Keith On...

Computer Organization

Keith Schubert

Contents

Part I

Data Representation and Manipulation

Chapter 1

Codes

Codes are used to represent members of a set by a sequence of symbols. For our purposes, the sequence of symbols will always be a sequence of $\{0, 1\}$. Codes have an encoding for each member to be represented. Codes can be fixed or variable in length. Fixed length codes like ascii have the same number of symbols in every encoding of the code. Variable length codes use different numbers of symbols to represent the encodings. For instance if '1' is 'a', '01' is 'b', and '00' is 'c', then the code is variable length. The major trouble with variable length codes is splitting the message up into the individual encodings. If the code is prefix (postfix) then the code can be directly read from left to right (right to left).

1.1 Standard Codes

1.1.1 Unsigned

decimal	Binary	Gray	BCD	2421	Residue(5,3)	Residue(7,2)
0	0000	0000	0000	0000	000,00	000,0
1	0001	0001	0001	0001	001,01	001,1
2	0010	0011	0010	0010	010,10	010,0
3	0011	0010	0011	0011	011,00	011,1
4	0100	0110	0100	0100	100,01	100,0
5	0101	0111	0101	1011	000,10	101,1
6	0110	0101	0110	1100	001,00	110,0
7	0111	0100	0111	1101	010,01	000,1
8	1000	1100	1000	1110	011,10	001,0
9	1001	1101	1001	1111	100,00	010,1
10	1010	1111			000,01	011,0
11	1011	1110			001,10	100,1
12	1100	1010			010,00	101,0
13	1101	1011			011,01	110,1
14	1110	1001			100,10	-
15	1111	1000			-	-

BCD is a decimal code designed to be compatible with standard binary numbers. It is sometimes called 8421 code due to the weights on the columns. The 2421 code was designed to be the same as BCD for 0-4 and make the 9's complement, which is important for easy subtraction, of 0-4 (i.e. 9-5 respectively) be easy to take because you can simply flip the bits.

Gray code is an alternate to binary. It is not a decimal code, and hence does not waste 6 codes for every four bits. Gray code was designed to have only one bit flip at any given time. This is helpful in systems which have analog components and need to count. For instance in an NC drill, we might want to encode the shaft position and hence put gray code bars on the shaft and have an ir sensor read them. Since only one bit flips between each consecutive number, it is easy to verify if we are reading correctly and thus get a good idea of how fast the shaft is spinning and where the shaft is. Gray code is also useful to us in Karnaugh maps and code maps because the one bit flipping property lets us find errors of type one easily (Karnaugh maps) and measure Hamming distance easily (code maps). Notice that the first bit of a gray code is just like binary (all 0's first then 1's), while the rest follow a 0110 pattern on reducing scales.

The easiest way to read grey code is to start from the left and just copy the first bit. From then on if the next digit to the right is 0 then repeat the last digit you wrote, if it is 1 flip the last digit you wrote.

Example 1 *What is the value of* 101111_{gray}*?*

Starting at the left copy the first bit:

Gray	*1*	*0*	*1*	*1*	*1*	*1*
Binary	*1*					

The next bit is a 0 so repeat the last bit you wrote (in this case a 1):

Gray	*1*	*0*	*1*	*1*	*1*	*1*
Binary	*1*	*1*				

The next bit is a 1 so flip the last bit you wrote (in this case 1 flips to 0):

Gray	*1*	*0*	*1*	*1*	*1*	*1*
Binary	*1*	*1*	*0*			

The next bit is a 1 so flip the last bit you wrote (in this case 0 flips to 1):

Gray	*1*	*0*	*1*	*1*	*1*	*1*
Binary	*1*	*1*	*0*	*1*		

The next bit is a 1 so flip the last bit you wrote (in this case 1 flips to 0):

Gray	*1*	*0*	*1*	*1*	*1*	*1*
Binary	*1*	*1*	*0*	*1*	*0*	

The next bit is a 1 so flip the last bit you wrote (in this case 0 flips to 1):

Gray	*1*	*0*	*1*	*1*	*1*	*1*
Binary	*1*	*1*	*0*	*1*	*0*	*1*

Binary 110101 is 53, so gray 101111 is 53.

Residue number systems (residue codes) are fun though rarely used because of the difficulty in converting back from them to binary. Residue codes are specified by a series of remainders, taken to relatively prime bases (listed parenthesis and separated by commas). The remainders are in the same order as the specified bases and also separated by commas. The advantage of this system is you can perform fast addition, multiplication, and subtraction (if the divisor is not zero in any of the residues you can also do division efficiently), extremely fast, as the modulo terms are independently calculated by the modulo of the arithmetic operation being performed.

Example 2 *Calculate* $7+3$*,* $3*4$*,* $14-8$*, and* $14/7$ *in Modulo(5,3). Note we can do division because* $7 \mod 5 = 2 > 0$ *and* $7 \mod 3 = 1 > 0$.

$7+3 = (010, 01) + (011, 00) = (010 + 011 \mod 5, 01 + 00 \mod 3) = (000, 01) = 10$

$3*4 = (011, 00) * (100, 01) = (011 * 100 \mod 5, 00 * 01 \mod 3) = (010, 00) = 12$

$14-8 = (100, 10) - (011, 10) = (100 - 011 \mod 5, 10 - 10 \mod 3) = (001, 00) = 6$

$14/7 = (100, 10) - (010, 01) = (100/010 \mod 5, 10/01 \mod 3) = (010, 10) = 2$

1.1.2 Signed

decimal	Signed Binary	1's Comp	2's Comp	Excess-7	Excess 8
8	-	-	-	1111	-
7	0111	0111	0111	1110	1111
6	0110	0110	0110	1101	1110
5	0101	0101	0101	1100	1101
4	0100	0100	0100	1011	1100
3	0011	0011	0011	1010	1011
2	0010	0010	0010	1001	1010
1	0001	0001	0001	1000	1001
0	0000,1000	0000,1111	0000	0111	1000
-1	1001	1110	1111	0110	0111
-2	1010	1101	1110	0101	0110
-3	1011	1100	1101	0100	0101
-4	1100	1011	1100	0011	0100
-5	1101	1010	1011	0010	0011
-6	1110	1001	1010	0001	0010
-7	1111	1000	1001	0000	0001
-8	-	-	1000	-	0000

Note that both signed binary and 1's compliment have a positive and negative 0. Signed binary was an early development, but is not that useful because you can't use a standard adder/subtractor.

1's compliment is easy to calculate (flip the bits to convert from positive to negative), and is useful in turning an adder into a subtractor (the number to be subtracted is turned into the 2's complement, by finding the 1's complement, then setting the carry-in bit of the adder to do the +1).

2's compliment is the standard form for storing negative numbers in computers because you can easily convert (either by flipping bits and adding 1, or by starting on the right and copying bits up to and including the first 1, then flipping the remaining bits), and standard adder/subtractor circuits can be used.

Excess codes are most commonly used in floating point number exponents, as they preserve the numeric order of greatness (you can use standard compare circuits to check size). The excess is either half the total numbers ($16/2 = 8$ for excess 8) or half the total numbers minus 1 ($16/2 - 1 = 7$ for excess 7).

1.2 Huffman Codes

Huffman codes are variable length codes that produce optimal expected code lengths.

$$ecl = \sum_{l \in C} (freq(l) \times length(l))$$

Example:

Consider the string "adabaabcaabacadaccac" that we want to encode. There are four members of the set (a, b, c, d) which means the members can be represented by a two bit fixed code. But consider the following encoding (a=1, b=001, c=01, d=000). The frequencies of the members are (a=10/20=.5, b= 3/20=.15, c=5/20=.25, d =2/20=.1). The ecl of the variable code is

$$\begin{aligned} ecl &= .5 * 1 + .15 * 3 + .25 * 2 + .1 * 2 \\ &= 1.65 \end{aligned}$$

The expected code length is only 1.65 bits/character.

1.2.1 Huffman Algorithm

1. Calculate the frequencies of each member

$$\frac{\#\, occurrences\, of\, member}{Total\, occurrences}$$

2. Form decode tree from forest

   (a) make 1 node tree for each member with frequency and member name

   (b) join two trees with the smallest frequency on root node by making them branches of a new root node and giving the new root node the sum of the frequencies of the old root nodes

   (c) put new tree in forest and repeat joining till only one tree remains (the answer)

3. encode or decode message

1.3 Error Detection and Correction

Errors can happen in a variety of ways. Bits can be added, deleted, or flipped. Errors can happen in fixed or variable codes. For simplicity we will consider only bit flips in fixed codes. Note that variable codes can be packed into fixed length blocks for transmission and storage, so this is not as restrictive as it might sound at first.

The Hamming distance (d_H) between two codewords is the number of bit flips to turn one codeword into the other codeword. It can also be thought of as the number of bits that are different between two codewords. The Hamming distance can be extended to a set, by defining it as the minimum distance between any two codewords in the set. The Hamming distance is useful in codes because it tells us how many errors can be detected (E_d) and how many errors can be corrected (E_c) The relations are given by

$$\begin{aligned} d_H &\geq 1 + E_d \\ d_H &\geq 1 + 2 \times E_c \end{aligned}$$

Example

Consider the codes $(00001, 01100)$.

1. What is the Hamming distance?

   3

2. How many errors can be detected? How many can be corrected?

   $3 \geq 1 + d$ thus detect 2

   and

   $3 \geq 1 + 2c$ thus correct 1

3. It is desired to add another codeword without reducing the Hamming distance. What codeword do you suggest?

   any of the following will work:

   - 10010
   - 10110

- 10111
- 11010
- 11011
- 11111

1.3.1 Hamming Code

To detect and/or correct errors, two pieces of information must be sent, the original data (D_i) and check bits (C_j). Consider numbering in binary each position in an array of bits to be sent starting at 1, and positioning the check bits at the powers of two.

Address	0	0	0	0	0	0	0	1	1	1
	0	0	0	1	1	1	1	0	0	0
	0	1	1	0	0	1	1	0	0	1
	1	0	1	0	1	0	1	0	1	0
Code	C_0	C_1	D_1	C_2	D_2	D_3	D_4	C_3	D_5	D_6

The check bits are then calculated by taking the exclusive-or (xor) of all the data bits (D_i), whose address contains a 1 in the same place as the check bit. Thus,

Address	0	0	0	0	0	0	0	1	1	1
	0	0	0	1	1	1	1	0	0	0
	0	1	1	0	0	1	1	0	0	1
	1	0	1	0	1	0	1	0	1	0
Code	C_0	C_1	D_1	C_2	D_2	D_3	D_4	C_3	D_5	D_6

$$C_0 = D_1 \oplus D_2 \oplus D_4 \oplus D_5$$

Address	0	0	0	0	0	0	0	1	1	1
	0	0	0	1	1	1	1	0	0	0
	0	1	1	0	0	1	1	0	0	1
	1	0	1	0	1	0	1	0	1	0
Code	C_0	C_1	D_1	C_2	D_2	D_3	D_4	C_3	D_5	D_6

$$C_1 = D_1 \oplus D_3 \oplus D_4 \oplus D_6$$

And so on.

The Hamming distance is three, which will be proved in three cases.

1. If the data portion of two codewords differs by only one bit, then note that the address of each data bit has at least two ones in it. This means that the data bit that is different will cause at least two check bits to be different, yielding a Hamming distance of three.

2. If the data portion of two codewords differs by two bits, then note that no two data bits affect all the same check bits. Thus, there exists at least one check bit that is affected by only one of the two data bits that differs, and will thus be different between the two codewords, yielding a Hamming distance of three.

3. If the data portion of two codewords differs by more than two bits the result is trivial.

Q.E.D.

A Hamming distance of three means

$$
\begin{aligned}
3 &\geq 1 + E_d \\
2 &\geq E_d \\
3 &\geq 1 + 2 \times E_c \\
2 &\geq 2 \times E_c \\
1 &\geq E_c.
\end{aligned}
$$

One error can be corrected or two detected. To find the error for correction you create its address by taking the exclusive-or of the check bits and the data that created them. A 1 will result only if an odd number of errors happened in the subset checked. The address that results is the address of the error, which is fixed by toggling.

Example

the data "1010" is to be sent by Hamming Code. Since there are only four bits of data, only three check bits are needed. The data is put in place.

Address	0	0	0	1	1	1	1
	0	1	1	0	0	1	1
	1	0	1	0	1	0	1
Code	C_0	C_1	1	C_2	0	1	1

Next the check bits are calculated and

$$
\begin{aligned}
C_0 &= D_1 \oplus D_2 \oplus D_4 \\
&= 1 \oplus 0 \oplus 1 \\
&= 0 \\
C_1 &= D_1 \oplus D_3 \oplus D_4 \\
&= 1 \oplus 1 \oplus 1 \\
&= 1 \\
C_2 &= D_2 \oplus D_3 \oplus D_4 \\
&= 0 \oplus 1 \oplus 1 \\
&= 0
\end{aligned}
$$

Thus,

Address	0	0	0	1	1	1	1
	0	1	1	0	0	1	1
	1	0	1	0	1	0	1
Code	0	1	1	0	0	1	1

Now, assume an error happens. It could be anywhere, but for this example assume that the bit in position 6 is toggled.

Address	0	0	0	1	1	1	1
	0	1	1	0	0	1	1
	1	0	1	0	1	0	1
Code	0	1	1	0	0	0	1

To find it get the address by

$$\begin{aligned}
A_0 &= C_0 \oplus D_1 \oplus D_2 \oplus D_4 \\
&= 0 \oplus 1 \oplus 0 \oplus 1 \\
&= 0, \\
A_1 &= C_1 \oplus D_1 \oplus D_3 \oplus D_4 \\
&= 1 \oplus 1 \oplus 0 \oplus 1 \\
&= 1, \\
A_2 &= C_2 \oplus D_2 \oplus D_3 \oplus D_4 \\
&= 0 \oplus 0 \oplus 0 \oplus 1 \\
&= 1.
\end{aligned}$$

Yielding the address, $A_2A_1A_0 = 110 = 6$, which is the error.

