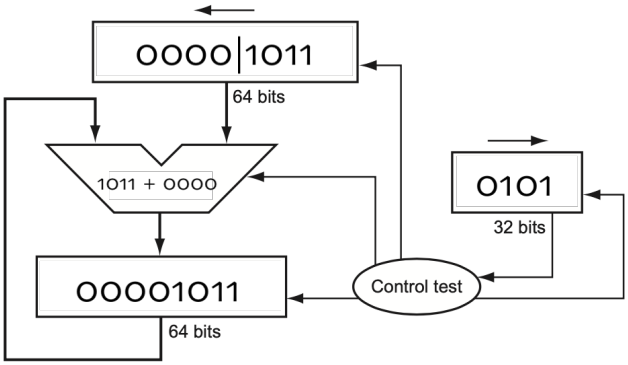


Figure 12. So at first, the product register is set to 0000, the multiplicand register is set to 1011, and the multiplier register is set to 0101. The control test tests the first bit of the multiplicand (1), and so tells the ALU to sum the product and multiplicand registers, and store the value in the product register. The control then shifts the multiplicand 1 bit to the left, and the multiplier 1 bit to the right.

3.3.1 Faster Multiplication

Parallel Processing Speedup

- The multiplier and multiplicand are shifted while the multicland is added to the product if the multiplier bit is a 1.
- The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand.

Resource Optimization Speedup If you look back at doing the multiplication by hand, you may notice that at each step you are really only adding four bits at once. What we will do now, is have a 32-bit multiplicand register, a 32-bit adder, and a 64-bit product, and shift the product register instead of the multiplicand register. Let's se how this works in practice.

- **Step 1**
  Multiplicand: 1011
  Multiplier: 0101
  Product: 1011 0000
- **Step 2**
  Multiplicand: 1011
  Multiplier: 0101
  Product: 0101 1000
- **Step 3**
  Multiplicand: 1011
  Multiplier: 0101
  Product: 1101 1100
- **Step 4**
  Multiplicand: 1011
  Multiplier: 0101
  Product: 0110 1110
  The last shift will then set the product as 0011 0111.

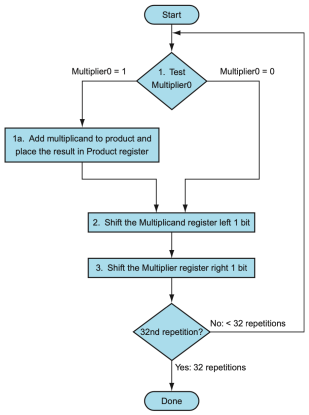


Figure 13. The multiplication algorithm.

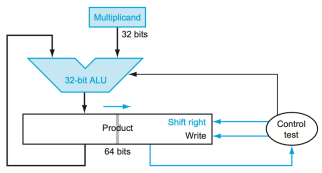


Figure 14. The faster multiplication hardware, taking advantage of parallel processing as well as reduced register size

3.3.2 Even Faster Multiplication

SUBSECTION 3.4

Division

Let's first examine a little bit more closely the long division from elementary school:

$$\begin{array}{r} 1001 \\ 1000\overline{)1001010} \\ \underline{1000} \\ 1010 \\ \underline{1000} \\ 10 \end{array}.$$

Notice how as we are testing to see the divisor is less than the n MSBs of the dividend where n increases by 1 each cycle. If the divisor is less than, we subtract the divisor from the dividend and put 1 in the quotient.

So how does all of this transfer over to hardware?

- Well, so that we are comparing divisor to the MSBs of the divident we will store the 32-bit divisor in a 64-bit integer, but start with it in the 32 MSBs. This way, after each cycle we just shift the divisor to the right one.
- Because we keep subtracting from the dividend, we just make our life simple and start the dividend as the remainder.
- Instead of placing the 1 or 0 of the quotient starting from the LSB, we place start with the quotient as zero and shift the quotient left 1 and set LSB to 1 if applicable. This way at the end this will get shifted over to become the MSB.

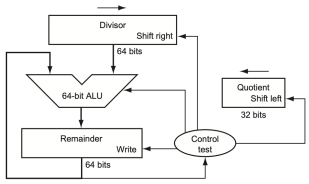


Figure 15. Hardware implementation of division

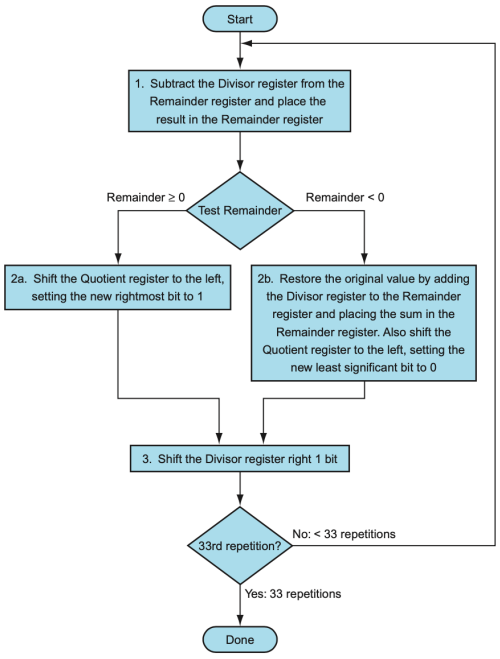


Figure 16. The division algorithm that we will implement in hardware.

3.4.1 Faster Division

We can make this a little bit more efficient. Instead of shifting the divisor right, we can shift the remainder/quotient left. So we start with the quotient in the 32 LSBs, shift the remainder left one cycle and compare the 32 MSBs of the remainder with the divisor (we want the divisor to be less than the upper 32 MSBs of the quotient/remainder). In this way we now only need a 32 bit ALU.

At the end of this you may notice that the 32 LSBs of the quotient/remainder are all zeros. To utilize this space we will use this as the spot for the remainder to go, saving us an extra 32 bits.

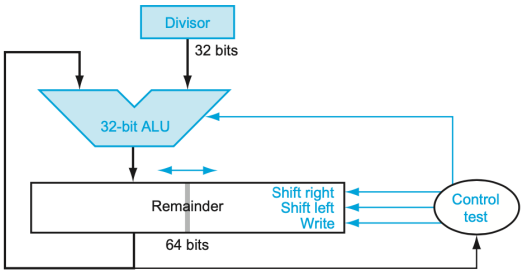


Figure 17. The faster division hardware

Section 4

Basics of Logic Design

For the most part we will be focusing on *combinational logic*. This is logic that has no memory (state) and therefore

$$\text{Same Input} \iff \text{Same Output}\ .$$

Subsection 4.1

Truth Tables

Because of the above fact, a combinational block can be completely described by a truth table. In a logic block an input is either true or false. Therefore, a logic block with n inputs has 2^n possible different possibilities.

Let's coonsider the logic function with inputs (A, B) and outputs (C, D, E) where:

- C is true if no inputs are true,
- D is true if at least on input is true, and
- E is true if all inputs are true.

The truth table would be defined as follows

Inputs		Outputs		
A	B	C	D	E
0	0	1	0	0
0	1	0	1	1
1	0	0	1	0
1	1	0	1	1

.

Sometimes truth tables are hard to read, so we can express each output of the truth table as a boolean expression:

$$C = \bar{A} \cdot \bar{B}$$
$$D = A + B$$
$$E = A \cdot B.$$

Subsection 4.2
Boolean Algebra

There are three elementary operators:

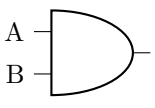
- logical OR: $A + B$,
- logical AND: $A \cdot B$, and
- logical NOT: $\bar{A}$.

4.2.1 Laws

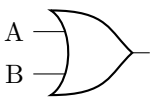
$$\begin{aligned}
\text{Identity Laws: } & A + 0 = A \\
& A \cdot 1 = A \\
\text{Zero and One Laws: } & A + 1 = 1 \\
& A \cdot 0 = 0 \\
\text{Inverse Laws: } & A + \bar{A} = A \\
& A \cdot \bar{A} = 0 \\
\text{Commutative Laws: } & A + B = B + A \\
& A \cdot B = B \cdot A \\
\text{Associative Laws: } & A + (B + C) = (A + B) + C \\
& A \cdot (B \cdot C) = (A \cdot B) \cdot C \\
\text{Distributive Laws: } & x \cdot (y + z) = (x \cdot y) + (x \cdot z) \\
\text{DeMorgan's Laws: } & \overline{(A + B)} \Leftrightarrow \bar{A} \cdot \bar{B} \\
& \overline{(A \cdot B)} \Leftrightarrow \bar{A} + \bar{B}.
\end{aligned}$$

Subsection 4.3
Logic Blocks

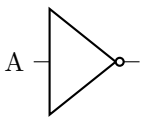
We can implement truth tables and boolean expressions with the use of three types of logic gates:



AND gate

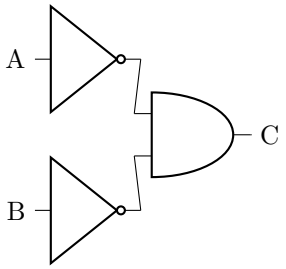


OR gate



NOT gate

For example, we can build function $C = \bar{A} \cdot \bar{B}$ as follows:



4.3.1 Decoder

A decoder is a logic block that has n-bit input and 2^n outputs, where only one output is asserted for each input combination. For example:

Inputs		Outputs			
A	B	x_0	x_1	x_2	x_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	1	0	0	1

We can construct a 2 in, 4 out decoder using AND gates:

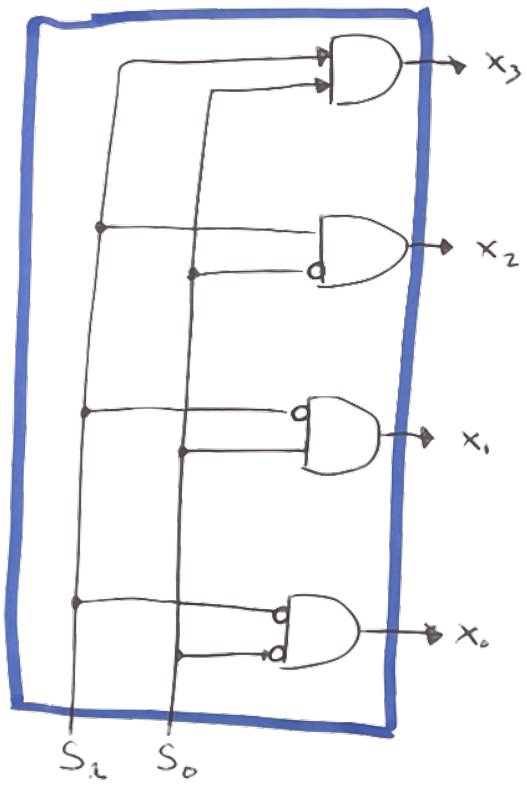


Figure 19. Notice how only one output will be asserted, all others will be deasserted.

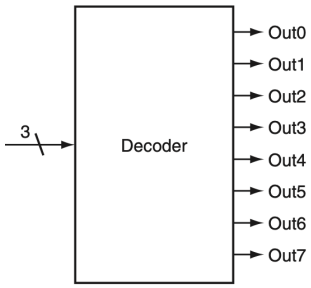


Figure 18. A 3 bit decoder

4.3.2 Multiplexer

A multiplexer is selects one of the given inputs based upon the select signal given. A multiplexer has three parts:

1. A decoder that generates n signals (from $\log_2 n$ inputs), each indicating a different input value
2. An array of n AND gates, each combining one of the inputs with a signal from the decoder
3. A signle large OR gate that incorporates the outputs of the AND gates.

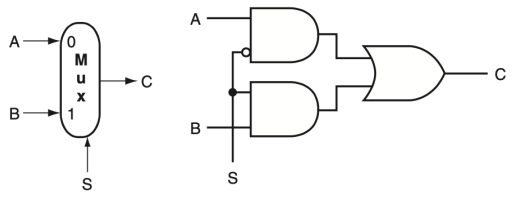


Figure 20. We can see either A or B is seleted based upon if the value of S is 0 or 1.

The truth table for the above MUX will be

index	S	C
0	0	A
1	1	B

,

and in boolean algebra:

$$C = (\bar{S} \cdot A) + (S \cdot B).$$

4.3.3 Two-Level Logic

Any logic function can be written where every **input** is either true or a complemented variable and there are only two levels of gates:

- an AND level and
- an or level.

There may be a possible inversion of the final output. This is a lot to unpack, so let's look at an example.

Let's say we are given the following truth table for D:

Inputs			Outputs
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

.

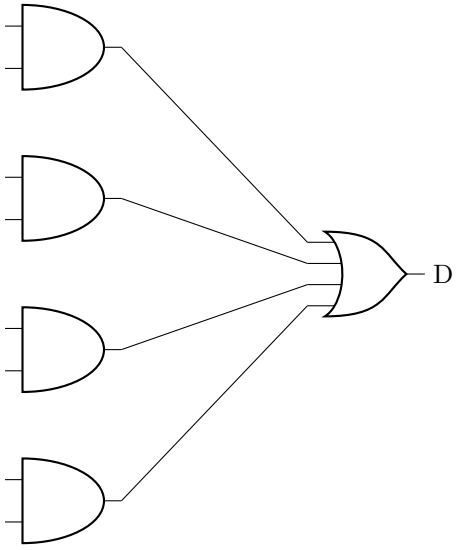
A **product term** or a **minterm** is a set of logical inputs joined by conjunction (AND operations); the product terms for the first logic stage of the PLA.

Notice that there are four input combinations for which D is true:

- $\bar{A} \cdot \bar{B} \cdot C$
- $\bar{A} \cdot B \cdot \bar{C}$
- $A \cdot \bar{B} \cdot \bar{C}$

- $A \cdot B \cdot C$

We call each of these terms a **product term**. For example the function D can be implemented in hardware using two-level logic as follows:



4.3.4 Programmable Logic Array

When dealing with a set of logic functions (multiple inputs and multiple outputs) there will be an

- AND gate for each unique set of inputs for which there is at lest one corresponding true output
- OR gate for each output function.

This way of structured-logic implementation is called a programmable logic array. Let's say we have the following logic table:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

.

Then the associated PLA that implements the logic function is:

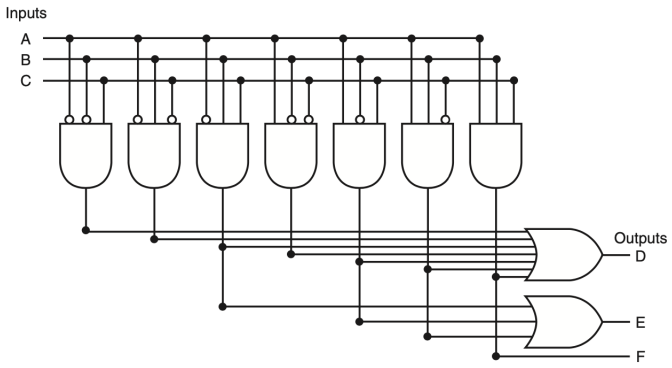


Figure 21. Notice that only 7 AND gates are required because no function is true for $A = B = C = 0$, and so we can ignore that. Also note that although there are three functions, there are only two OR gates because only combination of inputs for turns F on.

4.3.5 Don't Cares

Don't cares, as the name suggests, are values of an input, or values of an output for which we don't care what the value is; the value could be either 0 or 1 and the effect of the logic block would be unchanged.

- Input Don't Cares: Arise when an output depends on only some of the inputs
- Output Don't Cares: Arise when we don't care about the value of an output because another output is true.

Don't cares are shown as X's on a truth table. They are important because they make it easeir to implement the optimization of a logic function. For example, consider the following logic functions:

- D is true if A or C is true
- E is true if A or B is true
- F is true if exactly one of the inputs is true. But we don't care about F if both D and E are true

The truth table is as follows:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

.

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

.

Figure 22. Notice that with the use of don't cares, we can have the truth table be smaller because of repeated rows.

4.3.6 Arrays of Logic Units

Up until now we have been performing combinational operations on just single bits, but we often want to do the same operations on whole words (32-bits). For example you may want to see if the value in one register is the same as the value as the other. Or you may want to have a MUX that selects between two words, and not just two bits:

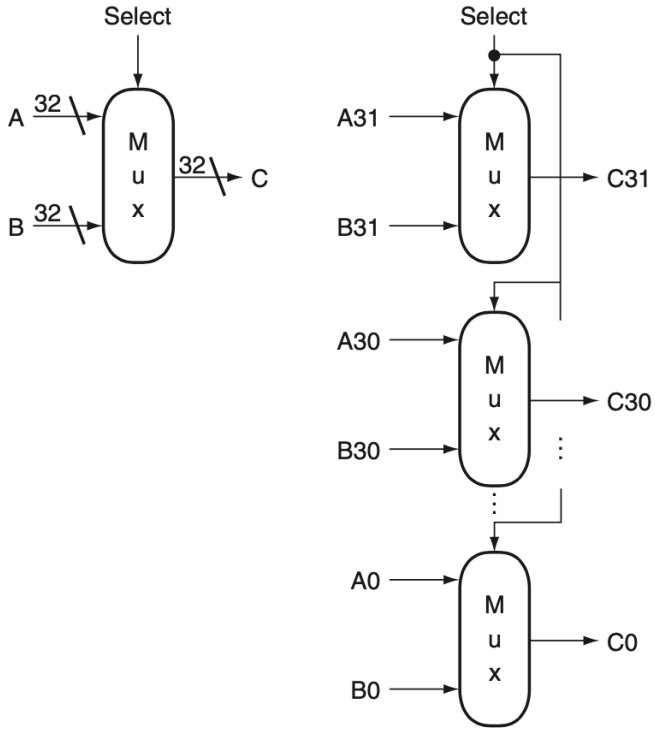


Figure 23. The 32-bit MUX is just 32 1-bit MUX's connected together.

Section 5

Constructing a Basic Arithmetic Logic Unit

An ALU is the part of the processor that is responsible for performing both logical and arithmetic operations such as addition, subtraction, AND, and OR.

Subsection 5.1

1-Bit ALU

Let's go about implementing each of the different operations an ALU can perform.

5.1.1 AND / OR Logical Operation

To implement this you just have a MUX that selects between the AND, and OR operations:

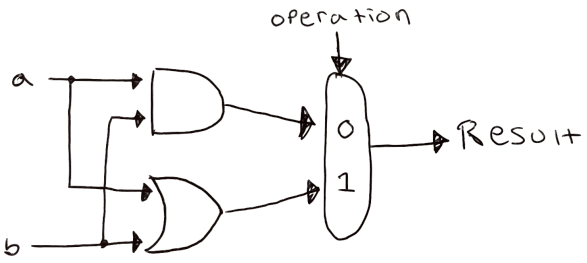


Figure 24. The multiplexor selects between and or or depending upon the opcode recieved.

5.1.2 Addition

To implement addition, with each bit in addition to looking at the bits adding together we also want to see if there was carryover from the the prior (less significant) bit. Thus, to deploy our operation in hardware we will need three **inputs**:

- a
- b
- carry in

and two **outputs**:

- computed sum
- carry out.

How is the carry out computed? Well, there is carry out if 2 or 3 of the inputs are true. Thus we have

$$\begin{aligned}\text{Carry Out } &= (a \cdot b) + (a \cdot \text{ carry in}) + (b \cdot \text{ carry in}) + (a \cdot b \cdot \text{ carry in}) \\ &= (a \cdot b) + (a \cdot \text{ carry in}) + (b \cdot \text{ carry in}).\end{aligned}$$

Implementing this using logic gates we have:

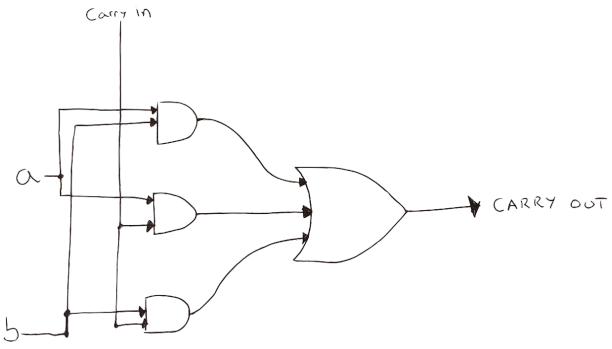


Figure 25. For cary out to be true we need to have t least two of the inputs to be true.

To calculate the **sum** using logic we see that sum will be true if an odd number of inputs are true (1 or 3). Therefore:

$$\text{sum } = (a \cdot \bar{b} \cdot \overline{\text{carry in}}) + (\bar{a} \cdot b \cdot \overline{\text{carry in}}) + (\bar{a} \cdot \bar{b} \cdot \text{ carry in}) + (a \cdot b \cdot \text{ carry in}).$$

Implementing this using logic gates we have:

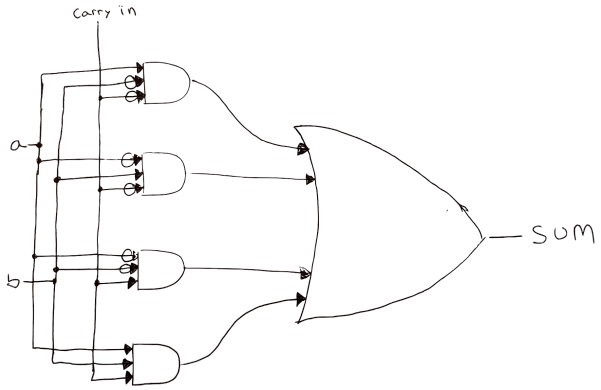


Figure 26. Sum is true if there are an odd number of true inputs.

5.1.3 Completing the 1-Bit ALU

To complete the 1-bit ALU we just connect the AND/OR/ADD operations to a mux that selects the desired operation.

Subsection 5.2

32-Bit ALU

The 32-bit ALU is created by connecting 32 1-bit ALU together. Although we have our 1-bit ALU, there are some additional features that we want to make sure we have.