Example: Hello There

Compress "hello there" using a Huffman code designed off it. Then use a Hamming code on 11 bit blocks of the compressed message. How does the overall message size compare to the original? I will just list the code, the tree is obvious from it. Note that other trees are possible.

letter	frequency	code
h	$\frac{2}{11}$	100
e	$\frac{3}{11}$	11
l	$\frac{2}{11}$	101
o	$\frac{1}{11}$	011
sp	$\frac{1}{11}$	010
t	$\frac{1}{11}$	001
r	$\frac{1}{11}$	000

Huffman code: 10011101101 01101000110 01100011

Hamming Code

Since I don't have enough bits to do 3 groups of 11, I could pad with 0's or 1's or I could make the last packet shorter. Alternately I could have made an EOF code in my Huffman code. In this case I will just skip them so you see how that works. You should mention the problem and what you will do along with the solution.

Data Section	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
First	c_0	c_1	1	c_2	0	0	1	c_3	1	1	0	1	1	0	1
Second	c_0	c_1	0	c_2	1	1	0	c_3	1	0	0	0	1	1	0
Third	c_0	c_1	0	c_2	1	1	0	c_3	0	0	1	1			

Data Section	1	2	**3**	4	**5**	6	**7**	8	**9**	10	**11**	12	**13**	14	**15**
First	1	c_1	1	c_2	0	0	1	c_3	1	1	0	1	1	0	1
Second	1	c_1	0	c_2	1	1	0	c_3	1	0	0	0	1	1	0
Third	0	c_1	0	c_2	1	1	0	c_3	0	0	1	1			

Data Section	1	2	**3**	4	5	**6**	**7**	8	9	**10**	**11**	12	13	**14**	**15**
First	1	0	1	c_2	0	0	1	c_3	1	1	0	1	1	0	1
Second	1	0	0	c_2	1	1	0	c_3	1	0	0	0	1	1	0
Third	0	0	0	c_2	1	1	0	c_3	0	0	1	1			

Data Section	1	2	3	4	**5**	**6**	**7**	8	9	10	11	**12**	**13**	**14**	**15**
First	1	0	1	0	0	0	1	c_3	1	1	0	1	1	0	1
Second	1	0	0	0	1	1	0	c_3	1	0	0	0	1	1	0
Third	0	0	0	1	1	1	0	c_3	0	0	1	1			

Data Section	1	2	3	4	5	6	7	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
First	1	0	1	0	0	0	1	1	1	1	0	1	1	0	1
Second	1	0	0	0	1	1	0	1	1	0	0	0	1	1	0
Third	0	0	0	1	1	1	0	0	0	0	1	1			

The length is thus 42 bits for the compressed code with error correction. The original message was 11 chars $\times$ 7 bits/char $=$ 77 bits. The new message is much smaller (less than 4/7).

Chapter 2

Integers

2.1 Integer numbers

unsigned All the bits are used for the magnitude of the number. (0 to $2^n - 1$)

signed int The first bit indicates the sign (1 is negative), the remaining $n-1$ bits are used for magnitude. ($-2^{n-1}+1$ to $2^{n-1}-1$)

1's complement Positive numbers are the same as signed int, but negative are found by inverting each bit of the positive number with the same magnitude. ($-2^{n-1}+1$ to $2^{n-1}-1$)

2's complement As 1's complement, but negative numbers have 1 added to them after the bitwise inversion. This removes a -0 code, so the extra code is assigned to -2^{n-1}. This is the natural way to handle numbers if addition and subtraction of mixed sign numbers are needed. (-2^{n-1} to $2^{n-1}-1$)

2^{n-1} **excess** The code is found by adding 2^{n-1} to the value (hence the name). This gives a slightly larger negative range. (-2^{n-1} to $2^{n-1}-1$)

$2^{n-1}-1$ **excess** The code is found by adding $2^{n-1}-1$ to the value (hence the name). This gives a slightly larger positive range. ($-2^{n-1}+1$ to 2^{n-1})

Example 3 *Convert the following*

1. *-39 to 8 bit 2's complement*

39	
19	1
9	1
4	1
2	0
1	0
0	1

   $39_{10} = 100111_2 = 00100111_2$

   $-39_{10} = 11011001_2$

2. *234 to 8 bit unsigned*

234	
117	0
58	1
29	0
14	1
7	0
3	1
1	1
0	1

$234_{10} = 11101010_2$

2.2 Addition

The basic addition routines can be modified to work for any of the codes as well as subtraction for the codes. The special customizations will be considered later. Right now, the typical techniques for addition are considered.

Example 4 *Calculate the following in binary using 8 bits.*

1. $42 - 51$
2. $51 - 42$

Sol:

	42	51
+	00101010	00110011
-	11010110	11001101

42	00101010	51	00110011
-51	11001101	-42	11010110
	11110111		100001001
-9	-00001001	9	00001001

2.2.1 Ripple Adders

This is the technique that is covered in CSCI 310. Basically, full bit adders, see Figure 2.1, are created and cascaded together. The carry bit from the previous full adder must arrive before the result is added. The resulting valid carries thus ripple down to the most significant bit (hence the name). Adding n bit numbers, thus takes the propagation time of $n + 1$ levels of logic, i.e. it is O(n) in time to calculate addition. Thus if 32 bit numbers are added on fast logic (1ns per stage/gate) the process would take 33ns. This is way too slow. On the bright side, none of the gates take more than 2 inputs so the size of the gates is O(1).

2.2.2 Conditional Sum

Conditional sum is a divide and conquer algorithm, and hence exploits binary tree parallelism. The algorithm works by calculating both possible results for each bit (if carry in was 1 or 0), then performing paired conditional concatenation using the actual carry bit of the lower number, see Figure 2.2.

1. form conditional terms for each digit in summation → (digit with carry, digit without carry) = $(x_i + y_i + 1, x_i + y_i)$
2. group by twos from right and for both conditional values in the right parenthesis form the result as follows:

Figure 2.1: (left) Half Adder, (right) Full Adder

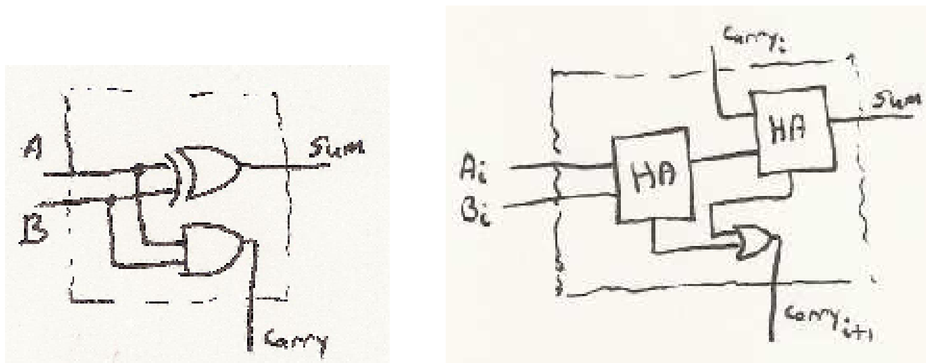
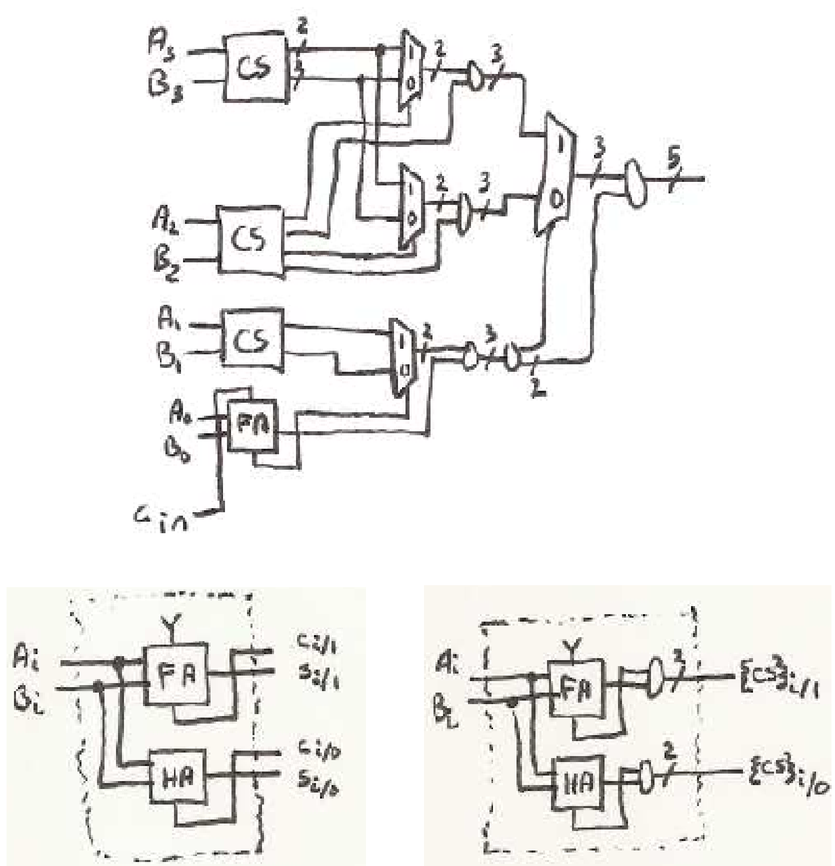


Figure 2.2: Conditional Sum Adder (above), and its sub-blocks (below, left and right).



(a) the leftmost bit of the two terms on the right are the carry bits used to select the term on the left

(b) concatenate the appropriate term on the left (picked by carry) with each term on right after removing the parity bits of the right terms

3. continue pairings until only 1 term remains. pick right number if $c_{in} = 0$ else pick left.

Example 5 *Add $x = 0110$ and $y = 1111$ by conditional sum and indicate if overflow occurred.*

0+1   1+1   1+1   0+1
↓   ↓   ↓   ↓
(10,01)   (11,10)   (11,10)   (10,01)
↘   ↙   ↘   ↙
(101,100   (110,101)
↘   ↙
(10110,10101)
1 0101

No overflow occurred (added a positive and negative number).

Example 6 *Calculate $7 - 8$ by conditional sum.*

$7 = 0111$ *and* $-8 = 1000$

0   1   1   1
+1   0   0   0
(10,01)   (10,01)   (10,01)   (10,01)
↘   ↙   ↘   ↙
(100,011)   (100,011)
↘   ↙
(10000,01111)

Since this was done as addition no carry-in was set so the solution is $\underline{0 \mid 1111}$ *or* -1 *in signed base ten.*

Example 7 *Add by conditional sum $x = 01100110$ and $y = 00110011$.*

$0+0$ $1+0$ $1+1$ $0+1$ $0+0$ $1+0$ $1+1$ $0+1$
↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
(01, 00)   (10, 01)   (11, 10)   (10, 01)   (01, 00)   (10, 01)   (11, 10)   (10, 01)
↘   ↓   ↘   ↓   ↘   ↓   ↘   ↓
(010, 001)   (110, 101)   (010, 001)   (110, 101)
↘   ↙   ↘   ↙
(01010, 01001)   (01010, 01001)
↘   ↙
(010011010, 010011001)
0 10011001

Why go through this? First, by a folk theorem of Dr. Alan Laub, "*What is hard for us tends to be easy for computers (and vice versa).*" In reality this process is really easy for a computer to do. Second, the process is highly parallel, so it can be done very fast. If the numbers to be added are n bits long this takes $2(\log_2(n) + 1)$ levels of logic, much better than the $n + 1$ levels of logic required by ripple calculations. Thus it is O(log(n)) in time complexity. For example, for adding the 32 bit numbers considered already, conditional sum would take $2(\log_2(32) + 1) = 12$ levels of logic, so on the fast logic described it would be 12ns, a huge improvement.

2.2.3 Carry-Lookahead

This is also referred to as lookahead carry. Assume $x + y = z$. Pre-generate all carries with 2-level logic. Usually form (g,p,c) generate, propagate, carry.

$$\begin{aligned}
G_i &= x_i \cdot y_i \\
P_i &= x_i + y_i \\
C_i &= G_i + P_i \cdot C_{i-1} \\
&= G_i + P_i \cdot (G_{i-1} + P_{i-1} \cdot C_{i-2}) \\
&= G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot C_{i-2} \\
&= G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot G_{i-2} + \ldots + P_i \cdot P_{i-1} \cdot \ldots \cdot P_0 \cdot C_{in}
\end{aligned}$$

This method is very fast (regardless of size it take 5 levels of logic) but requires large gates for problems of reasonable size (even 16 or 32 bit numbers) and thus has problems with fan-in, fan-out, and size.

Often blocks of a number are handled with lookahead, and the blocks are connected in some fashion (for example ripple) to get the net result (i.e. just like single bit adds from a full adder are connected to propagate the carry bit, blocks or 4, 8, or more could be handled lookahead then connected to propagate the carry bit between them to handle a larger number, say 32 bits). Even better than cascading (ripple connection) the adders, is to us group carry-lookahead, in which each of the carry-lookahead adders output their group propagate and group generate variables to a circuit that generates the carry-in bits for each group. It takes 5 logic levels to generate the carries to each individual carry-lookahead adder, and each adder then takes 5 levels of logic to get the result, for a total of 10 levels of logic. For the example of adding 32 bit numbers with fast logic, it would take 10ns with group carry-lookahead adders (probably four or eight bits in a group).

Example 8 *Specify the equations of a two bit binary adder with carry in (i.e.: one equation for each of the sum bits and one equation for the carry out). Put the equations in sum of products form.*

Sol: Let the two numbers to be added be A_1A_0 and B_1B_0. Let the resulting sum be S_1S_0. Let the carries be C_{in} and C_{out}. Finally, let C_0 be the carry from the first bit added (saves writing).

$$\begin{aligned}
S_0 &= A_0 \oplus B_0 \oplus C_{in} \\
C_0 &= A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in} \\
S_1 &= A_1 \oplus B_1 \oplus C_0 \\
C_{out} &= A_1 \cdot B_1 + A_1 \cdot C_0 + B_1 \cdot C_0
\end{aligned}$$

Putting this in sum of products form yields

$$
\begin{aligned}
S_0 &= A_0' \cdot B_0' \cdot C_{in} + A_0' \cdot B_0 \cdot C_{in}' + A_0 \cdot B_0' \cdot C_{in}' + A_0 \cdot B_0 \cdot C_{in} \\
S_1 &= A_1' \cdot B_1' \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in}) + A_1' \cdot B_1 \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in})' + \\
&\quad A_1 \cdot B_1' \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in})' + A_1 \cdot B_1 \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in}) \\
&= A_1' \cdot B_1' \cdot A_0 \cdot B_0 + A_1' \cdot B_1' \cdot A_0 \cdot C_{in} + A_1' \cdot B_1' \cdot B_0 \cdot C_{in} \\
&\quad + A_1' \cdot B_1 \cdot (A_0' \cdot B_0' + A_0' \cdot C_{in}' + B_0' \cdot C_{in}') \\
&\quad + A_1 \cdot B_1' \cdot (A_0' \cdot B_0' + A_0' \cdot C_{in}' + B_0' \cdot C_{in}') \\
&\quad + A_1 \cdot B_1 \cdot A_0 \cdot B_0 + A_1 \cdot B_1 \cdot A_0 \cdot C_{in} + A_1 \cdot B_1 \cdot B_0 \cdot C_{in} \\
&= A_1' \cdot B_1' \cdot A_0 \cdot B_0 + A_1' \cdot B_1' \cdot A_0 \cdot C_{in} + A_1' \cdot B_1' \cdot B_0 \cdot C_{in} \\
&\quad + A_1' \cdot B_1 \cdot A_0' \cdot B_0' + A_1' \cdot B_1 \cdot A_0' \cdot C_{in}' + A_1' \cdot B_1 \cdot B_0' \cdot C_{in}' \\
&\quad + A_1 \cdot B_1' \cdot A_0' \cdot B_0' + A_1 \cdot B_1' \cdot A_0' \cdot C_{in}' + A_1 \cdot B_1' \cdot B_0' \cdot C_{in}' \\
&\quad + A_1 \cdot B_1 \cdot A_0 \cdot B_0 + A_1 \cdot B_1 \cdot A_0 \cdot C_{in} + A_1 \cdot B_1 \cdot B_0 \cdot C_{in} \\
C_{out} &= A_1 \cdot B_1 + A_1 \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in}) \\
&\quad + B_1 \cdot (A_0 \cdot B_0 + A_0 \cdot C_{in} + B_0 \cdot C_{in}) \\
&= A_1 \cdot B_1 + A_1 \cdot A_0 \cdot B_0 + A_1 \cdot A_0 \cdot C_{in} + A_1 \cdot B_0 \cdot C_{in} \\
&\quad + B_1 \cdot A_0 \cdot B_0 + B_1 \cdot A_0 \cdot C_{in} + B_1 \cdot B_0 \cdot C_{in}
\end{aligned}
$$

2.2.4 Other notes

Integer numbers larger than the word size of the computer can be handled by chaining. Two special assembly commands are often available to aid in chaining: addc, subb. Normally when you add the first carry in is zero, but for blocks of bits after the first block, the lower block might need to carry up. Addc uses the carry bit as c_{in} rather than assuming $c_{in} = 0$.

Two different signals are used to warn that the integer result might not be valid[1] : carry (c) and overflow (v). Carry is used for unsigned integers, and overflow is used for two's complement. Since both carry and overflow bits are both calculated at the same time[2] it is important to know what they mean, when they are relevant, and how they are calculated.

Overflow set if last two carries are different.

2.2.5 Signed Int

Addition

- if signs are same then add two $n-1$ digit numbers and keep sign
- else flip sign of second term and subtract (subtracting with same signs).

Subtraction ($S_1 - S_2$)

- if $S_1 \geq S_2 \geq 0$ or $S_1 \leq S_2 < 0$ then preserve sign and subtract absolute magnitudes,

[1] Overflow and carry are two of the typical condition codes. It is possible for a condition code to be set but the result is still valid. For instance carry could be set and overflow could be unset after an operation with 2's complement numbers. In this case the number is still valid since overflow is the signal for 2's complement.

[2] On some machines every arithmetic operation generates the condition codes, on other machines, like the SPARC, the condition codes are set only when special versions of the arithmetic commands that end in *cc* are used.

- if $S_2 > S_1 \geq 0$ or $S_2 < S_1 < 0$ then flip sign and subtract absolute magnitudes reversed,
- else flip sign of second term and add (adding with same signs).

2.2.6 2's Comp

For addition you just add the numbers normally with $c_{in} = 0$(no special cases).

For subtraction you take the 1's complement of the second number and add with $c_{in} = 1$(no special cases, note 1's complement +1 is 2's complement).

2.2.7 Excess

For addition, you need to carry extra bits while calculating, because you have to subtract the excess number after adding. This is needed because the excess was in each of the numbers added, so an extra excess is present which must be removed.

For subtraction, the excess gets removed in the process so it must be added back in after subtraction. Note the subtraction can result in an intermediate negative number, so extra bits are needed during calculation.

2.3 Multiplication

2.3.1 unsigned

Algorithm 1

1. set v to 0
2. for each digit do:
   (a) if lsb of x is 1, add y to v
   (b) left shift y
   (c) right shift x

This basically only handles numbers whose product fits in 1 register. In general multiplication could take up to 2 registers.

Algorithm 2

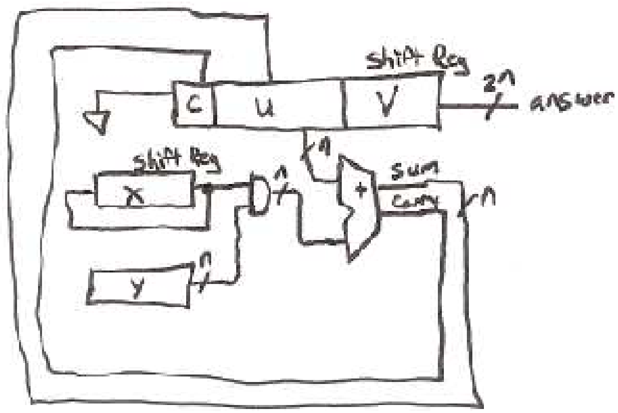
1. group two regs (u,v) for product, set to 0
2. for each digit do:
   (a) add (y and lsb(x)) to u hold carry in c
   (b) right shift (c,u,v)
   (c) circulant right shift x

Right shifting the product with carry is the same as left shifting (y_{hi},y), but without the need for a second register to hold the high order bits. The algorithm can be implemented in a circuit as is done in Figure 2.3.

Example 9 *Multiply 10 and 12 in binary using algorithm 2*

First we need to convert our numbers to binary: $x = 10_{10} = 1010_2$ and $y = 12_{10} = 1100_2$.

Figure 2.3: Unsigned Multiplier of Algorithm 2



c	*u*	*v*	*x*	*Comments*
0	*0000*	*0000*	*1010*	*Setup (Step 1)*
				Round 1
0	*0000*			*Step 2a: add* $y \cdot 0$ *to u (0+0=0)*
0	*0000*	*0000*		*Step 2b: rotate right cuv*
			0101	*Step 2c: circulant right shift x*
0	*0000*	*0000*	*0101*	*End of round 1*
				Round 2
0	*1100*			*Step 2a: add* $y \cdot 1$ *to u (0+12=12)*
0	*0110*	*0000*		*Step 2b: rotate right cuv*
			1010	*Step 2c: circulant right shift x*
0	*0110*	*0000*	*1010*	*End of round 2*
				Round 3
0	*0110*			*Step 2a: add* $y \cdot 0$ *to u (6+0=6)*
0	*0011*	*0000*		*Step 2b: rotate right cuv*
			0101	*Step 2c: circulant right shift x*
0	*0011*	*0000*	*0101*	*End of round 3*
				Round 4
0	*1111*			*Step 2a: add* $y \cdot 1$ *to u (3+12=15)*
0	*0111*	*1000*		*Step 2b: rotate right cuv*
			1010	*Step 2c: circulant right shift x*
0	*0111*	*1000*	*1010*	*End of round 4*

Note x is returned to its original value and uv $= 01111000_2 = 120_{10}$.

Example 10 *Multiply 14 and 7 in binary using algorithm 2*

First we need to convert our numbers to binary: $x = 14_{10} = 1110_2$ *and* $y = 7_{10} = 0111_2$.

c	*u*	*v*	*x*	*Comments*
0	*0000*	*0000*	*1110*	*Setup (Step 1)*
				Round 1
0	*0000*			*Step 2a: add* $y \cdot 0$ *to u (0+0=0)*
0	*0000*	*0000*		*Step 2b: rotate right cuv*
			0111	*Step 2c: circulant right shift x*
0	*0000*	*0000*	*0111*	*End of round 1*
				Round 2
0	*0111*			*Step 2a: add* $y \cdot 1$ *to u (0+7=7)*
0	*0011*	*1000*		*Step 2b: rotate right cuv*
			1011	*Step 2c: circulant right shift x*
0	*0011*	*1000*	*1011*	*End of round 2*
				Round 3
0	*1010*			*Step 2a: add* $y \cdot 1$ *to u (3+7=10)*
0	*0101*	*0100*		*Step 2b: rotate right cuv*
			1101	*Step 2c: circulant right shift x*
0	*0101*	*0100*	*1101*	*End of round 3*
				Round 4
0	*1100*			*Step 2a: add* $y \cdot 1$ *to u (5+7=12)*
0	*0110*	*0010*		*Step 2b: rotate right cuv*
			1110	*Step 2c: circulant right shift x*
0	*0110*	*0010*	*1110*	*End of round 4*

Note x is returned to its original value and $uv = 01100100_2 = 98_{10}$.

2.3.2 2's complement

Human's have tons of shortcuts to speed up our calculations, so it should come as no surprise that it is similar with digital circuits. One shortcut we often use in calculating things is based on estimating. Say you wanted to multiply 99 and 56. It would be easier to do it as $(100 - 1) * 56 = 5600 - 56 = 5544$. It would be no different if we wanted to multiply 99,099 and 56; just do $f(100,000 - 1,000 + 100 - 1) * 56 = 5,600,000 - 56,000 + 5,600 - 56 = 5,549,544$. This technique forms the basis of Booth's Algorithm, which works even nicer since we are dealing with binary. Consider, 0111_2 times 011_2. The first number can be written as $01000_2 - 01_2$, so we have $(01000_2 - 01_2) * 011_2 = 011000_2 - 011_2 = 010101_2$. We want to find a pattern to do this automatically, so lets consider a slightly bigger example: $01100111_2 * 011_2$ or $103 * 3$. We need to break up the first number, and we will add a radix point and do it in a table to make it easier to see something:

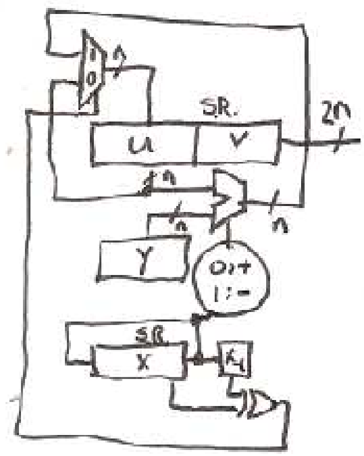
0	0	1	1	0	0	1	1	1 . 0
0	1	0	0	0	0	0	0	0 . 0
-	0	0	1	0	0	0	0	0 . 0
+	0	0	0	0	1	0	0	0 . 0
-	0	0	0	0	0	0	0	1 . 0

Notice that in the original number when the current digit is a zero and the previous was a 1 we add a 1 (I will highlight this in blue), and when the current digit is a 1 and the previous was a 0 we subtract 1 (I will highlight this in red):

0	0	1	1	0	0	1	1	1 . 0
0	1	0	0	0	0	0	0	0 . 0
-	0	0	1	0	0	0	0	0 . 0
+	0	0	0	0	1	0	0	0 . 0
-	0	0	0	0	0	0	0	1 . 0

I like this pattern, because 10_2 is a negative number in two's compliment and that is where I subtract,

Figure 2.4: Booth's Algorithm



and 01_2 is a positive number in two's compliment and that is where I add. It is thus memorable. Since we are multiplying the location of the 1's tell us where to add or subtract the other number. Thus we have in our example (with one extra column to fit the multiplied numbers):

0	1	1	0	0	0	0	0	0	0 . 0
-	0	0	1	1	0	0	0	0	0 . 0
+	0	0	0	0	1	1	0	0	0 . 0
-	0	0	0	0	0	0	0	1	1 . 0
0	1	0	0	1	1	0	1	0	1 . 0

So we find that $01100111_2 * 011_2 = 0100110101_2$ or $103 * 3 = 309$. It is worth noting a couple things in the resulting pattern. First, since we alternate addition and subtraction it is impossible to get overflow, and thus the carry bit isn't needed. Second, if we were multiplying by a negative number, like $-2_{10} = 10_2 = 110_2 = 1110_2$, we can note the leftmost of our 01 or 10 patterns is 10, which means subtract. The leftmost is the most significant, thus a negative number times a positive number would result in a negative. If you think about it a negative times a negative will result in a positive. This means our technique handles signed multiplication directly! Since this works nicely we want to generalize it, which is what we have as Booth's algorithm.