5.2.1 Subtraction

To compute subtraction we will use the addition we have already implemented and add a negative number. We can do this because $-b = \bar{b} + 1$. To get $\bar{x}$ we thus need an inverter that inverts each of the bits of b. We need to connect that inverter to a MUX which selects the inverted b (if subtraction) or the original b (if addition). We will add one through the use of the carry in input of the lsb since that will be 0 when addition and 1 when subtraction.

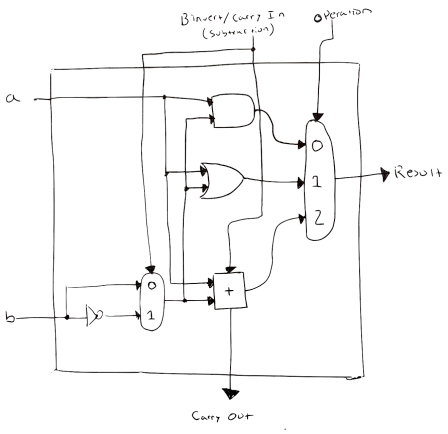


Figure 27. For subtraction we use the value of carry in to additionally select whether we should invert b or not. The operation code selects between AND, OR, and ADD (which also performs subtraction depending upon value of carry in).

5.2.2 NOR (Neither a nor b)

Because

$$(\overline{a + b}) = \bar{a} \cdot \bar{b}$$

we are able to utilize the existing Binvert and AND gate and only need to add an inversion of a.

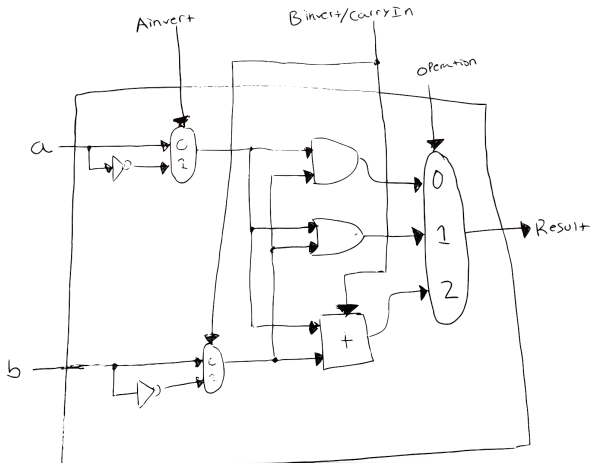


Figure 28. For NOR the output of the AND gate is used but with both Ainvert and Binvert inputs set as true.

5.2.3 Shift Less Than

For `slt` we return 1 if `rs < rt` and 0 otherwise. To test for this we can use subtraction because

$$a < b \Leftrightarrow (a - b) < 0$$

and thus if the result of $a - b$ is negative we set the lsb to 1. How does this work in practice? Well the 31 most significant ALUs are easy, we just connect 0 directly to the MUX that selects the desired operation. We will label this input `sltResult`. For the least significant bit we connect the result of the adder of the msb to the least significant bit input `sltResult`.

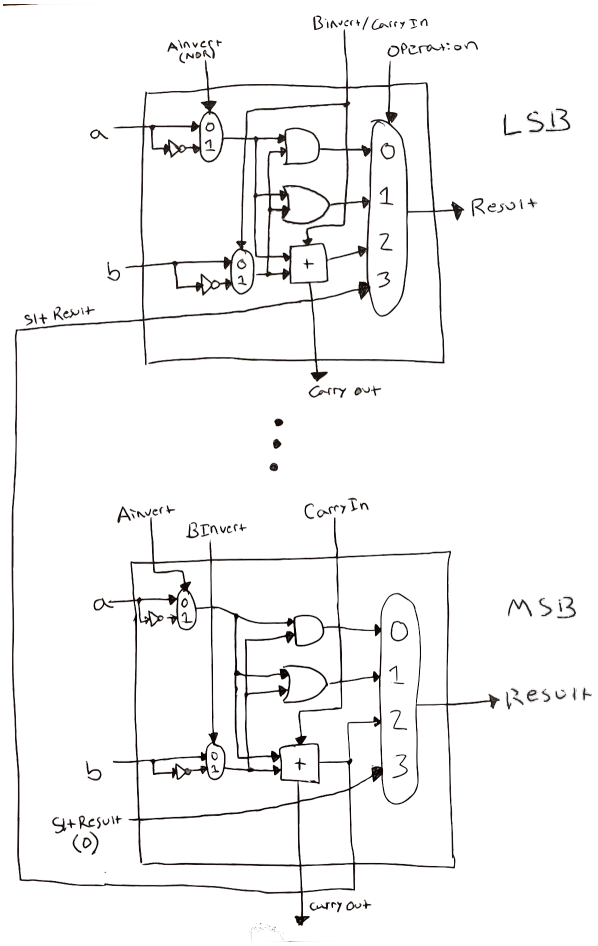


Figure 29. To perform `slt` we set operation to 3, Binvert to 1 and `sltResult` to 0 for the 31 MSBs but use the result of the adder for the MSB to set the value of `sltResult` of the LSB

Section 6

Memory Elements

In addition to providing combinational logic, we also want our processor to have state. Let's look at how we can creatae memory elements.

Subsection 6.1

S-R Latch

One of the main building blocks of memory is a **pair of cross-coupled NOR gates**. Let's say you have two lights, and you wanted it so that turning one one would turn the other off and vice-versa. You could use these cross coupled NOR gates to achieve this.

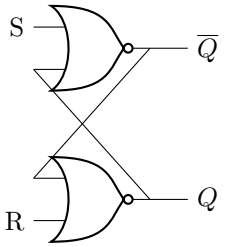


Figure 30. We have two inputs R (Reset) and S (Set). If you activate R and not S then the output of the top NOR gate will be 0 (since one of the inputs is 1). This will then connect with the bottom NOR gate and thus $\bar{Q} = 1$. The reason that the output of one gate can be the input of another is that you wait until the system is stable (sice for same inputs you will always have same outputs in a combinational circuit)

Inputs		Outputs	
S	R	Q	$\bar{Q}$
0	0	X	X
0	1	0	1
1	0	1	0
1	1	!	!

Figure 31. When both S and R are deasserted then the value of Q and $\bar{Q}$ remain unchanged. When both are asserted, then undefined behavior is the result. When only S is asserted then $Q = 1$ and when only R is asserted then $\bar{Q} = 1$.

Subsection 6.2

Latches

6.2.1 Transparent Latch

A transparent latch is one which immediately responds to a change in the input. In this way it is an asynchronous memory element.

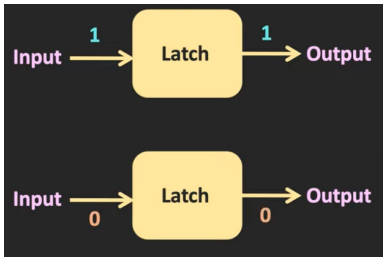


Figure 32. A transparent latch will immediatly respond to a change in input. So in this example the input of a latch changes from 0 to 1 and that change is immediatly reflected in the output.

Notice that the output is the same as the input. This is because the output of a latch is the same as the memory value that is stored inside.

6.2.2 Gated Latch

A gated latch is a latch whose output only changes to changes in input (becomes transparent) when the clock is high.

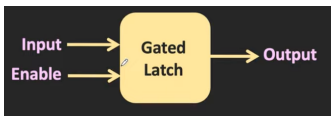


Figure 33. The gated latch has an additional "enable" input. This is asserted when the clock level is high and deasserted when not.

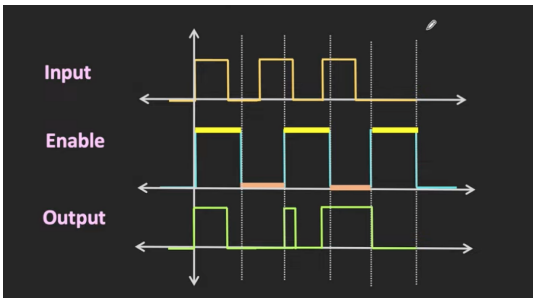


Figure 34. We see that in a gated latch that changes to input when the clock is low are not affected in the output, and that the output only changes when the clock is high.

We can utilize our S-R latch to build a gated D latch. What is our goal? We want our output Q to reflect our data input D only when our clock C is asserted. Therefore, we will have two AND gates with C as an input of each. If we connect the output of these AND gates to the input of our NOR gates then that means that if C is deasserted then the the inputs of our SR Latch will be $(0, 0)$ and nothing will happen. If we connect D with the AND gate going into the set input of our SR Latch, then that will set $Q = 1$. Similarily if we have the inverse of D going into the AND gate whose output is the input for the reset, then that will turn $Q = 0$.

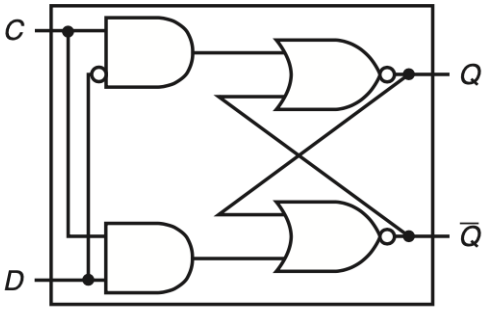


Figure 35. Implementing a D Latch using logic gates.

6.2.3 Flip-Flop

A flip-flop is very similar to a gated latch except that it becomes transparent only on the rising edge or falling edge of the clock, depending upon if it is *positive-edge triggered* or *negative-edge triggered*.

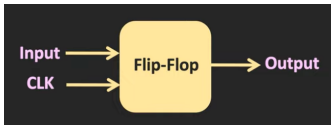


Figure 36. A flip-flop also accepts the clock signal as an input, and is transparent only on the rising or falling edge (exclusive or).

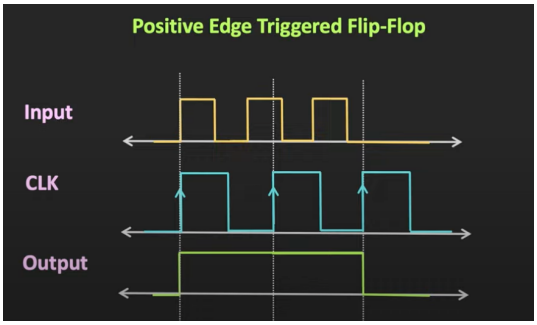
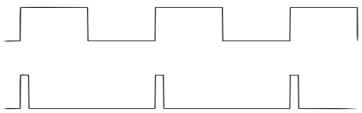


Figure 37. We see that the output only is changed when there is a rising clock edge.

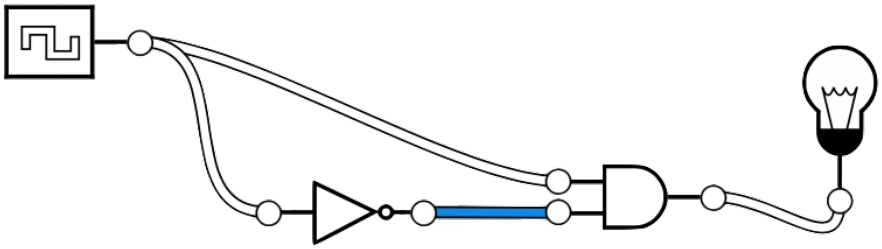
How can we implement a Flip-Flop? Well what if we had some kind of circuit that took the clock signal and converted the rising (or falling) edge into little blips:



Well, then we can use our D-Latch implementation, because it will only be transparent on rising edges (since E will only be asserted for a very short time). How can we implement this?

Making Use of Inverter Delay

Well one way is with the use of AND gate whose inputs are D and D passing through a NOT gate:



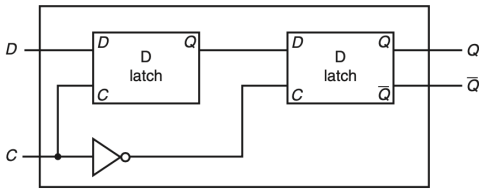
You may think that the light bulb will never turn on, and you would almost be correct. But because it takes a little bit of time for the signal to pass through the inverter, when the clock goes from low to high (rising edge), then both inputs of the and function will be 1 for a split second and the lightbulb will turn on.

Master-Slave Flip Flop

Let's say you want to design a negative-edge triggered D flip flip. Well, if we constrain ourselves to having the data input being stable for the duration of the rising and falling edge of the clock cycle (and a bit before and after), then we can create a flip flop by connecting two D-latches together:

- The first latch will be transparent on a high clock signal and take in D as the input setting Q. The output Q will then connect to another D latch.
- The second latch will be transperent on a low clock signal and take in D as the input (passed through the first D-Latch).

In this way the flip flip will only update on a falling-edge trigger.



6.2.4 Register Files

A register file consists of a set of registers that can be read and written to by supplying a register number to be accessed.

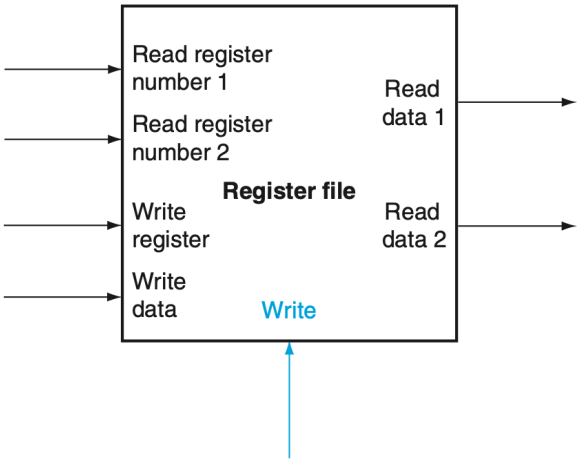
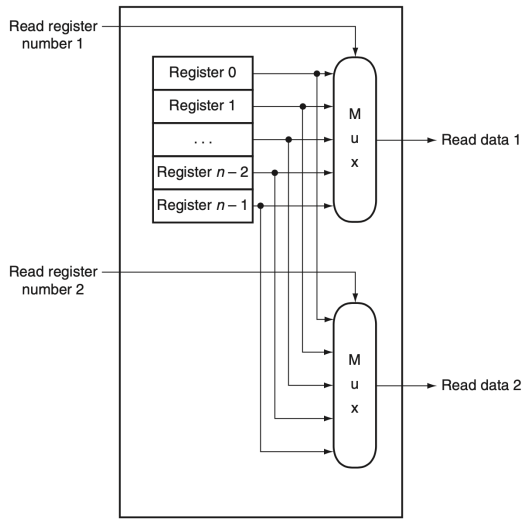


Figure 38. A register file needs two read register inputs (one for each register that will be read), an input that says what register should be written to, an input that says what data should be written, and a control input specifiying if a write operation will be performed.

The Read Port

Implementing the read port is rather simpl. You can have all n registers connected to two different $n \to 1$ MUXs, which use the read register numbers to select the register to be read from:



The Write Port

In the read port, we could "read" from all the registers and select only the one that we want using a MUX. For the write port though, we can only write to one register, and so the logic is a little bit more complicated.

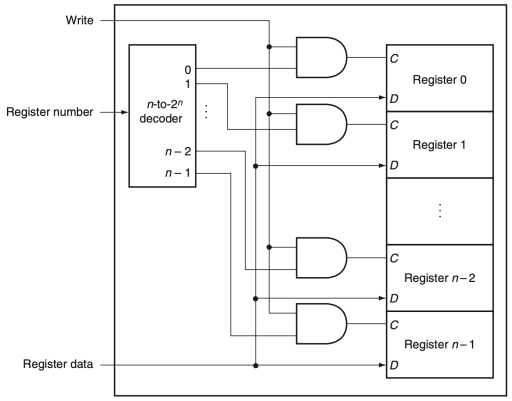


Figure 39. The main part of the write port is a decoder where n is the number of bits needed to describe the number of registers there are. The decoder will only assert the given register, thus data will only get written to the desired register and none of the others (through the use of the AND gates).

Definition 4

D Latch: A latch with one data input (called D) that stores the value of that input signal in the internal memory when the clock is high.

D Flip-Flop: A flip-flop with one data input (called D) that stores the value of that input signal in the internal memory when the clock edge occurs.

Section 7

The Processor

Subsection 7.1

Introduction

We will be building a simplified RISC processor that implements a subset of the RISC ISA. We will be implementing instructions related to:

- Memory-Reference: `lw` and `sw`
- ALU: `add`, `sub`, `AND`, `OR`, and `slt`
- Branch: `beq` and `j`

What does a processor look like?

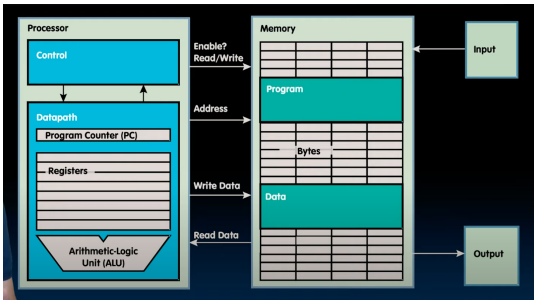


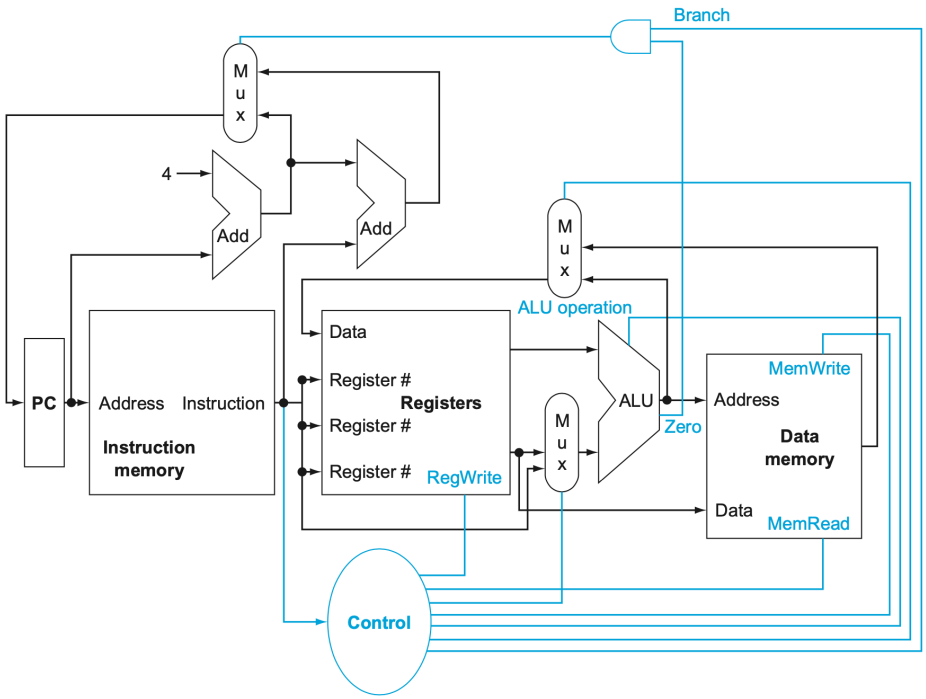
Figure 40. A big picture view of a processory

What are the two main components of a CPU:

Processor (CPU): This is the active part of the computer that does all the work (data manipulation and decision-making)

- Datapath: This is the portion of the processor that contains the hardware necessary to perform operations required by the processor. This is the brawn of the processor. We could build a sperate datapath for every instruction. Instead, we build one datapath that operates many instructions.
- Control: Another hardware portion of the processor that tells the datapath what needs to be done (the brain)

How does a processor perform our instructions? Well, let's look at a big picture view into a processor:



What is going on here?

- The first thing that happens is we load the instruction using the address stored in the PC. Recll that the PC stores the address of the next instruction to be performed.
- The address from the PC also gets sent to an adder to load the next instruction.
- But if the instruction is a branch instruction then the PC needs to reflect that, so thus the MUX.
- The register fields from the instruction are then sent to the register file giving us the appropriate register operands which can then can be used to perform our operation.
- The operation could be to compute a memory address (for load or store), or to perform an ALU operation.
- If it is a memory store operation, then the appropriate value is stored in memory.

- If it was an arithmetic operation or a memory load operation, the value is then written back to the appropriate register.
- Note that the ZERO output of the ALU is used on `beq` instructions.

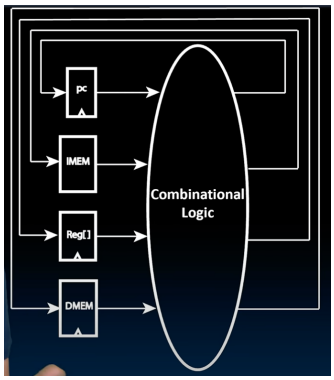


Figure 41. The only state elements are the `PC`, the instruction memory, the register file, and the data memory. These state elements connect to the combinational logic blocks of our processor whose result is then fed back into our state elements.

Subsection 7.2
Logic Design Conventions

The processor is a combination of state and combinational logic.

- Instead of thinking about the combinational logic as one big bubble, we break it up into multiple blocks of logic as that is a lot easier to manage.
- A state element has at least two inputs (and one output). The inputs are the data value to be written and the clock, which determines when the value is to be written.

7.2.1 Clocking Methodology

Because we are reading from our state elements, performing combinational logic, and then writing back into the same state elements it is really important that we carefully control when our state elements are allowed to be read and written to. Because otherwise we would get undefined behavior (maybe reading the new value instead of the old or some combination of the new and old value).

We have looked at different ways in which we can control the time in which state changes are allowed. For now, we will assume that all state changes are **edge triggered**.

Subsection 7.3
Building a Datapath

7.3.1 Reading an Instruction

To read an instruction we need to have the program counter (`PC`) as an input to the instruction memory, from where the instruction is outputted. We also need to update the PC which we can do using an adder.