Booth's Algorithm

1. group two regs (u,v) for product, set to 0
2. set x_{-1} to 0 (this is a single bit)
3. for each digit do:
   (a) if (lsb of x is 1,) and (x_{-1}=0), subtract y from u
   (b) if (lsb of x is 0) and (x_{-1}=1), add y to u
   (c) arithmetic right shift (u,v)
   (d) circular right shift x

Booth's algorithm can be implemented in a circuit as is done in Figure 2.4.

Example 11 *Multiply* 6 *(*$x = 0110$*) and* -1 *(*$y = 1111$*) using Booth's algorithm. Show the values at each stage in a table.*

Booth's

u	v	x	x_{-1}
0000	0000	1111	0
1010	0000		
1101	0000	1111	1
1110	1000	1111	1
1111	0100	1111	1
1111	1010	1111	1

Note the answer is 11111010, which is -6 *in 2's complement.*

Example 12 *Multiply -3 and 5 using Booth's algorithm and 4 bit numbers. Perform the indicated calculations showing all steps.*

$y = 5 = 0101$

$-y = -5 = 1011$

u	v	x	x_{-1}
0000	0000	1101	0
1011			
1011	0000	1101	0
1101	1000	1110	1
0101			
0010	1000	1110	1
0001	0100	0111	0
1011			
1100	0100	0111	0
1110	0010	1011	1
1111	0001	1101	1

The result is 11110001, which is -15 *in 2's complement.*

Example 13 *Multiply -3 and -6 using Booth's algorithm and 4 bit numbers. Perform the indicated calculations showing all steps.*

$x = -3 = 1101$, $y = -6 = 1010$ *and* $-y = 6 = 0110$.

U	V	X	X_{-1}
0000	0000	1101	0
0110	0000	1101	0
0011	0000	1110	1
1101	0000	1110	1
1110	1000	0111	0
0100	1000	0111	0
0010	0100	1011	1
0001	0010	1101	1

$00010010 = 18$

2.3.3 Systolic Array

The preceding algorithms are $O(n^2)$ if implemented with ripple adders, $O(n \log(n))$ if implemented with conditional sum adders, or $O(n)$ if implemented with look-ahead adders. The look-ahead adders have a large constant, so the $O(n)$ is not a perfect indicator of performance, and they are currently not practical beyond about 8 bits. It would be nice to find a way to multiply that has $O(n)$ and a small constant multiplier.

Systolic arrays are $O(n)$, and have a constant multiplier of about 6 depending on your hardware, which is about half what it takes with even block (group) carry look-ahead adders using serial routines.

2.4 Integrated Examples

Example 14 *Calculate the following expression in binary using 2's complement and 8 bits total. Show all work.*

$$(9 * 9 - 24)/3$$

Sol:

$9_{10} = 00001001_2$ *and* $3_{10} = 00000011_2$

$$\begin{array}{r|l} 24 & \\ \hline 12 & 0 \\ 6 & 0 \\ 3 & 0 \\ 1 & 1 \\ 0 & 1 \end{array}$$

$24_{10} = 00011000_2$ *thus* $-24_{10} = 11101000_2$. *Thus* $9 * 9$,

$$\begin{array}{ccccccccccc} & & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ & & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ \hline & & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & & & \\ \hline & & & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

Then (subtracting 24),

$$\begin{array}{ccccccccc} & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}$$

Now perform the division:

$$\begin{array}{cc|cccccc} & & & & 1 & 0 & 0 & 1 & 1 \\ \cline{3-8} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ & & 1 & 1 & & & & \\ \cline{3-4} & & & 0 & 1 & 0 & 0 & \\ & & & & & 1 & 1 & \\ \cline{6-7} & & & & & & 1 & 1 \\ & & & & & & 1 & 1 \\ \cline{7-8} & & & & & & & 0 \end{array}$$

The answer is thus $00010011_2 = 19_{10}$.

2.5 Residue Arithmetic

We have shown different ways of calculating the sum and product of binary numbers. In this section we will examine a different way to represent numbers and thus to calculate. In residue arithmetic numbers are represented by their remainders of a group of numbers that constitute the basis of the representation. Let's consider a simple example of how numbers can be represented in this method.

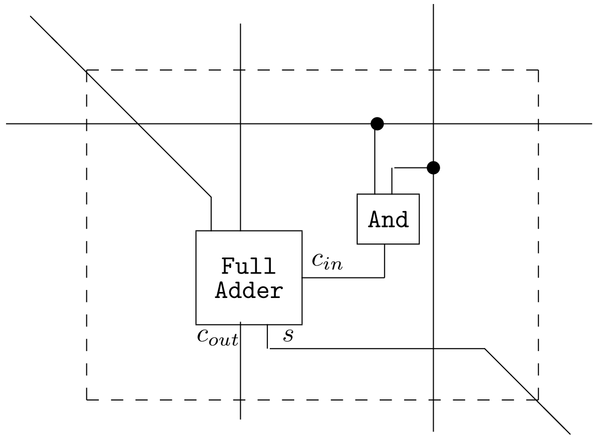


Figure 2.5: Individual Cell of Systolic Array

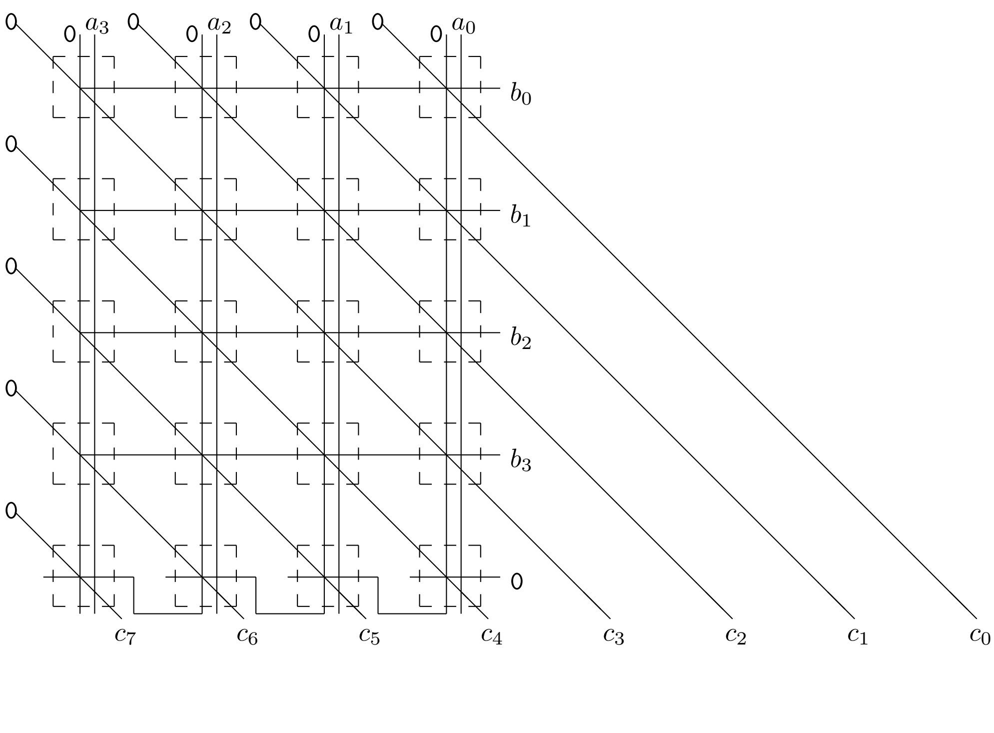


Figure 2.6: Systolic Array For 4 Bit Numbers

Number	%2	%3
0	0	0
1	1	1
2	0	2
3	1	0
4	0	1
5	1	2

Note that each of the numbers from 0 through 5 can be represented uniquely by their remainders. Note that the number 6 would be 0,0 and thus not distinguishable from 0. You can represent six numbers (1-5) because the product of the basis numbers is $2 \times 3 = 6$. That we can represent the numbers is one thing, being able to calculate easily is another. Lets consider addition first:

1=1,1	2=0,2
2=0,2	3=1,0
3=(0+1)%2,(1+2)%3	5=(0+1)%2,(2+0)%3
=1,0	=1,2

If you look up (1,0) in our table you will find it corresponds to 3, similarly (1,2) corresponds to 5. Now lets try some multiplication problems:

2=0,2	1=1,1
2=0,2	3=1,0
4=(0×0)%2,(2×2)%3	3=(1×1)%2,(1×0)%3
=0,1	=1,0

If you look up (0,1) in our table you will find it corresponds to 4, similarly (1,0) corresponds to 3. Subtraction is slightly more complex, similar to the 2's complement[3] an inverse of each remainder (the representation) must be found. This is done by subtracting each remainder from the number it was modulused from. This is easiest to see in an example.

Example 15 *First, let's get a table of the numbers and their negatives (additive inverses):*

Number Decimal	*Residue %2,%3*	*Negative %2,%3*	*Negative Decimal*
0	*0,0*	*(2-0)%2=0,(3-0)%3=0*	*0*
1	*1,1*	*(2-1)%2=1,(3-1)%3=2*	*5*
2	*0,2*	*(2-0)%2=0,(3-2)%3=1*	*4*
3	*1,0*	*(2-1)%2=1,(3-0)%3=0*	*3*
4	*0,1*	*(2-0)%2=0,(3-1)%3=2*	*2*
5	*1,2*	*(2-1)%2=1,(3-2)%3=1*	*1*

Now let's do some calculations.

$$\begin{aligned} 5-2 &= (1,2)-(0,2) \\ &= (1,2)+(0,1) \\ &= (1+0,2+1) \\ &= (1,0) \\ &= 3 \end{aligned}$$

[3]In fact it is a radix complement, in particular since for our example their are 6 numbers in our example, we will be calculating the 6's complement and then finding its residue.

$$\begin{aligned} 4-4 &= (0,1)-(0,1) \\ &= (0,1)+(0,2) \\ &= (0+0,1+2) \\ &= (0,0) \\ &= 0 \end{aligned}$$

$$\begin{aligned} 2-1 &= (0,2)-(1,1) \\ &= (0,2)+(1,2) \\ &= (0+1,2+2) \\ &= (1,1) \\ &= 1 \end{aligned}$$

The basis of the representation must be relatively prime, that is they must have unique prime factors (they cannot share prime factors with other basis numbers). This means that you can have a number like 4 (2×2) as long as no other basis had 2 as a factor, but you could not have 9 (3×3) and 12 ($2 \times 2 \times 3$), or 6 (2×3) and 10 (2×5) in the same basis. To see why consider the basis (4,6), it should give unique representations for $4 \times 6 = 24$ numbers (0-23).

Number	%4	%6	Number	%4	%6
0	0	0	12	0	0
1	1	1	13	1	1
2	2	2	14	2	2
3	3	3	15	3	3
4	0	4	16	0	4
5	1	5	17	1	5
6	2	0	18	2	0
7	3	1	19	3	1
8	0	2	20	0	2
9	1	3	21	1	3
10	2	4	22	2	4
11	3	5	23	3	5

Notice the first and second column are the same, and thus do not give us the full range we wanted.

Chapter 3

Floating Point

The main goal of this chapter is to introduce floating point numbers and the issues around their use and misuse. Toward that end, we will first cover fixed point numbers.

3.1 Fixed Point Numbers

Example:

Convert π to binary and hexadecimal. Assume you have four bits before the radix point and 8 bits after the radix point.

Sol:

before the decimal we have $3 = 0011$

after the decimal

0.1415926...	
0.2831852	0
0.5663704	0
1.1327408	1
0.2654816	0
0.5309632	0
1.0619264	1
0.1238528	0
0.2477056	0

combining gives 0011.00100100

To convert to hexadecimal we group the digits together in groups of four starting at the radix point, thus we are forcing the hexadecimal digits to represent either integer or fractional portions.

0011	0010	0100
3	2	4

Thus the answer is $0x3.24$.

Example:

Convert 25.6875 to binary.

25	/2	.	*2	.6875
12	1		1	.375
6	0		0	.75
3	0		1	.5
1	1		1	0
0	1			

11001.1011

3.2 Floating Point Numbers

I came up with the following program in my doctoral work at UCSB.

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

int main(){
    double pi, e, result;
    int i;

    e=exp(1);

    pi=atan(1)*4;

    result=pi;

    for(i=1;i<53;i++){
        result=sqrt(result);
    }

    for(i=1;i<53;i++){
        result=result*result;
    }

    cout << setiosflags(ios::showpoint | ios::fixed) << setprecision(16);
    cout << "Pi     = " << pi << endl;
    cout << "Result = " << result << endl;
    cout << "e      = " << e << endl;

    return 0;
}
```

The results are

```
Pi     = 3.1415926535897931
Result = 2.7182818081824731
e      = 2.7182818284590451
Press any key to continue
```

Notice that Result is e to 7 significant digits, but it should be π. This underscores the importance of being numerically aware when writing programs.

3.3 IEEE 754

Floating point numbers are based off scientific notation. Consider a typical number in base 10 scientific notation,

$$-1.23 \times 10^3.$$

The number is composed of five pieces of information,

1. sign of the number (-),
2. significant or mantissa (1.23),
3. base (10),
4. sign of the exponent (+),
5. magnitude of the exponent (3).

There are two basic number formats called out in IEEE 754, single precision (float in c/c++), and double precision (double in c/c++). In addition there are two extended formats, which are only used as intermediate results while calculating.

e	f	Category	Interpretation
$1\ldots11$	$1\ldots11$ $\vdots$ $0\ldots01$	NaN	See Codes
$1\ldots11$	$0\ldots00$	$\pm\infty$	$\pm\infty$
$1\ldots10$ $\vdots$ $0\ldots01$	$1\ldots11$ $\vdots$ $0\ldots00$	Numbers	$(-1)^s \times 1.f \times 2^{(e-127)}$
$0\ldots00$	$1\ldots11$ $\vdots$ $0\ldots00$	Denormals	$(-1)^s \times 0.f \times 2^{(-126)}$
$0\ldots00$	$0\ldots00$	±0	±0

NaN codes:

Dec	Meaning	Example
1	invalid square root	$\sqrt{-1}$
2	invalid addition	$\infty + -\infty$
4	invalid division	$\frac{0}{0}$
8	invalid multiplication	$0 \times \infty$
9	invalid modulo	$x mod 0$

For this discussion, the notation $fl(x)$ will be used to mean the number x as it is represented in floating point on a computer.

$$(-1)^s \cdot 1.f \times 2^{e-127}$$

<table>
<tr><td>0
1</td><td>0
2</td><td>0
3</td><td>0
4</td><td>0
5</td><td>0
6</td><td>0
7</td><td>0
8</td><td>0
9</td><td>1
0</td><td>1
1</td><td>1
2</td><td>1
3</td><td>1
4</td><td>1
5</td><td>1
6</td><td>1
7</td><td>1
8</td><td>1
9</td><td>2
0</td><td>2
1</td><td>2
2</td><td>2
3</td><td>2
4</td><td>2
5</td><td>2
6</td><td>2
7</td><td>2
8</td><td>2
9</td><td>3
0</td><td>3
1</td><td>3
2</td></tr>
<tr><td>s</td><td colspan="8">e</td><td colspan="23">f</td></tr>
</table>

This is equivalent to saying

$$(-1)^s \cdot 1.f \times 2^E$$

0	0 0 0 0 0 0 0 0	0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
1	2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
s	e=E+127	f

They are the same because $e - 127 = E$ is the same equation as $e = E + 127$. I think the latter is easier to use because you read E from the number and want e. The first form (standard for most texts) involves you guessing what number produced what you are seeing (rather than calculating it). It is like trying to solve $y = mx + b$ for y given x but using the form $\frac{(y-b)}{m} = x$ to do it. It works, just not well. In any case, consider some examples.

Example:

Convert 7.892 to single precision IEEE.

Step 1: Convert 7.892 to binary

$7.892 = 111.11100100010110100000111$

Step 2: Normalize and note sign

$7.892 = (-1)^0 1.1111100100010110100000111 \times 2^2$

Step 3: Calculate Excess 127 code for exponent

$e = 2 + 127 = 129 = 10000001$

Step 4:Round $1.f$ to 24 digits

$fl(1.1111100100010110100000111) = 1.11111001000101101000100$

Step 5: Assemble

0	1 0 0 0 0 0 0 1	1 1 1 1 1 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0 1 0 0

Example:

Calculate 3.75×29.625 in IEEE-754 single precision floating point.

Convert:

$3.75 = 11.11 = 1.111 \times 2^1$

$29.625 = 11101.101 = 1.1101101 \times 2^4$

Multiply Significants:

	1.	1	1	0	1	1	0	1			
×	1.	1	1	1							
	1.	1	1	0	1	1	0	1			
	0.	1	1	1	0	1	1	0	1		
	0.	0	1	1	1	0	1	1	0	1	
	0.	0	0	1	1	1	0	1	1	0	1
1	1.	0	1	1	1	1	0	0	0	1	1

1.10111100011×2^1

Add exponents to normalization exponent and put in excess 127:

$1 + 4 + 1 + 127 = 133 = 10000101$

Write in single precision:

0	10000101	1011 1100 0110 0000 0000 000

Example:

Perform the following for IEEE-754, single precision

1. Show the representation of $x = 93.3125$

   $x = 93.125_{10} = 1011101.001_2 = 1.011101001 \times 2^6$

0	1 0 0 0 0 1 0 1	0 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

2. calculate $x * y$ for y equal to

0	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

   exponent: 128+133-127=134

   float: shortcut, note that y only has two 1's in the expansion (hidden and near end) and they are farther apart than the length of the significant portion of x. This will cause the x float to be placed starting at these locations. The comma below notes where the last bit of precision lies.

   $$z_{fl} = 1.01110100100000000000101, 1101001$$

   Note that the first bit after the comma is a 1 so the number gets rounded up.

   z is

0	1 0 0 0 0 1 1 0	0 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0

Example:

Convert 3.03125 to IEEE single precision

3	.		03125
1	1	0	0625
0	1	0	125
		0	25
		0	5
		1	0

$3.03125_{10} = 11.00001_2 = 1.100001_2 \times 2^1$

$1 + 127 = 128$

0	1 0 0 0 0 0 0 0	1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Now perform the following on your result and

0	1 0 0 0 0 1 0 0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

1. Addition

   $x = 1.0000000100000001_2 \times 2^5$

   $y = 1.100001_2 \times 2^1 = 0.0001100001_2 \times 2^5$

   $$\begin{aligned} x + y &= 1.0000000100000001_2 \times 2^5 + 0.0001100001_2 \times 2^5 \\ &= (1.0000000100000001_2 + 0.0001100001_2) \times 2^5 \\ &= (1.0001100101000001_2) \times 2^5 \end{aligned}$$

0	1 0 0 0 0 1 0 0	0 0 0 1 1 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0

2. Multiplication

   exponent is $132 + 128 - 127 = 133$

   significant is $1.0000000100000001 \times 1.100001 = 1.1000010110000101100001$

0	1 0 0 0 0 1 0 1	1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0

Example:

Perform the following for IEEE-754, single precision

1. Show the representation of $x = 0.8125$

0	0 1 1 1 1 1 1 0	1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

2. calculate (show steps) $x * y$ for x from above and

y is

1	1 0 0 0 0 0 0 1	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Exponent: $(10000001 + 01111110) - 01111111 = 11111111 - 01111111 = 1000000$

float$= 1.101 * 1.11 = 10.11011 = 1.011011 \times 2^1$, so add 1 to exponent

1	1 0 0 0 0 0 0 1	0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

3. Perform the multiplication above in decimal and verify the answer.

$.8125 * (-7) = -5.6875 = -101.1011_2$

3.4 Rounding versus Chopping

Rounding is almost always used because of two reasons. To see both, let the interval between two numbers in the representation is 2δ then for rounding $x - fl(x) \in [-\delta, \delta)$, while for chopping it is $x - fl(x) \in [0, 2\delta)$. The first problem is that the error magnitude is up to twice as large for chopping. This is obviously bad, but it is not as bad as the second problem. The second problem is that all the errors of chopping have the same sign, so no error cancellation is possible when calculations are done. To see why this is bad, consider the following.

Example:

Find out the error in calculating $\sum_{i=1}^n x_i$ on a computer. First note that what you actually calculate is $\sum_{i=1}^n fl(x_i)$. The error (actual minus calculated) is thus $Err = |(\sum_{i=1}^n x_i) - (\sum_{i=1}^n fl(x_i))|$. Also let $fl(x_i) = x_i + \gamma_i$ for γ_i in the error interval of your method.

$$\begin{aligned}
Err &= \left|(\sum_{i=1}^n x_i) - (\sum_{i=1}^n (x_i + \gamma_i))\right| \\
&= \left|(\sum_{i=1}^n x_i) - (\sum_{i=1}^n x_i + \sum_{i=1}^n \gamma_i)\right| \\
&= \left|\sum_{i=1}^n x_i - \sum_{i=1}^n x_i - \sum_{i=1}^n \gamma_i\right| \\
&= \left|\sum_{i=1}^n \gamma_i\right| \\
&\leq \sum_{i=1}^n |\gamma_i|
\end{aligned}$$

For chopping the last inequality is actually an equality, i.e. chopping always has the worst case error. For a typical case on rounding the errors are distributed with some positive and some negative, thus cancellation can occur. For large sums (many terms) the law of large numbers and an assumed uniform distribution of γ_i indicates that the error for rounding will go to 0! This is a great result.

Example

Write C/C++ code to sum the following $\sum_{i=1}^{100} \frac{1}{i}$. Make sure you do it in the right order.

```
double sum=0;
int i;

for(i=100;i>=0;i--){
    sum+=1.0/i;}
```

3.5 Evaluating a Polynomial

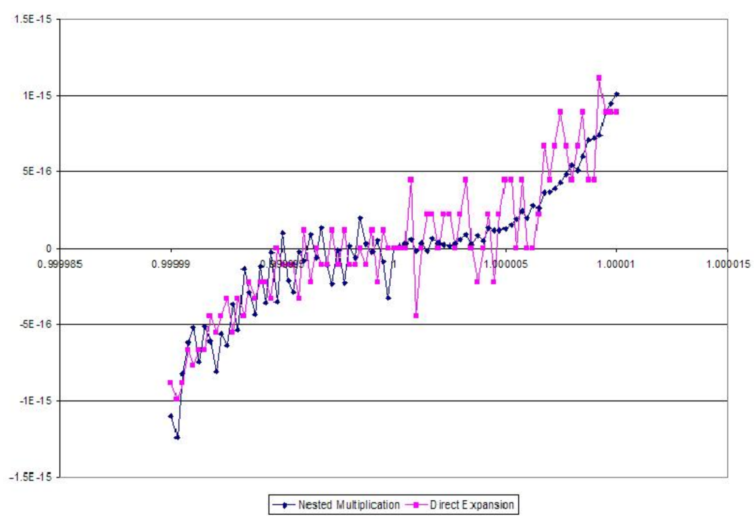


Figure 3.1: Close-up Look at Resulting Values of Two Evaluation Methods for $y = x^3 - 3x^2 + 3x - 1$

Part II

Organization

Chapter 4

Arithmetic Operations

We have looked at number representation and calculation techniques, now we will look at how to specify the operations to a computer. In order to do an arithmetic operation, we need to know where the two operands (sources) are located and where the result should be placed (destination). Computers are classified by how many of the addresses must be explicitly stated and how many are implicit.

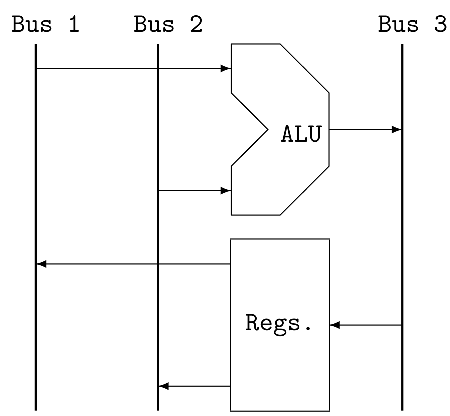
4.1 Three Address Machines

This is the most flexible form. Each address can be specified by the user. The commands are of the form

command source1, source2, destination

or

command destination, source1, source2



4.2 Two Address Machines

The destination is also a source in this case. The commands are of the form

command destination, source

4.3 One Address Machines

A special register, called the accumulator, is designated to be a source and destination. The accumulator has two special instructions, load accumulator and store accumulator. Accumulator machines rarely use additional registers, though it is not technically required. The arithmetic commands are of the form

command source

4.4 Zero Address Machines

The internal registers are arranged as a stack. The source operands are taken from the stack in order (first operand on top, second operand below). The result is pushed on the stack. These are often called stack machines. The arithmetic commands are of the form

command

4.5 Comparison Code

Consider the following equation:

$$\begin{aligned} y &= x^2 + 2x + 3 \\ &= (x+2) * x + 3 \end{aligned}$$

Assume x is at 100, 2 is at 104, 3 is at 108, and y is at 112. The following uses a three address scheme with destination first.

version 1 $y = x^2 + 2x + 3$	version 2 $y = (x+2) * x + 3$
mpy 112,100,100	add 112,100,104
mpy 116,100,104	mpy 112,112,100
add 112,112,116	add 112,112,108
add 112,112,108	

The following shows the second version on different machines.

3 address	2 address	1 address	0 address
add 112,100,104	move 112,100	load 100	push 100
mpy 112,112,100	add 112,104	add 104	push 104
add 112,112,108	mpy 112,100	mpy 100	add
	add 112,108	add 108	push 100
		store 112	mpy
			push 108
			add
			pop 112

Assume x is in R_1, 2 is in R_2, 3 is in R_3, and y is in R_4.

Chapter 5

Stack Machines

Stack machines are also known as 0-address machines, because no address must be specified for arithmetic operations. The most common example of a stack machine is an HP calculator. The application "Toy Stack" is an executable for Windows XP, which is available at the website. It has 64 bytes of memory split into 32 for instructions and 32 for data. All variables are 1 byte long and stored in 2's complement or unsigned form. Instructions are 1 byte long, but can have two commands in it in some cases. There is no branch delay slot. The commands are

Memory

0 0	P	Addr

where,

$$\begin{aligned} P &= \begin{cases} 0, & \text{Push;} \\ 1, & \text{Pop.} \end{cases} \\ Addr &= \text{5-bit address in memory.} \end{aligned}$$

Branching

0 1	C	Addr

where,

$$\begin{aligned} C &= \begin{cases} 0, & \text{Always;} \\ 1, & \text{Less (i.e. the top number on the stack is negative).} \end{cases} \\ Addr &= \text{5-bit address in memory to branch to.} \end{aligned}$$

Note: branch less is also branch bit set, for the most significant bit on the top of the stack.

Arithmetic

1 0	Op_1	Op_2

where,

$$Op_i = \begin{cases} 000, & \text{halt } (Op_1) \text{ or nop } (Op_2); \\ 001, & \text{addition;} \\ 010, & \text{subtraction;} \\ 011, & \text{negation;} \\ 100, & \text{unsigned multiplication;} \\ 101, & \text{signed multiplication;} \\ 110, & \text{unsigned division;} \\ 111, & \text{signed division.} \end{cases}$$

Note: Nop is no operation, and is used to allow, just one arithmetic command to execute rather than two. Halt is used to terminate the program run. If something other than nop is in Op_2 after a halt then that command is executed before termination.

Shifting

1	1	0	L/R	mode	times

where,

$$L/R = \begin{cases} 0, & \text{left shift;} \\ 1, & \text{right shift.} \end{cases}$$

$$mode = \begin{cases} 00, & \text{fill with 0's;} \\ 01, & \text{fill with 1's;} \\ 10, & \text{arithmetic shift;} \\ 11, & \text{circulant shift.} \end{cases}$$

$$times = \text{shift (1+times) bits (times is a two bit number).}$$

Push Signed Constant

1	1	1	0	Const

where, Const is a four bit number that is sign extended to eight bits and pushed on the stack.

Logic

1	1	1	1	0	Op

where,

$$Op = \begin{cases} 000, & \text{or;} \\ 001, & \text{nor;} \\ 010, & \text{orn;} \\ 011, & \text{xor;} \\ 100, & \text{and;} \\ 101, & \text{nand;} \\ 110, & \text{andn;} \\ 111, & \text{equivalence.} \end{cases}$$

Note: all logic functions are bitwise.

Undefined

1	1	1	1	1	Op

where, Op is a three bit operand. This operation is left undefined.

At the moment you have to enter your programs and data values manually, sorry I just started writing this. A load and save feature has been added which saves the memory to a file in encrypted format. You can only load programs that were encrypted with your exact name (spelling and caps count). Essentially this removes sharing data files as you need to submit your solutions electronically to me, with the exact spelling of your name (so I can load them). I will not give credit to you unless the name is yours.

5.1 Affine Encryption Program

Affine encryption is one of the simplest methods for doing encryption. Let P_i be the i^{th} character in the plain text message, and let C_i be the corresponding encoded character. Let there be n possible characters to encode, then the basic idea is to pick two numbers (a, b) to encode a message such that $\gcd(a, n) = 1$ (so a has an inverse). No requirement on b is needed if your modulus function has been encoded correctly. The encoded character can then be found by

$$a \times P_i + b = C_i \mod n.$$

Note that the " mod n" at the end says the equation holds in $\mathbb{Z}_n$, the set of integers mod n with appropriately defined arithmetic.

To decrypt the message, the equation

$$\bar{a} \times (C_i + d) \quad = \quad P_i \mod n$$

is used. The term $\bar{a}$ is the inverse of a in $\mathbb{Z}_n$, which is found by solving

$$\begin{aligned} a \times \bar{a} &= 1 \mod n \\ &\textbf{or} \\ a \times \bar{a} &= m \times n + 1. \end{aligned}$$

Note that m is any whole number. The term d is the additive inverse of b in $\mathbb{Z}_n$, which is found by solving

$$d = n - (b \mod n).$$

We can summarize this by saying an affine cipher is an encryption technique that encodes using three integers: a, b, and n. If *plain* is the character to be encoded (with 'A'=0 and 'Z'=25) then $code = (a{*}plain{+}b) \mod n$. Decoding is also done using three integers: c, d, and n. If *code* is the character to be encoded (with 'A'=0 and 'Z'=25) then $plain = (c * (code + d)) \mod n$. The requirements on (a, b, c, d, n) are:

- $\gcd(a, n) = 1$
- $(ac) \mod n = 1$
- $(b + d) \mod n = 0$

Below is C code to implement a particular case of affine cyphers.