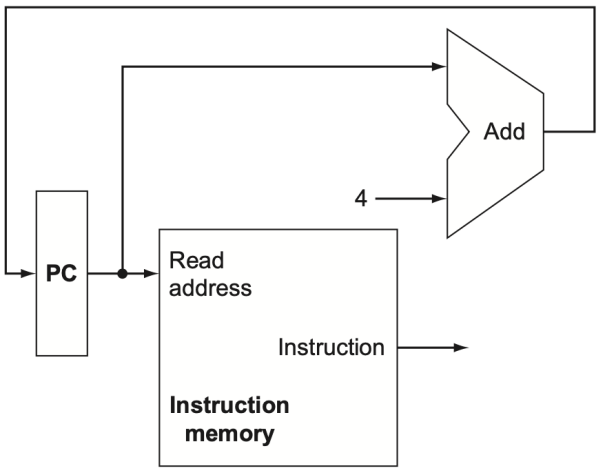


Figure 42. The part of the datapath responsible for fetching the instruction and incrementing the `PC`.

7.3.2 R-Type Instructions

Let's say that after reading our instruction, it turns out to be an R-type instructios. These will be our ALU instructions `add`, `sub`, `or`, `and`, and `slt`. For these instructions we will be reading from two registers, performing an ALU operation and writing back to a third register.

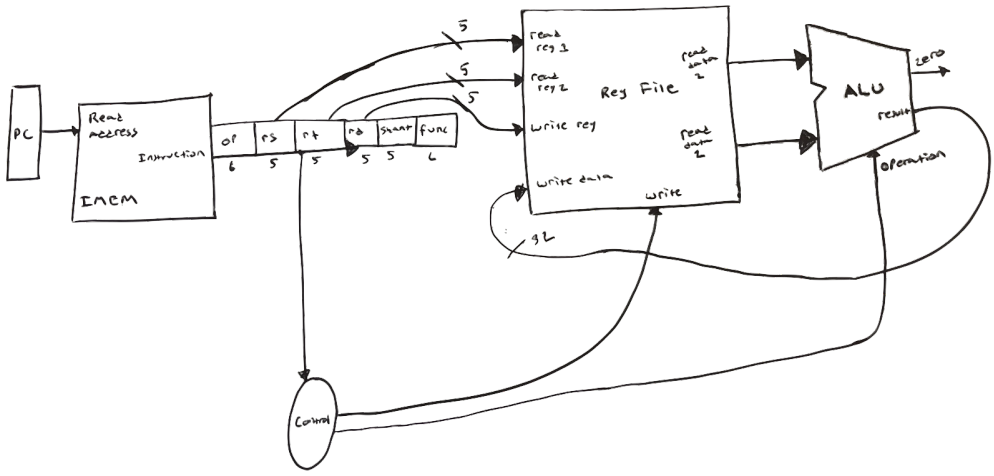


Figure 43. We see that the register file takes in as inputs the various fields from the instruction, which takes in the register numbers and outpus the data contained in those registers to the ALU. The ALU then performs the operation dictated by control, and then sends the result back to the register file.

7.3.3 Memory Operations – Loads and Stores

Before thinking about how to build the datapath, what is happening when loads and stores are performed?

- Loads
  - Add base register to the 16-bit signed offset field.
  - Read the value at that memory location and write back to register using register file.
- Stores
  - Add base register to 16-bit signed offset value
  - use register file to read the contents of provided register and write those contents into the computed memory address.

For the compution of the offset:

- We need to sign extend the offset field from 16 $\rightarrow$ 32 bits.
- We add that value to the address specified by the register (normally `$sp`)

Therefore in addition to the program counter, instruction memory, register file and ALU, we will need a way to sign-extend the immediate value, and we will also make use of the data memory segment.

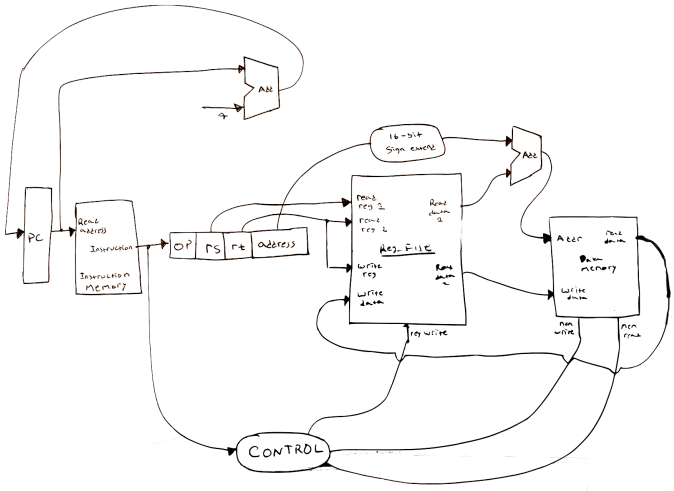


Figure 44. The datapath and control lines for load and store instructions.

7.3.4 Branch if Equal

For the branch if equal (`beq`) instruction the two operands (`rs` and `rt`) are compared and if equal, will jump to the address that the immediate operand encodes. There are a couple things that need to be implemented to create the datapath of this instruction:

- It will use the `zero` output of the ALU to test for equality. If equal, it will be asserted.
- The 16-bit immediate value will be sign extended to 32-bits.
- This value will then be shifted left 2 bits and added to `PC+4` to compute the address to be branched to.

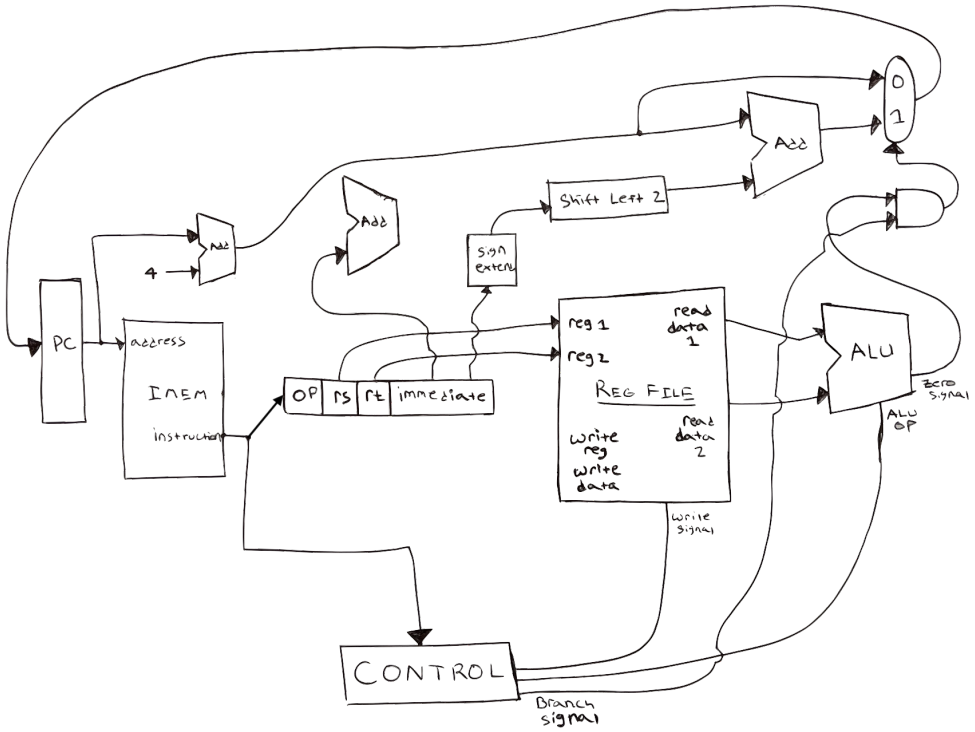
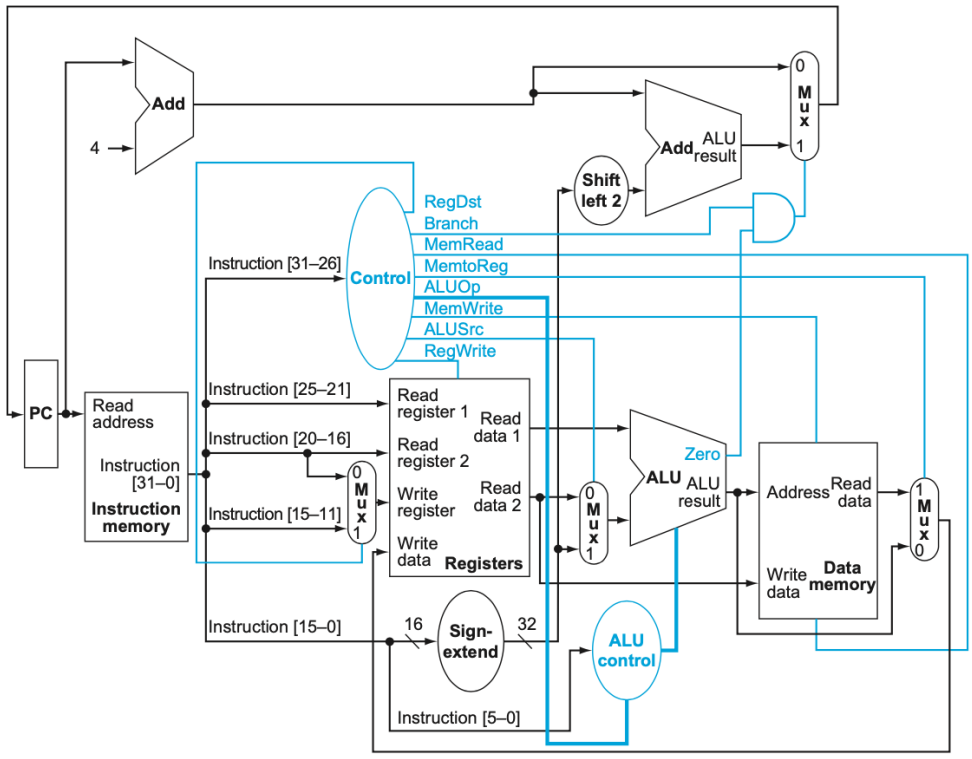


Figure 45. The datapath and control signals needed to implement the branch if equal instruction.

7.3.5 Putting Everything Together

Combining the hardware of these three instructions into one, we get:



The value of each control line is completely determined by the opcode fields of the instruction:

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

The control function can be entirely described by the following truth table:

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

7.3.6 Jump Instruction

Recall what the jump instruction looks like:

```
 ______ ____________________________
|000010|        encoded address     |
 ______ ____________________________
  31:26              25:0
```

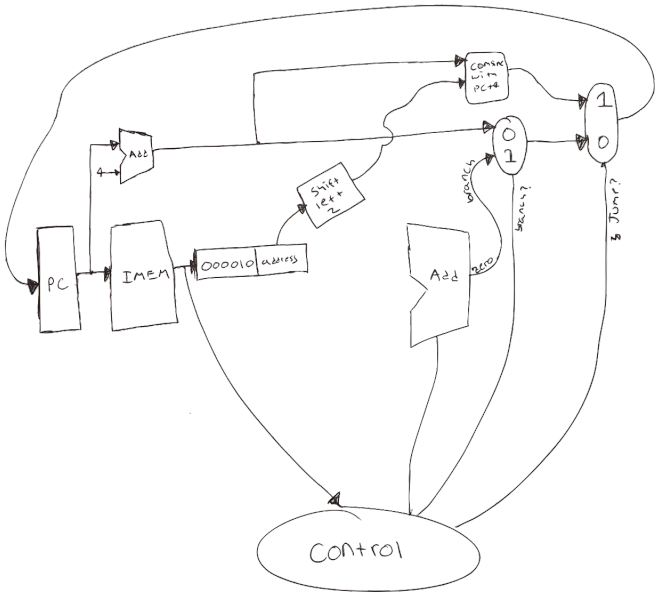
This performs an uncoditional jump instruction to the following decoded address where:

- Bits 31:28: upper 4 bits of PC + 4
- Bits 27:2: encoded address
- Bits 1:0: 00

Thus we we have the address that will be jumped to is:

```
 ____ ____________________________ __
|PC+4|        encoded address      |00|
 ____ ____________________________ __
 31:28              27:2
```

We can add the jump instruction by selecting between `(jump || (branch || PC+4))`, and adding the hardware to compute the jump address:



Subsection 7.4

Pipelining Overview

Let's say you have a car factory composed of four people:

- An engine builder
- A frame builder
- An interior builder
- A tester

A really inefficient way to build the car would for the engine guy to build the engine and then wait around while the frame builder finished the frame, who then finishes and waits for the interior guy to do the interior, who then finishes and waits for the tester to finish. Then once the whole car is finished, the process starts over again.