```
char affine_encode(char plain){
    // affine codes capital letter in plain using a=5, b=12 thus this is modulo 26
    int iCode, iPlain, a=3,b=0;

    // convert char to integer and shift so A=0
    iPlain=int(plain)-65;

    // do the encoding
    iCode = (a*iPlain+b)%26;

    // return the result as a char
    return char(iCode+65);
}

char affine_decode(char code){
    // affine decodes capital letter in plain using c=21, d=8 thus this is modulo 26
    int iCode, iPlain, c=9, d=0;

    // convert char to integer and shift so A=0
    iCode=int(code)-65;

    // do the decoding
```

```
    iPlain = (c*(iCode+d))%26;

    // return the result as a char
    return char(iPlain+65);
}
```

Using this we consider affine encryption for standard ASCII including the control codes. In this case $n = 2^7 = 128$. Note that the standard arithmetic on our stack machine is $\mathbb{Z}_{2^8}$ so we can calculate normally then drop the leading bit to get $\mathbb{Z}_{2^7}$. As long as a does not have 2 as a factor it will meet the requirement $\gcd(a, n) = 1$. Let $a = 3$ then $3 \times \bar{a} = m \times n + 1$ for some $m \in \{1, 2, \ldots\}$. Start with $m = 1$, then $\bar{a} = 129/3 = 43$. Since the result is an integer, it is an inverse. If the result was not an integer, m would be incremented and the process would continue. Finally, let $b = 57$ then $d = 128 - 57 = 71$.

Let the memory locations of the variables be:

Variable	Address	Value
P	00000	your choice
C	00001	per calculation
P(calc)	10000	per calculation
a	11100	00000011
$\bar{a}$	11101	00101011
b	11110	00111001
d	11111	01000111

The variable P(calc) was added so the decoded plain text would not overwrite the original. The program to encode is thus:

Machine	Assembly	;Comment
00011110	push b	;load data
00011100	push a	;
00000000	push P	;
	unsigned multiply	;aP+b
10100001	add	;
11000000	shl0 1	;drop leading bit
11010000	shr0 1	;
00100001	pop C	;store
10000000	halt	;done

The program to decode is thus:

Machine	Assembly	;Comment
00011101	push $\bar{a}$	;load data
00011111	push d	;
00000001	push C	;
	add	;$\bar{a}(C + d)$
10001100	unsigned multiply	;
11000000	shl0 1	;drop leading bit
11010000	shr0 1	;
00110000	pop P(calc)	;store
10000000	halt	;done

5.2 Babylonian Algorithm

Implement the following Babylonian algorithm to find Pythagorean Triples[1] on the Toy Stack.

- Start with 2 (unsigned) integers p, q with $p > q$ (assume these are present)

[1]The algorithm actually predates Pythagoras.

- calculate the three numbers by: $n_1 = 2pq$, $n_2 = p^2 - q^2$, $n_3 = p^2 + q^2$

To understand how this works note that

$$\begin{aligned} n_1^2 &= (2pq)^2 \\ &= 4p^2q^2 \end{aligned}$$

and

$$\begin{aligned} n_2^2 &= (p^2 - q^2)^2 \\ &= p^4 - 2p^2q^2 + q^4 \end{aligned}$$

and

$$\begin{aligned} n_3^2 &= (p^2 + q^2)^2 \\ &= p^4 + 2p^2q^2 + q^4 \end{aligned}$$

thus

$$\begin{aligned} n_1^2 + n_2^2 &= (4p^2q^2) + (p^4 - 2p^2q^2 + q^4) \\ &= p^4 + 2p^2q^2 + q^4 \\ &= n_3^2 \end{aligned}$$

The assembly is

```
push 0   ! calculate 2pq
push 1
push #2
umul
umul
pop 16   ! 2pq stored in 16
push 0   ! calculate p^2
push 0
umul
pop 2    ! p^2 stored in 2
push 1   ! calculate q^2
push 1
umul
pop 3    ! q^2 stored in 3
push 3   ! calculate p^2 - q^2
push 2
sub
pop      ! p^2 - q^2 stored in 17
push 3   ! calculate p^2 + q^2
push 2
add
pop      ! p^2 + q^2 stored in 17
```

For the machine code see the website.

Chapter 6

Instruction Set Architecture

6.1 RISC vs. CISC

RISC reduced instruction set computer- For high level language programmers (reduces time for each instruction)

CISC complex instruction set computer- For assembly programmers (reduces instructions for same program)

	RISC	CISC
Number of addressing modes	few	many
Access to main memory	Only in loads and stores (hence load-store architecture)	One or more operands in most instructions can access
Size of instruction set	small	large
Complexity of each instruction	small	large

RISC is currently and has been more efficient.

6.2 Memory Access

Most machines are byte addressable (i.e. each byte in memory has an address). Memory access typically come in three sizes and are often distinguished by the operand suffix .b (byte), .h (halfword), .w (word).

6.3 Branching

Conditional branching

Three ways: compare two, compare to zero, condition registers
cmp
Branch delay and pipelining
short circuit (positional) put in sum of expressions form and then do a series of conditional branches
Bitwise (and,or,xor,andn,orn)
bb (bitbranch reg,bit,targ)
bset
bclr
shift L/R

zero fill
one fill
rotate
usually to carry

Chapter 7

Addressing

- .bss
- .data
- .text

.bss (block started by symbol) memory, reserved only

.data memory, predefined values

.text instructions

.reserve val (alternately ".skip val") sets aside val bytes of memory

.equate name, val (alternately ".set name, val") makes name a constant with value val

.byte val (alternately .b, ub, sb) specifies the operation to be on a byte

.half val (alternately .h, uh, sh) specifies the operation to be on a half word (2 bytes)

.word val (alternately .w) specifies the operation to be on a word (4 bytes)

.align val aligns the memory location counter

Note that val may be a constant expression for readability.

Name	Generic	Sparc	Uses
memory direct	mX	[%r0+X]	
register direct	rX	%rX	
immediate	#X	X	
memory indirect	@mX	-	pointers
register indirect	@rX	[%rX]	pointers
memory indexed	label[mX]	-	arrays
register indexed	label[rX]	[%rY + %rX]	arrays
			(note %rY is loaded with label)
pre-increment	+[rX]	-	increments by size (stride) each time
post-increment	[rX]+	-	increments by size (stride) each time
pre-decrement	-[rX]	-	decrements by size (stride) each time
post-decrement	[rX]-	-	decrements by size (stride) each time
memory displaced	mX $\rightarrow$ label	-	struct
register displaced	rX $\rightarrow$ label	[%rX + label]	struct

m0	0x00	0x00	0x00	0x12
m4	0x00	0x00	0x00	0x08
m8	0x01	0x23	0x45	0x67
m12	0x89	0xAB	0xCD	0xEF
m16	0x12	0x34	0x56	0x78
m20	0x9A	0xBC	0xDE	0xF0
m24	0x11	0x11	0x11	0x11

r0	0x00	0x00	0x00	0x00
r1	0x00	0x00	0x00	0x08
r2	0x00	0x00	0x00	0x0C
r3	0x00	0x00	0x00	0x04
r4	0x00	0x00	0x00	0x10

Let var1 be a label for the value 8.

Representation	X=4	Effective Address	Expression
mX	m4	0x00000004	0x00000008
rX	r4	-	0x00000010
#X	#4	-	0x00000004
@mX	@m4	0x00000008	0x01234567
@rX	@r4	0x00000010	0x12345678
var1[mX]	8[m4]	0x00000010 (i.e.: 8+8)	0x12345678
var1[rX]	8[r4]	0x00000018 (i.e.: 8+16)	0x11111111
+[rX]	+[r4]	0x00000014	0x9ABCDEF0
			r4 ← 0x00000014 before
[rX]+	[r4]+	0x00000010	0x12345678
			r4 ← 0x00000014 after
-[rX]	-[r4]	0x0000000C	0x89ABCDEF
			r4 ← 0x0000000C before
[rX]-	[r4]-	0x00000010	0x12345678
			r4 ← 0x0000000C after
mX → var1	m4 → 8	0x00000010 (i.e.: 8+8)	0x12345678
rX → var1	r4 → 8	0x00000018 (i.e.: 8+16)	0x11111111

7.0.1 Arrays

For instance consider an array of 10 integers.

```
int my_int[10];
```

This creates both the array of integers and a pointer to the first element. The elements are numbered 0 to 9 and are accessed by $my_int[i]$ for $i \in \{0, 1, \ldots, 9\}$. They can also be accessed by $*(my_int + i)$. In assembly we would have:

```
my_int:  .skip 10*4    ; each int is 4 bytes
```

The contents can be accessed by:

```
set i, %r2
ld  [%r2], %r2
umul %r2, 4, %r3
set my_int, %r4

ld [%r4 + %r3], %r5
```

or if my_int (the address) fits in a 13 bit signed constant:

```
set i, %r2
ld  [%r2], %r2
umul %r2, 4, %r3

ld [%r3+my_int], %r5
```

Essentially the address is my_int + i*4, but this assumes that start of my array is zero. How about a language like Pascal or VB which allows other starting values? Consider defining an array (-m,-m+1,..., -1, 0, 1, ..., n). To use the address my_int + i*size we have

```
.skip m*size       ! negatives
.skip (n+1)*size   ! zero and positives
```

Alternately,

```
.skip (m+n+1)*size  ! whole thing
```

This causes the address to be my_int + (i+m)*size. Now you might think this will be longer, but note that it can be rewritten as

my_int + (i+m)*size
my_int + i*size + m*size
(my_int + m*size) + i*size

That is, rather than constantly biasing the index, it makes more sense to bias the base. Essentially it makes the second method look like the first, but it works for a positive starting number (by making m a negative). Since it is more general the later form is what is used in practice.

7.0.2 String Storage

string256 (aka length plus value) length of string in first byte, string following

NULL terminated string followed by 0

7.0.3 Structs

```
struct book{
  int pages;
  float price;
  char title[20];
}library[100];
```

Would be implemented:

```
.set pages, 0
.set price, 4
.set title, 8
.set book_size, 28
```

```
.bss

library: .skip 100*book_size
```

.bss is done in .data on some assemblers or machines

Chapter 8

Subroutines

8.1 Basic Overview

Before we get into this, let's establish some basic definitions.

Caller the section of code that initiates the call

Callee the section of code that is called

Return Address The address of the instruction to be executed after the call is done (usually the one following the branch or jump)

Subroutine Linkage data structure used to share information between caller and callee

8.1.1 What needs to be passed?

A subroutine can be called from different sections of code and with different parameters. The subroutine needs to know what data it must operate on and where to resume execution when it finishes. Additionally the subroutine usually must return some data, and thus it must place the data in an easy to locate area. The basic data that must be exchanged is thus,

- return address
- return value
- parameters

8.1.2 General Call Sequence

Caller		Callee
Startup	$\rightarrow$	Prologue
Sequence		Body
Cleanup		Body
Sequence	$\leftarrow$	Epilogue

8.2 Return Addresses in Leaf and Non-Leaf Subroutines

For the moment we will look only at the issues surrounding return addresses. The following distinctions must be made:

> Leaf subroutines do not make subroutine calls, where as non-leaf subroutines call at least one subroutine (itself or another subroutine).

The most basic leaf subroutine call looks like:

Caller		Callee
address+(4 or 8) to r31	→	none
branch sub		Body
none		Body
none	←	branch @r31

The basic leaf routine is quick and easy, but it cannot be used on non-leaf procedures as the return address would be lost. Consider the following subroutine to calculate x^n:

Code

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! name: pow
!! desc: calculates x^n
!! meth: recursive function call
!!            x*(x^{n-1})
!! parm: x in r8
!!       n in r9
!! pre : nothing in r16, it is used as
!!       a temporary variable
!! post:
!! ret : x^n in r8
!! date: 20 May 2003
!! rev : 1.0
!! revh:
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
pow:        cmp r9,r0         ! see if x^0
            breq,a pow_done   !  if n=0
            add r0,1,r8       !  then ans=1

            cmp r9,1          ! see if x^1
            breq pow_done     ! if n=1
            nop               ! then ans=x

            mv r8,r16         ! else n>1
            call pow          ! calc r8=x^{n-1}
            sub r9,1,r9       !
pow_r:      smul r16,r8,r8    ! ans = x*x^{n-1}
pow_done:   retl
            nop
```

Sample run

Assume the call was to calculate 5^2 and return to the label "retn". For our machine the return address is stored in r31. We will assume that annulled commands become nop's (they really do, the results are just sent to r0 and the condition codes are not set).

Instruction	r8	r9	r16	r31
cmp r9,r0	5	2	-	retn
breq,a pow_done	5	2	-	retn
nop	5	2	-	retn
cmp r9,1	5	2	-	retn
breq pow_done	5	2	-	retn
nop	5	2	-	retn
mv r8,r16	5	2	5	retn
call pow	5	2	5	pow_r

Notice at this point we lost the return address!

Instruction	r8	r9	r16	r31
sub r9,1,r9	5	1	5	pow_r
cmp r9,r0	5	1	5	pow_r
breq,a pow_done	5	1	5	pow_r
nop	5	1	5	pow_r
cmp r9,1	5	1	5	pow_r
breq pow_done	5	1	5	pow_r
nop	5	1	5	pow_r
retl	5	1	5	pow_r
nop	5	1	5	pow_r
smul r16,r8,r8	25	1	5	pow_r
retl	25	1	5	pow_r
nop	25	1	5	pow_r

At this point it should have gone back to "retn" but since that address was lost it will loop endlessly.

If the subroutine is non-leaf and not part of a cycle (recursive or otherwise) then the following modification will work nicely.

Caller		Callee
address+(4 or 8) to r31	→	r31 to mem
branch sub		Body
none		mem to r31
none	←	branch @r31

the two versions can be combined as:

Caller		Callee
address+(4 or 8) to r31	→	if nonleaf r31 to mem
branch sub		Body
none		if nonleaf mem to r31
none	←	branch @r31

8.3 Parameter Passing

We now turn our attention on the parameters. First we need to consider how to represent the data. For instance if you just need to send an integer to do a calculation but you don't want it modified then you would pass by value. If on the other hand you need to pass an instance of a class you must pass by reference. The three ways data may be handled are

1. pass by value (not returned)
2. pass by value/result (modify and return)
3. pass by ref (pointer to actual object)

Beyond these basic considerations, there is a question as to where to locate the data for the subroutine call. The information could be located in the registers for speed, or in static variables in RAM (parameter block). Neither of the options discussed so far will handle cyclic subroutines or dynamic local variables. If either cyclic subroutines or dynamic local variables are needed the information must be passed on the stack (dynamic variables in RAM). The methods are:

1. register
   - fast
   - leaf subroutine
2. parameter block
   - larger data
   - non-leaf and non-cyclic subroutines
3. stack
   - larger data
   - (dynamic) local variables
   - cyclic and recursive calls

8.4 Register

Caller		Callee
mv params into r8 to r13		
address+(4 or 8) to r31		none
branch sub	↗	Body
none		mv result to r8
none	↖	branch @r31

Example

We have discussed affine ciphers already. You might have noticed that the equation for encoding and decoding is very similar. We can combine them with only a small alteration to the decoding formula and one of the requirements. Decoding is still done using three integers: c, d, and n. If *code* is the character to be decoded (with 'A'=0 and 'Z'=25) then $plain = (c * code + d) \mod n$. The requirements on (a, b, c, d, n) are:

- $\gcd(a, n) = 1$
- $(ac) \mod n = 1$
- $(cb + d) \mod n = 0$

Below is C code to implement a particular case of affine cyphers.

```
char affine(char letter, int scale, int offset){
    // affine codes capital letter in 'letter' thus this is modulo 26
    int iCode, iLetter;

    // convert char to integer and shift so A=0
    iLetter=int(plain)-65;

    // do the encoding
    iCode = (scale*iLetter+offset)%26;

    // return the result as a char
    return char(iCode+65);
}
```

The SPARC syntax is then
affine

```
            ! calculates affine encryption:
            !    crypt = (a*(orig-off)+b) mod p + off
            ! a     is passed   in r8
            ! b     is passed   in r9
            ! n     is passed   in r10
            ! off   is passed   in r11
            ! orig  is passed   in r12
```

```
        ! crypt is returned in r
        .text
affine: sub  r12, r12, r11  ! orig-off
        mult r8,  r12, r8   ! a*(orig-off)
        add  r8,  r8,  r9   ! a*(orig-off)+b
        div  r9,  r8,  r10  ! x= y mod z = y - y/z*z
        mult r9,  r9,  r10
        sub  r8,  r8,  r9   ! (a*(orig-off)+b) mod n
        add  r8,  r8,  r11  ! done
        retl
```

encrypt call

```
! affine encrypt
! a is passed in r8
! b is passed in r9
! n     is passed   in r10
! off   is passed   in r11
! orig  is passed   in r12
! crypt is returned in r8
.text
set r8, 3            ! given
set r9, 0            ! given
set r10, 26          ! letters in alphabet
set r11, 65          ! A in ascii
call affine          ! call and link
ld.b r12, add_plain  ! assume have label add_plain
                     !   where plain text is stored
st.b r8,  add_code   ! assume have label add_code where
                     !   cypher text is to be stored
```

decrypt call

```
! affine decrypt
! a is passed in r8
! b is passed in r9
! n     is passed   in r10
! off   is passed   in r11
! orig  is passed   in r12
! crypt is returned in r8
.text
set r8, 9            ! given
set r9, 0            ! given
set r10, 26          ! letters in alphabet
set r11, 65          ! A in ascii
call affine          ! call and link
ld.b r12, add_code   ! assume have label add_code
                     !   where cypher text is stored
```

```
          st.b r8,  add_plain  ! assume have label add_code where
                               !   plain text is to be stored
```

Example

Write the MIPS assembly code for the following function. Assume the array a has been defined as size **n**. The following registers are to be used to pass the values:

pointer to a	`$a0`
n	`$a1`
sum	`$v0`

You do not need to write the code to call the function.

```
int sum(int* a, int n){
  int sum;
  sum=0;
  for(int i=0;i<n;i++){
    sum+=a[i]}
  return sum;}
```

Solution

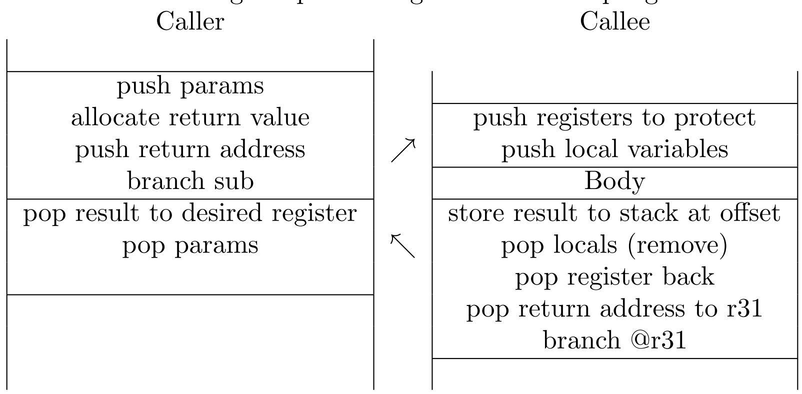
```
sum:
  add $v0, $zero, $zero   # sum=0
  sll $a1, $a1, 2         # 4*n
  add $a1, $a1, $a0       # one element after last in array
  ble $a1, $a0, sum_done  # array empty
sum_loop:
  lw $t0, 0($a0)          # get element
  addi $a0, $a0, 4        # increment pointer
  add $v0, $v0, $t0       # add element to sum
  bne $a0, $a1, sum_loop  # check if more elements
sum_done:
  jr $ra                  # return
```

8.5 Parameter Block

Caller		Callee
store params into block using labels		allocate block and labels in .data
store address+(4 or 8) to block		none
branch sub	↗	Body
load result to desired register		store result to block
none	↖	ld return address to r31
none		branch @r31

8.6 Stack

The stack is a large block of RAM which data is pushed onto. Any piece of information can be pushed onto the stack. All the data passed to and from the subroutine with all the local variables composes a block of information on the stack called the frame. The frame is created in the startup and prologue and removed in the epilogue and cleanup. The startup allocates space for all the information that must be passed (return address, parameters, and return values), and the cleanup removes it. The prologue allocates any local variables or storage to protect registers and the epilogue removes this local information.




```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! name: pow
!! desc: calculates x^n
!! meth: recursive function call
!!            x*(x^{n-1})
!! parm: stack passing:
!!       x              at fp+20
!!       n              at fp+16
!!       return value   at fp+12
!!       return address at fp+8
!! pre :
!! post:
!! ret : x^n at fp+12
!! date: 22 May 2003
!! rev : 1.1
!! revh:
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            .set s16,0       ! offset to save r16
            .set s17,4       ! offset to save r17
            .set ra,8        ! offset to ret add
            .set rv,12       ! offset to ret val
            .set n,16        ! offset to n
            .set x,20        ! offset to x
pow:        sub sp,8,sp      ! allocate save space
            mv sp,fp         ! set frame
            st r16,[fp+s16]  ! save r16
            st r17,[fp+s17]  ! save r17

            ld [fp+n],r17    ! load n
```

```
            cmp r17,r0       ! see if x^0
            breq,a pow_done  !  if n=0
            add r0,1,r16     !  then ans=1

            cmp r17,1        ! see if x^1
            breq pow_done    !  if n=1
            ld [fp+x],r16    !  then ans=x

                             ! else n>1
            sub sp,4,sp      !  decrement pointer
            st r16,[sp]      !  push x
            sub r17,1,r17    !  calc n-1
            sub sp,4,sp      !  decrement pointer
            st r17,[sp]      !  push n-1
            sub sp,8,sp      !  decrement pointer
                             !  for return value
                             !  and address
            call pow         !  calc r8=x^{n-1}
            st r31,[sp]      !  push return address

            ld [sp],r16      !  get x^{n-1}
            add sp,12,sp     !  deallocate
            mv sp,fp         !  restore frame

            ld [fp+x],r17    !  get x
            smul r16,r17,r16 !  ans = x*x^{n-1}

pow_done:   st r16,[fp+rv]   ! store return value
            ld [fp+s16],r16  ! restore r16
            ld [fp+s17],r17  ! restore r17
            ld [fp+ra],r31   ! get return address
            retl
            add sp,12,sp     !  deallocate ra, s16, s17
```

8.7 Temperature Conversion

Write a function that converts Fahrenheit to Celsius by following the steps below. A C/C++ command to do the conversion is:

```
celsius = ((fahrenheit - 32)* 5) / 9;
```

Note: I added an extra set of parenthesis to let you know you must do the multiplication first! Why does the multiplication have to be done first? Include an example.

If you do not multiply first, you can loose precision. ex: 2/9*5=0, while 2*5/9=1 (in integer math).

1. State the passing convention you will use (include what needs to be passed and where you will pass it) and any other reasonable assumptions on the machine.

I will use register passing and will use register r8 to pass both the parameter and the result. Since this is a leaf procedure and I do not need other registers, I will use the book's leaf procedure (return address in r31). I will further assume that my machine has call and retl that automatically store and access the return address. Finally, I will assume there is a branch delay slot, the destination is always the first location, and I have all addressing modes. (your choices may be different).

2. Write the function.

```
fahr_2_cels:   sub r8, r8, 32
               mpy r8, r8, 5
               retl
               div r8, r8, 9
```

3. Show how it would be called. Assume that the Fahrenheit temperature is stored in a memory location specified by the label "fahr_temp". The result should be stored at the memory location specified by the label "cels_temp".

```
set r1, fahr_temp
call fahr_2_cels
ld.w r8, @r1
set r1, cels_temp
st.w @r1, r8
```

Chapter 9

MIPS Assembly

R-Format	op	rs	rt	rd	shamt	funct
Bits	6	5	5	5	5	6
add \$r1,\$r2,\$r3	0	\$r2	\$r3	\$r1	0	32
addu \$r1,\$r2,\$r3	0	\$r2	\$r3	\$r1	0	33
sub \$r1,\$r2,\$r3	0	\$r2	\$r3	\$r1	0	34
subu \$r1,\$r2,\$r3	0	\$r2	\$r3	\$r1	0	35

I-Format	op	rs	rt	address
Bits	6	5	5	16
lw \$r1,off(\$r2)	35	\$r2	\$r1	off
sw \$r1,off(\$r2)	43	\$r2	\$r1	off

9.1 Registers

Number	Name	Use
0	$zero	0
1	$at	assembler use
2	$v0	return value (value)
3	$v1	return value (value)
4	$a1	parameters (arguments)
5	$a2	parameters (arguments)
6	$a3	parameters (arguments)
7	$a4	parameters (arguments)
8	$t0	temp (not saved)
9	$t1	temp (not saved)
10	$t2	temp (not saved)
11	$t3	temp (not saved)
12	$t4	temp (not saved)
13	$t5	temp (not saved)
14	$t6	temp (not saved)
15	$t7	temp (not saved)
16	$s0	saved temp
17	$s1	saved temp
18	$s2	saved temp
19	$s3	saved temp
20	$s4	saved temp
21	$s5	saved temp
22	$s6	saved temp
23	$s7	saved temp
24	$t8	temp (not saved)
25	$t9	temp (not saved)
26	$k0	OS
27	$k1	OS
28	$gp	global pointer (0x10008000) points to middle of 64k block
29	$sp	stack pointer
30	$fp	frame pointer
31	$ra	return address

9.2 Keeping Your Ends Straight

Big (LR) and little (RL) endian

Consistent (same for bits)

Sparc is inconsistent big-endian.