What would be a lot quicker would be for the engine guy to build an engine, and when he finishes he starts building another one. The frame guy gets the first engine and builds a frame around it and once he finishes he gets another frame from the engine guy, and so on and so forth. This is called Pipelining.

Non-Pipelined Factory

Time	1:00 PM	2:00 PM	3:00 PM	4:00 PM	5:00 PM	6:00 PM	7:00 PM	8:00 PM
Car								
A	Engine	Frame	Interior	Testing				
B					Engine	Frame	Interior	Testing

Pipelined Factory

Time	1:00 PM	2:00 PM	3:00 PM	4:00 PM	5:00 PM	6:00 PM	7:00 PM	8:00 PM
Car								
A	Engine	Frame	Interior	Testing				
B		Engine	Frame	Interior	Testing			
C			Engine	Frame	Interior	Testing		
D				Engine	Frame	Interior	Testing	
E					Engine	Frame	Interior	Testing

Figure 46. We see that in the same amount of time that the non-pipelined factory built two cars (8 hrs), the pipelined factory built five cars.

7.4.1 Designing Instruction Sets for Pipelining

What about our MIPS instruction set design makes pipelining easy?

- All MIPS instructions are the same length. This allows instruction fetch and instruction decode to be easily seperated into two phases.
- There are only a couple instruction formats. And all source register fields are located in the same place. This allows the second stage (instruction decode) can begin reading the register file at the same time that the hardware is determining the instruction type.
- Memory operands only appear in loads or stores. This means we can use the execute stage to calculate the memory address and then access the memory in the following stage
- Each MIPS instruction writes at most one result and does this in the last stage of the pipeline. Forwarding is harder if there are multiple results to forward per instruction or if there is a need to write a result early on in instruction execution.

Subsection 7.5

Pipeline Hazards

Let's look at a code example with and without pipeline hazards so that we can better understand them:

TIME --->									
Instruction	200 ps	400 ps	600 ps	800 ps	1000 ps	1200 ps	1400 ps		
sub $s1, $s0, $t0	IM	Reg | ALU	| reg						
add $t1, $t2, $t3		IM	reg | ALU	| reg					
sw $s1, 0($sp)			IM	reg | ALU	| DM				
add $t1, $t1, $t1				IM	reg | ALU	| reg			
lw $s3, 4($sp)					IM	reg | ALU	| DM	| reg	
beq $s1, $s3, exit						IM	reg | ALU	|	

TIME --->										
Instruction	200 ps	400 ps	600 ps	800 ps	1000 ps	1200 ps	1400 ps			
sub $s1, $s0, $t0	IM	Reg | ALU	| reg							
sw $s1, 0($sp)		IM	reg | ALU	| DM						
add $t1, $t2, $t3			IM	reg | ALU	| reg					
lw $s3, 4($sp)				IM	reg | ALU	| DM	| reg			
add $t1, $t1, $t1					IM	reg | ALU	| reg			
beq $s1, $s3, exit						IM	reg | ALU	|		
add $t0, $t0, $t0							IM	reg | ALU	| reg	

Figure 47. The first code segment has no pipeline hazards, but the bottom one has three: a data hazard, a structural hazard, and a control hazard.

A **data hazard** is when we cannot execute a planned instruction in the proper clock cycle because data that is needed to execute the instruction is not yet available.

A **structural hazard** is two instructions in a pipeline need the same hardware resource at the same time.

A **control hazard** is when the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

7.5.1 Data Hazards

To avoid the data hazard in the code above we could stall (perform a `nop`) which would then make the data available to be used. The problem with this, is data hazards can occur frequently and having the compiler stall (sometimes up to three `nop`s) is incredibly

inefficient. Instead what we can do is instead of waiting for the ALU to write the result to the register file and then accessing that data from the register file, we can grab the result directly from the ALU output. This is called forwarding.

There is a limit to what forwarding can be achieved though. For example let's say we are performing the following code:

```
1  lw   $s0, 20($t1)
2  sub  $t2, $s0, $t3   #$
```

Without forwarding, we would need two `nop`s so that the register read from is the has the correct written value. We are assuming that if a register is written to and read from in the same cycle that the value read is the same that is written.

lw	IF	ID	EX	MEM	WB					
nop		X	X	X	X	X				
nop			X	X	X	X	X			
sub				IF	ID	EX	MEM	WB		

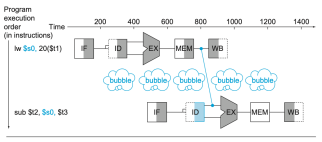


Figure 48. We see that even with forwarding, a pipeline stall is necessary when a `sub` follows a `lw`. One complication we have hidden away is that determining whether a stall is necessary won't really be known untill after the instruction is fetched and decoded (because we won't know if the register being read from is the same as that being written to in the `lw` instruction).

7.5.2 Control Hazards

Control hazards arise when a decision needs to be made which depends upon the the result of an instruction that is still processing. This often arises from conditional branch instructions: we cannot possibly know if we should branch on a `beq` instruction until we have determined if the arguments are equal.

One solution would be to just stall and wait until we have tested for equality. This is obviously not ideal since there will be a 100% chance of a slowdown when a branch is encountered. Another possibility is to always take the branch and only if we subsequently realize that we shouldn't have, to untake the branch. This means that there will only be a slowdown if we were wrong. Another option would be to dynamically predict if we should take the branch by keeping a history of branches taken and untaken and using that to predict if a branch should be taken.

Subsection 7.6

Pipelined Datapath and Control

Unfortunately we can't simply cut up our processor, put the instructions through and everythig will work out :(. For example whereas in a pipelined car factory the current car doesn't depend upon the previous car, in a pipelined processor the current instruction sometimes does depend upon the previoius instruction, and that makes things a little bit more complicated.

Idea:

- We are going to divide the instruction processing cycle into distinct "stages" of processing.
- At each stage we are going to ensure that there are enough hardware resources to process one instruction in each stage
- We will process a different instruction in each stage, where consecutive instructions in program order are processed in consecutive stages.

The benefit of pipelining is that it will increase processor **throughput**; under ideal circumstances we should be finishing an instruction every cycle. This however will not increase **latency** (the time it takes to complete an individual instruction).

In fact, pipelining will actually increase latency a little bit because of the overhead required in implementing the pipelined processor.

7.6.1 An Ideal Pipeline

Let's look at an ideal pipeline so that we can see how our pipelined processor strays from an idealized ones and what kind of complications we may encounter along the way.

- Repetition of identical operations. That is each instruction goes through the same steps. For our processor this won't be the case because an `add` doesn't access data memory like that of a `lw` instruction.
- Repetition of independent operations. That is, no instructions are dependent upone previous instructions. This of course won't always be the case (data or control hazards).
- Uniformly partitionable suboperations. That is processing can be evenly divided into uniform-latency suboperations (that don't share resources). Or more simply, each phase of the pipeline takes the same amount of time, and each phase is independent of the other phase. This would allow all parts of the processor to be active at all times. This isn't the case because the ALU takes more time than reading from a register.

7.6.2 Register Overhead

Recall that a processor is composed of state (memory) feeding into combinational logic, and that memory is formed using latches and flip-flops which can only be changed on rising edges. This means that when we split up our processor we need to have latches (registers/state) between each combinational logic phase. This places a limit on how many pipeline stages we can have, and thus the throughput is not increased by a factor of n where n is the number of pipeline stages; there comes a point when adding more pipeline stages just means that the processor will be spending more time on register overhead than actual processing.

The pipeline registers have to be large enough to contain the information required to compute all possible instructions, as we won't know which instruction will be performed until the instruction is fetched from memory.

In our five stage pipelined processor there is a pipelined register between each stage:

- **IF/ID** This holds the 32-bit instruction fetched from memory and the 32-bit `PC+4` which we need to save because we will need it in the case of a `beq` instruction (where the immediate value is sign extended, shifted left 2 bits, and added to `PC+4`).
- **ID/EX**
- **EX/MEM**
- **MEM/WB**

Also note that the PC is also a type of pipeline register as it serves to store state before connecting into combinational logic.

7.6.3 Instruction Fetch

In this stage we fetch the instruction at the address indicated by the `PC`. The `PC` is also incremented by 4. We store both values in the IF/ID pipeline register in case PC+4 is need for a later instruction (such as `beq`). We also save `PC+4` into the program counter so that the next instruction can be fetched in the next cycle.

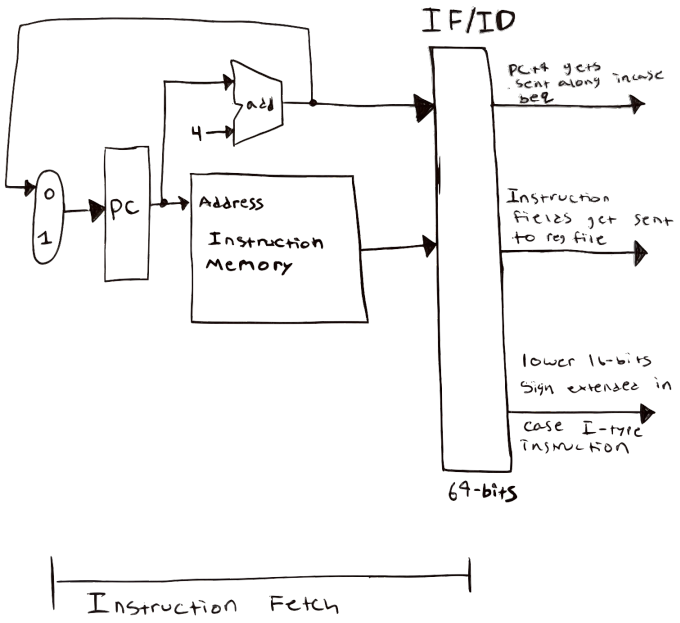


Figure 49. The first stage of our pipeline.

7.6.4 Instruction Decode

The sign-extended 16-bit immediate value, the register contents and `PC+4` are all sent to the ID/EX pipeline register.

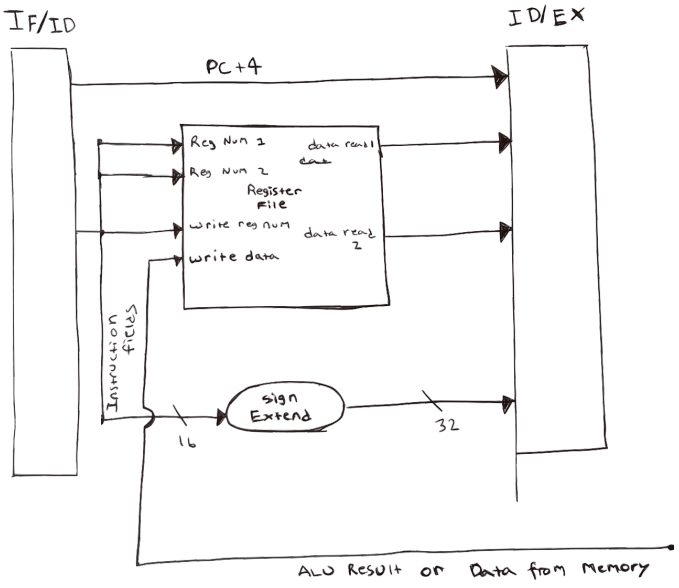


Figure 50. The instruction decode and register file read stage of our pipeline

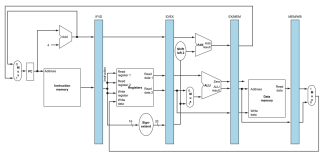


Figure 51. The key thing to remember is that the pipeline registers are necessary to connect the combinational logic with state in between pipeline stages. Also since we need to be able to work for all types of instructions, the data needed for any type of instruction needs to be passed through.

7.6.5 Bug Alert!!

Let's say that we perfor the following opperations:

Cycle →	1	2	3	4	5	6	7	8
Instruction ↓								
lw	IF	ID	EX	MEM	WB			
add		IF	ID	EX	MEM	WB		
sub			IF	ID	EX	MEM	WB	
or				IF	ID	EX	MEM	WB

Figure 52. If you look at the diagram on the right, we see that the data fetched from data memory by the `lw` instruction will be placed in the register designated by the `or` instruction!! What we must do instead is pass the write register number along the pipeline and then send that back along with the write data.

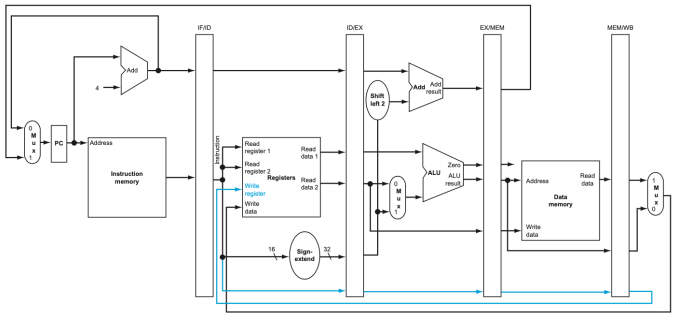


Figure 53. We see that we can fix the bug by passing the write register number along the pipeline and then back when it is needed in the writeback phase.

Subsection 7.7

Pipelined Control

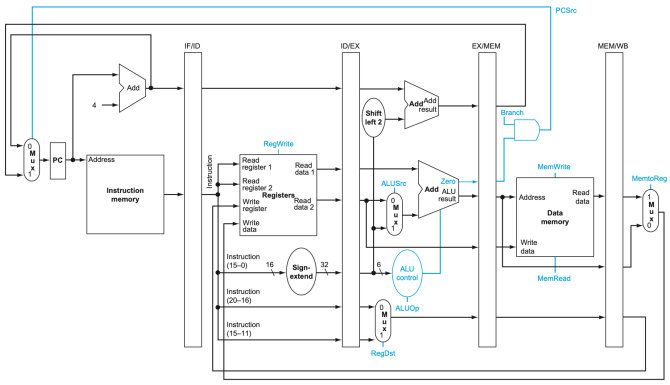


Figure 54. The pipelined datapath

Let's go through these signals by phase:

- **Instruction Fetch**
  - The mux recieves the result of `zero & Branch`. Recall that the `zero` signal comes from the ALU and is asserted when the two values being compared are equal, and the branch signal is asserted when the instruction is `beq`. If the signal is asserted than the branch address is set to PC. If deasserted than PC is set to `PC+4`.
  - A write signal for PC isn't necessary because we are designing the processor so that the PC is updated every clock cycle.

- **Instruction Decode**
  - The register write signal `RegWrite` determines whether the signal register should be written to or not. Note that just like the data to be written, this signal needs to get passed down the pipeline and then sent back to the register file so that the RegWrite signal corresponds to the correct instruction.
  - Signals aren't necessary for reg read or sign-extend since those happen every clock cycle.
- **Execute**
  - Two signals will need to be sent to the ALU control:
    - `ALUOp`: A two bit signal which determines if the instruction is a load/store, branch if equal, or an R-type instruction.
    - Funct Field: If an R-type insturction this 6-bit signal determins the R-type operation (`AND`, `OR`, `slt` (set less than), `add`, or `sub`).
    - The function field signal can come from the sign extended 16-bits as the least significant 6 bits will be correct if an R-type instruction and won't be used if not.
  - `RegDst`: determines if the register destination number for the write register comes from the `rt` field (deasserted), or from the `rd` field (asserted). If the instruction is a `lw` than the register write value will come from `rt` and if an R-type it will come from `rd`.
  - `ALUSrc`: This determines if the value to be added to the first read register is the second read register (in the case of an R-type instruction) or the immediate field (in the case of a load/store).
  - `Zero`: This is used in `beq` instructions and if asserted means that the two values being compared are equal and thus a branch should be taken.
- **Memory**
  - `MemWrite`: This is asserted on `sw` instructions.
  - `MemRead`: This is asserted on `lw` instructions.
  - `PCSrc`: Determines whether the PC should be `PC+4` or the branch address.
- **Writeback**
  - `MemtoReg`: This determines if the value read from memory or the result of the ALU should be written back to the register destination.

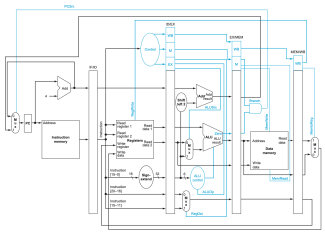


Figure 55. The pipelined processor with the addition of control signals

[b]2* Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALuOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
1w	0	0	0	1	0	1	0	1	1
SW	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Subsection 7.8
Data Hazards: Forwarding versus Stalling

Section 8
The Memory Hierarchy

Subsection 8.1
Introduction

The memory system of a computer utilizes both *temporal locality* and *spatial locality* to make the memory system seem much faster and the faster components much larger than they really are. Memory hierarchies take advantage of temporal locality by moving recently accessed items closer to the processor, and spatial locality by moving blocks of words to closer to the processor.

Temporal locality is the principle stating that if a data location is referenced then it will tend to be referenced again soon.

Spatial locality is the principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

We obviously want the data we look for to be located in the upper levels of memory (as this will be faster). The hit rate is the fraction of memory accesses found in the upper level; the higher the hit rate the better ¡performing our memory hierarchy is. The hit time is the time it takes to access the upper level of the memory hierarchy (including the time it takes to tell if the data we are looking for is in that level). The miss penalty is the time it takes to fetch the data if it is in a lower level, place it in the upper level, and deliver that to the processor.

Subsection 8.2
Memory Technologies

The memory hierarchy is made up of:

- **SRAM** (static random access memory). Static because it persists for as long as the device is powered on (without any form of refresh required). It is random access because the next memory location that can be read or written does not depend on the last access location. This is in contrast to spinning hard disks which cannot access random memory locations sequentially. Data is stored using flip-flops.
- **DRAM** (dynamic random access memory). The value is stored as a charge in a capacitor, and so will leak over time, requiring it to be refreshed to maintain its value. For this reason it is dynamic.
- **Flash Memory**. This is a type of electrically erasable programmable read-only memory. They are a consumable part in that the memory blocks wear out after a certain number of writes. Wear controllers try to minimize this by spreading out the write locations.
- **HDD**. A spinning collection of platters covered with magnetic material which can be written to using a moveable arm. These can only be read in a sequential manner, and so rotational latency exists. This is the time required for the desired sector of a disk to rotate under the read/write head (half the rotation time):

$$\begin{aligned}\text{Average rotational latency } &= \frac{0.5 \text{ rotation}}{5400\text{RPM}} = \frac{0.5 \text{ rotation}}{5400\text{RPM}/\left(60\frac{\text{seconds}}{\text{minute}}\right)} \\ &= 0.0056 \text{ seconds } = 5.6 \text{ ms}\end{aligned}$$

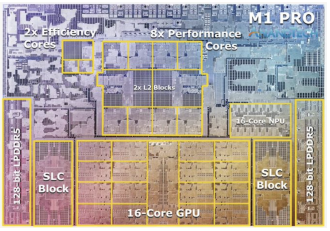


Figure 56. We can see the use of the memory hierarchy on the apple M1 Pro processor. Note the extensive use of SLC (system level cache), L2 cache, and DRAM, creating part of the memory hierarchy.

Subsection 8.3

The Basics of Caches

We will be looking at a very simple cache in which processor requests are one word and those words are stored in a block which only has room for one word.

How do we know whether an item is in cache or not, and how do we know what block that item is in? Well, the idea is to make it so that a given memory addresses can only be stored at certain spot in the cache. This way, all one has to do is check to see if that location is empty or not. This brings us to the difficulty that there are more memory addresses than cache spots. The solution to this is to map multiple addresses to each cache block. Direct-mapped caches use the following mapping to find a block:

(Block address) modulo (Number of blocks in the cache) .

If there are a 2^n entries in the cache, then the cache location is determined by the n LSBs of the memory address.

The question then is, what item is stored in the cache location (if any), and what item is it. To identify the element being stored we use a **tag** which is made up the upper $s - n$ bits (where s is the number of bits used to indicate memory address).

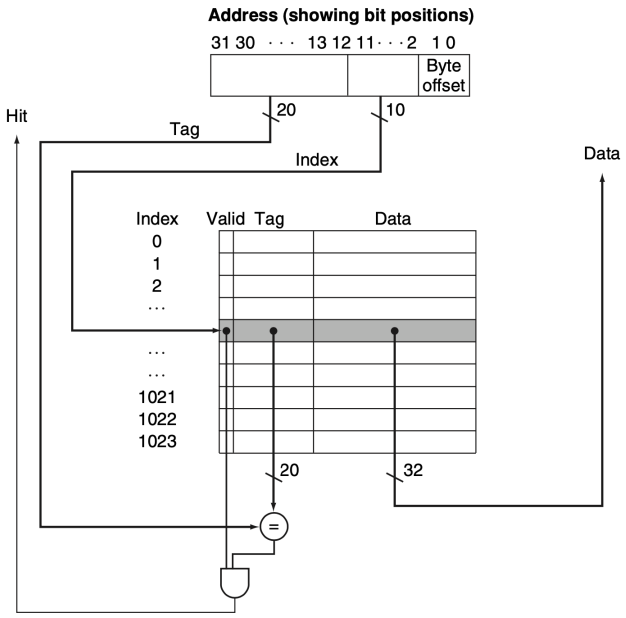


Figure 59. Address and cache organization.

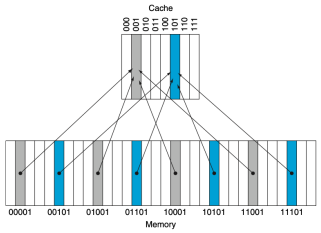


Figure 57. This is an example of direct-mapped cache. There are four memory locations that can be placed in each block, and the two MSB of the memory address are used as a tag to signify which address is being stored in the block.

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Figure 58. We see that because there are $2^3 = 8$ memory cache locations, the appropriate cache index is determined by the 3 LSBs of the address, and the tag indicating the address of the data being stored is the $5 - 3 = 2$ MSBs of the address.

8.3.1 Handling Cache Misses

Let's look at how an instruction miss will work (as data missees are very similar).

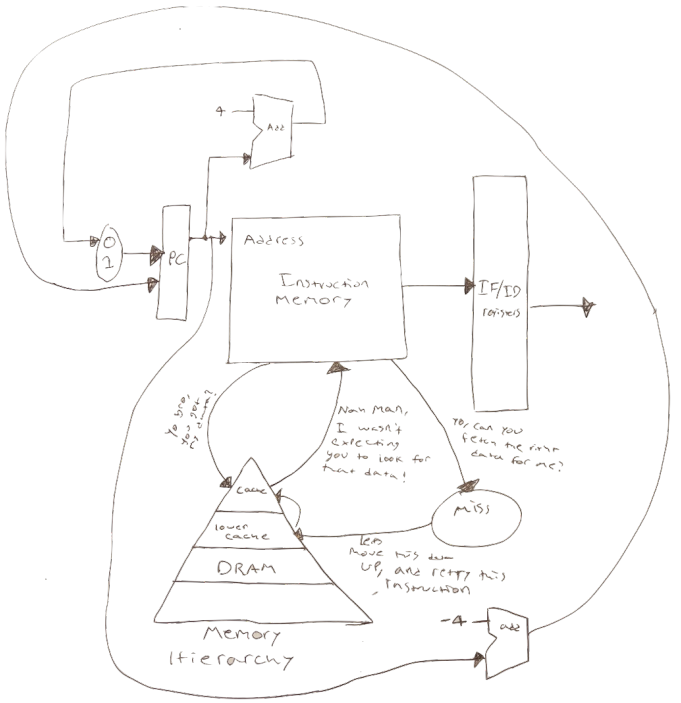


Figure 60. The instruction memory looks goes to the cache to look for the address, and if it is not found then a miss needs to be processed. This involves fetching the correct data from a lower level on the hierarcy, placing it in the cache and subtracting the PC by 4 so the instruction fetch stage can be redone.

Because the PC is incremented at the same time the instruction is fetched from cache, if a miss happens then the PC will need to be subtracted by 4, the correct data fetched from a lower level on the memory hierarchy, that data placed in the cache, and then then redoing the instruction fetch. As you might realize, this all takes time, and so a pipeline stall will need to take place.

8.3.2 Direct Mappped Cache Organization

With direct mapped cache organization we answer the two related questions:

- Is the data for address x stored in the cache
- Where in cache can we find this data.

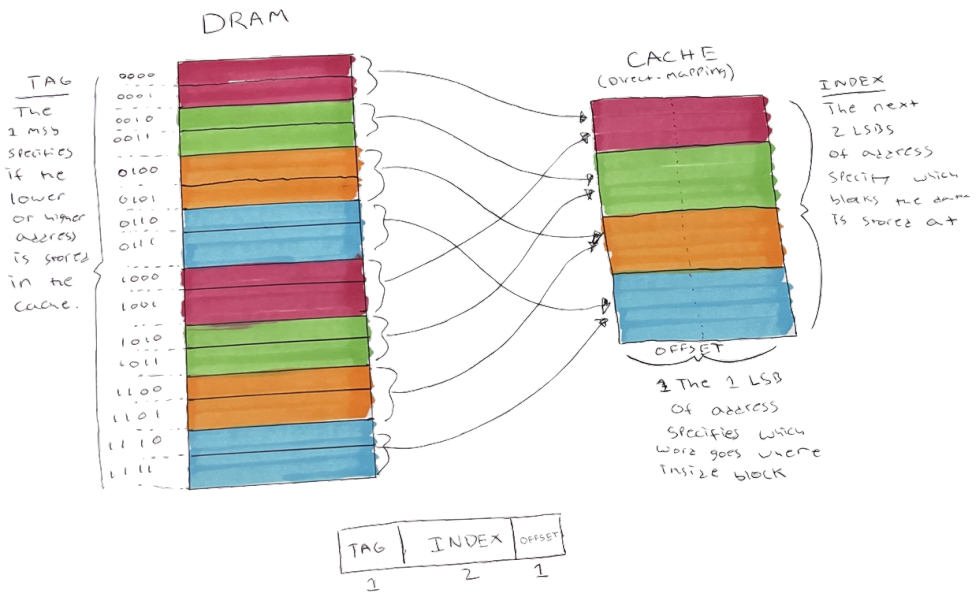


Figure 61. With direct mapping we answer the question of what addresses are being stored, in what block, and which word of the block using tag, index and offset bits (made up of the address).

More generally let's say we have the following:

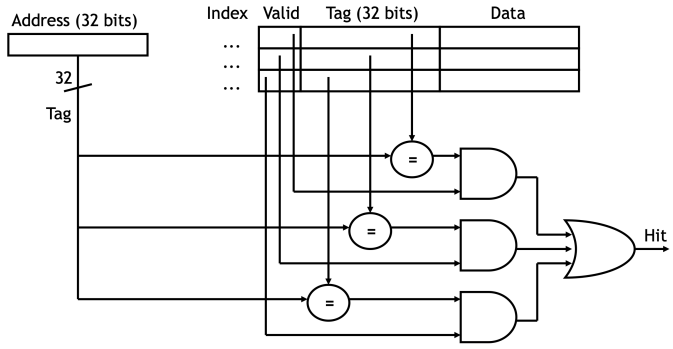
- 32-bit memory address
- 2^m cache blocks
- 2^n words inside each block

Then given any address

- The n LSBs make up the offset
- The m next LSBs make up the chache index
- The $32 - m - n$ MSBs make up the tag.

8.3.3 Fully Associative Cache Organization

The issue with a direct mapped cache is that we could encounter a situation where we are utilizing only a fraction of our cache because the mod of the address keeps returning the same cache block, even as we change the address. We can solve this issue by having a fully associative cache organization. This means that we can place the address anywhere in the cache. The problem with this though, is that we would have to search through the entire cache to check if our data was there:



Subsection 8.4

Virtural Memory

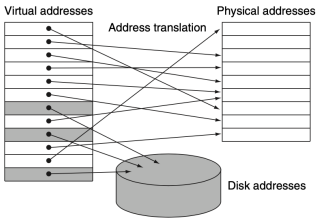


Figure 62. We see the mapping of virtural memory blocks (pages) mapped from virtural addresses to physical addresses.

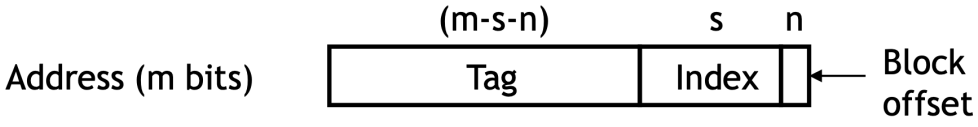
Virtural memory gives individual processes the illusion that they have access to their own, seperate, incredibly large and incredibly fast, address space. In fact the virtural address space is often larger then the physical address space because it often includes some solid state disk memory blocks as well (though this is not aparrent to the process). This virtural memory space maps to physical addresses. Although very similary in concept to that of a cache, different terminalogy is used. A virtural memory block is called a **page**, and a virtural memory miss is called a **page fault**. Virtural pages can be mapped to any phyisical page in memory. This allows the use of smart software algorithms to reduce the likelyhood of a page fault. Because of this, a **page table** is used which inexes the virtural page number to the physical page number. Because each process has its own virtural memory, each process has its own page table. The processor contains a **page table register** which points to the location of the page table.

8.4.1 Set-Associate Cache Organization

An intermediate possibility is a set-associative cache, where the cache is divided into groups of blocks, called sets. Each memory address maps to exactly one set in the cach, but the data can be placed in any block within that set. To find the set where our data is stored, we use a very similar technique to direct mapping. If

- a cache has 2^s sets and
- each block has 2^n bytes

then the memory address can be partitioned as follows:



The set index can be computed as:

- Block Offset = Memory Address mod 2^n
- Block Address = Memory Address $/2^n$
- Set Index = Block Address mod 2^s.

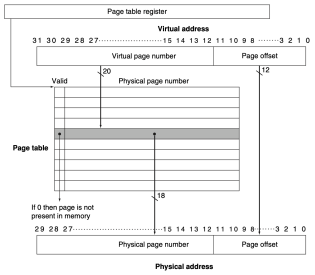


Figure 63. We see the page table register pointing to the start of the page table. The page table consists of a valid bit which indicates if the page is located in memory. Each entry consists of an index (the virtural page number) and an associated physical page number. The use of the page offest is to read up to 32 bits.

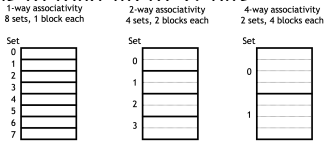


Figure 64. We can divide the cache into any number of sets. If each set has 2^x blocks, then we say the cache is a 2^x-way associative cache.

8.4.2 Page Faults

Page faults are incredibly expensive (millions of clock cycles) and so virtural memory systems are designed to reduce the chance of a page fault as much as possible. When a page fault does occure, the OS is given control via the exception mechanism. Then the operating system goes to the **swap space**. This is the area where the entire virtural memor space is, and indicates where the virtural page is stored on disk. When a page fault occurs and all the pages in main memory are in use, then the operating system chooses the LRU (least recently used) page to be replaced with the new page, so as to reduce the chance of a future page fault.

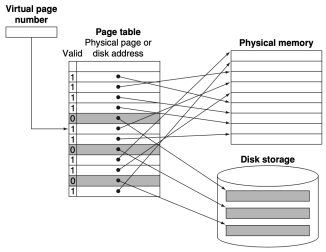


Figure 65. Note that although it may seem as thought the page table for physical memory and disk are one data structure, that is often not the case.

8.4.3 Translation Lookahead Buffer

Let's say that a process wants to fetch the next instruction to be performed. Well, then because the page table is located in main memory, then once memory access would be required to get the physical address location, and a second main memory access to get the instruction data. That is a bit cumbersome, performing two memory accesses for every memory access. To avoid this as much as possible processors have a smaller, but faster cache that keeps track of the most recently used addresses on our page table. In this sense, it is a subset of the page table. Thus if a TLB miss occurs, then the processor goes to the true page table, looks for the item, and if found places it in the TLB, and looks agaiin at the TLB (where it is now found). If a true miss occurs and the data is on disk, then the OS gets control and does its thing.