Endian	Consistent				Inconsistent			
	0	1	…	n	0	1	…	n
Big	0…7	0…7	…	0…7	7…0	7…0	…	7…0
	n	…	1	0	n	…	1	0
Little	7…0	7…0	…	7…0	0…7	0…7	…	0…7

9.3 Data Structures

Implement the following data structure in assembly then write a MIPS function to calculate $mykey.block = mykey.p \times mykey.q$.

```
struct keys{
   int p;
   int q;
   int public;
   int private;
   int block;
};
```

```
.data
mykey:
mykey_p: .word 0
mykey_q: .word 0
mykey_public: .word 0
mykey_private: .word 0
mykey_block: .word 0
.set mykey_off_p=mykey_p - mykey
.set mykey_off_q=mykey_q - mykey
.set mykey_off_public=mykey_public - mykey
.set mykey_off_private=mykey_private - mykey
.set mykey_off_block=mykey_block - mykey

.text
! Since this operates on data we know the location of,
!  we don't need to pass anything
la $t1, mykey
lw $t2, mykey_off_p($t1)
lw $t0, mykey_off_q($t1)
mul $t0,$t2
mflo $t0
sw $t0,mykey_off_block($t1)
```

9.4 Register Passing

9.4.1 Exponentiation by Multiplication

Write code to calculate n^m for n a non-zero finite integer and m a non-negative integer.

```
# n^m by loop
# n !=0 finite in a0
# m >=0 finite in a1
# n^m          in v0
# 0 in
pow_by_loop:
```

```
  # ensure arguments are ok
  mov  $v0,$zero
  beqz $a0,pow_done
  bltz $a1,pow_done
  # m=0 and setup
  addi $v0,$v0,1
  beqz $a1,pow_done
  # m>0, loop
pow_loop:
  mul  $v0,$a0
  mflo $v0
  subi $a1,$a1,1
  bgtz $a1,pow_loop
pow_done:
  jr $ra
```

Now how do we call it? Assume that n is in \$s0 and m is at address "int_m" and we want the result in \$s1.

```
mov $a0,$s0
la  $t1,int_m  # note I use $t1 for address scrap space
lw  $a1,0($t1)
jal pow_by_loop:
mv $s1, $v0
```

9.4.2 Polynomial Evaluation

Write the MIPS assembly code for the following function. Assume the array a has been defined as size n+1. You do not need to write the code to call the function but you need to state where you assume the parameters and return address will be.

```
int poly_eval(int* a, int n, int x){
  y=a[n];
  for(i=n-1;i>=0;i--){
    y=y*x+a[i];
  }
  return y;
}
```

```
############################################
# poly_eval
# leaf procedure to evaluate polynomials
# parameters:
# a1 : pointer to array of coefficients
# a2 : largest index in array
# a3 : point to evaluate polynomial
# return value:
# v0 : value of polynomial
# temporary values:
```

```
# t0 : offset in array
# t1 : address in array
poly_eval:  add  $t0, $a2, $a2         # four bytes per integer
            add  $t0, $t0, $t0
            add  $t1, $t0, $a1         # address of element to get
            lw   $v0,0($t1)            # initialize the answer
            beq  $t0,$zero, poly_done  # if only one element then done
poly_do:    mul  $v0, $v0, $a3         # y=y*x
            subi $t0, $t0, 4           # next coefficient is four bytes down
            add  $t1, $t0, $a1         # next coefficient's address
            lw   $t2, 0($t1)           # next coefficient
            add  $v0, $v0, $t2         # add next coefficient
            bne  $t0,$zero, poly_do    # more coefficients left
poly_done:  jr   $ra                   # return
```

9.4.3 Xor Encryption

Consider the problem of xor encryption. The i^{th} cipher text character, C_i is given by

$$C_i = P_i \oplus K_i$$

where P_i is the i^{th} plain text character and K_i is the i^{th} key character. The decryption is then given by

$$P_i = C_i \oplus K_i.$$

This encryption method is thus symmetric.

```
#
#  xor
#
# $a0 contains plaintext
# $a1 contains key
# $a2 contains ciphertext
xor:
  mov $t3,$a1
  lb $t0,0($a0)
  lb $t1,0($a1)
xor_loop:
  xor $t2,$t0,$t1
  sb  $t2,0($a2)
  addi $a0,$a0,1
  addi $a1,$a1,1
  addi $a2,$a2,1
  lb $t0,0($a0)
  beqz $t0, xor_done
xor_load:
  lb $t1,0($a1)
  bgtz $t1, xor_loop
  mov $a1,$t3
```

```
  j xor_load
xor_done:
  jr $ra
```

9.4.4 Bubble Sort

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
       swapped = false
       for i = 1 to n-1 inclusive do
          if A[i-1] < A[i] then
             swap(A[i-1], A[i])
             swapped = true
          end if
       end for
       n = n - 1
    until not swapped
end procedure
```

```
#
#  Bubble Sort
#
# $a0 points to start of array
# $a1 points to last element in array
         move $t0, $a0
         move $t1, $a1
outter:  move $t4, $0           # swapped this round is false
         lw   $t2, 0($t0)       # get the left compare value
inner:   lw   $t3, 4($t0)       # get the right compare value
         addi $t0, $t0, 4       # increment the left pointer
         ble  $t2, $t3, no_swap # if right>left  swap, else don't
swap:    sw   $t2, 0($t0)       # place left value on right in array
         sw   $t3, -4($t0)      # place right value on left in array
         ori  $t4, $0, 1        # set swapped true
         blt  $t0, $t1, inner   # if not at end then keep going
         subi $t1, $t1, 4       # if at end then shorten the list
         move $t0, $a0          # reset the first element
         b    outter            # start another major loop
no_swap: move $t2, $t3          # no swap, so right element is new left
         blt  $t0, $t1, inner   # if not at end then keep going
         subi $t1, $t1, 4       # if at end then shorten the list
         move $t0, $a0          # reset the first element
         bnez $t4, outter       # start another major loop if swapped
```

9.5 Block Passing

Let us reconsider affine encryption as outlined in Section 5.1

We will be passed a pointer to a string of plaintext, *P, and the length of the string, len. Additionally we need the affine parameters a, b, and n. Five parameters cannot be passed in registers, as we only have four, so we will use a block. Modulus is handled nicely by div in mips so we have no problems there. To be really careful I will use divu (unsigned division).

If an error is detected I will use `break $zero` to halt execution. You could also write your own error handler but that did not seem reasonable given the length of the code already (3 pages). I have tried to exhibit good commenting techniques. They greatly simplify others reading and editing.

```
#############################################
# _affine_encrypt
#
#
# Author: Keith Schubert
# Date  : Nov 4, 2005
# Desc  : Affine encryption of a string
# Method: calculate then modulus.
# BlkPtr: _affine_encrypt_block_pointer
#         var  contents     offset
# Return:
# RetAdd:                       _affine_encrypt_off_ra
# Params: *P   plaintext        _affine_encrypt_off_p
#         len  plaintext.length  _affine_encrypt_off_len
#         *C   ciphertext       _affine_encrypt_off_c
#         a    affine scale     _affine_encrypt_off_a
#         b    affine shift     _affine_encrypt_off_b
#         n    # of code chars  _affine_encrypt_off_n
# Pre   :
# Post  : contents of $t0-$t8 changed, $ra changed
#
#############################################
.data
_affine_encrypt_block_pointer:
_affine_encrypt_base_ra:
    .word 0
_affine_encrypt_base_p:
    .word 0
_affine_encrypt_base_c:
    .word 0
_affine_encrypt_base_len:
    .word 0
_affine_encrypt_base_a:
    .word 0
_affine_encrypt_base_b:
    .word 0
_affine_encrypt_base_n:
    .word 0
_affine_encrypt_block_bottom:

    .set _affine_encrypt_off_ra =
         _affine_encrypt_base_ra - _affine_encrypt_block_pointer
    .set _affine_encrypt_off_p =
         _affine_encrypt_base_p - _affine_encrypt_block_pointer
    .set _affine_encrypt_off_c =
         _affine_encrypt_base_c - _affine_encrypt_block_pointer
    .set _affine_encrypt_off_len =
         _affine_encrypt_base_len - _affine_encrypt_block_pointer
```

```
    .set _affine_encrypt_off_a =
         _affine_encrypt_base_a - _affine_encrypt_block_pointer
    .set _affine_encrypt_off_b =
         _affine_encrypt_base_b - _affine_encrypt_block_pointer
    .set _affine_encrypt_off_n =
         _affine_encrypt_base_n - _affine_encrypt_block_pointer
    .set _affine_encrypt_block_size =
         _affine_encrypt_block_bottom - _affine_encrypt_block_pointer
.text
_affine_encrypt:

#
# Setup
#
# t0 = current char index
# t1 = *p
# t2 = *c
# t3 = len
# t4 = a
# t5 = b
# t6 = n
# t7 = current char
# t8 = effective address
#
la  $t1, _affine_encrypt_block_pointer
lw  $t2, _affine_encrypt_off_c($t1)
lw  $t3, _affine_encrypt_off_len($t1)
bgtz $t3,_affine_encrypt_len_ok
break $zero  #error stop execution
_affine_encrypt_len_ok
lw  $t4, _affine_encrypt_off_a($t1)
lw  $t6, _affine_encrypt_off_n($t1)

#
# Data validity
#
# see if gcd(a,n)=1
mov  $t5, $t4
mov  $t0, $t6
break $zero  # MIPS error
break $zero
# Euclid's alg
_affine_encrypt_Euclid:
divu $t5,$t0
mov  $t5,$t0
mfhi $t0
bgez $t0,_affine_encrypt_Euclid
subi $t5,$t5,1
```

```
beqz $t5,_affine_encrypt_ab_ok
break $zero
_affine_encrypt_ab_ok:

#
# Finish loads
#
lw  $t5, _affine_encrypt_off_b($t1)
lw  $t1, _affine_encrypt_off_p($t1)
mov $t0,$zero

#
# main loop
#
# get char, scale, shift, mod, then store
#
_affine_encrypt_loop:
add  $t8,$t0,$t1
lbu  $t7,0($t8)
mulu $t7,$t4
mflo $t7
add  $t7,$t7,$t5
divu $t7,$t6
mfhi $t7
add  $t8,$t0,$t2
sb   $t7,0($t8)
addi $t0,$t0,1
sle  $t8,$t0,$t3
beqz $t8,_affine_encrypt_loop

#
# Return
#
la $t1,_affine_encrypt_block_pointer
lw $ra,_affine_encrypt_off_ra($t1)
jr $ra
```

9.6 Stack Passing

On some machines you can/must manually allocate your own stack using .bss and .skip. On MIPS the stack is predefined and the OS initializes the stack pointer for you. We are going to define two macros, push and pop. To define a macro we use .macro and .endmacro.

```
.macro push arg1
  addui $sp,$sp,-4  # allocate space
  sw    arg1,0($sp) # place contents
.endmacro

.macro pop arg1
```

```
  lw    arg1,0($sp) # get contents
  addui $sp,$sp,4   # deallocate space
.endmacro
```

Let's consider Euclid's algorithm for finding the GCD of two numbers

1. Let a,b be positive numbers
2. a=b and b=a mod b
3. repeat 2 until b=0
4. gcd=a

iteration	a	b
1	15	12
2	12	3
3	3	0

iteration	a	b
1	49	84
2	84	49
3	49	35
4	35	14
5	14	7
6	7	0

```
###############################################
#
# _euclid_alg_gcd
#
# Author: Keith Schubert
# Date  : Nov 4, 2005
# Desc  : greatest common divisor
# Method: Euclid's Algorithm, recursive
#         var       offset
# Return: gcd       _euclid_alg_gcd_off_gcd
# RetAdd: ra        _euclid_alg_gcd_off_ra
# Params: a         _euclid_alg_gcd_off_a
#         b         _euclid_alg_gcd_off_b
# Pre   :
# Post  : contents of $t0-$t8 changed, $ra changed
#
##############################################
_euclid_alg_gcd:
```

9.6.1 Towers of Hanoi

Implement a recursive function to solve the towers of Hanoi in MIPS.

```
#
# hanoi
#
# Frame: Return address
#        *Answer
#        Answer Size
```

```
#       Number of disks
#       Free
#       Destination
#       Source
.set hanoi_off_ra=0
.set hanoi_off_ans=4
.set hanoi_off_size=8
.set hanoi_off_num=12
.set hanoi_off_free=16
.set hanoi_off_dest=20
.set hanoi_off_source=24
.set hanoi_allocate=-28
.set hanoi_deallocate=28
.set newline="\n"
.set arrow=">"
hanoi:
  lw $t0,hanoi_off_num($t0)
  subi $t0,$t0,1
  blez $t0,done
  #
  # move stack-1 to free
  mov $fp,$sp
  addiu $sp,$sp,hanoi_allocate
  sw $t0,hanoi_off_num($sp)    # num-1
  lw $t0,hanoi_off_ans($fp)    # same string
  sw $t0,hanoi_off_ans($sp)
  lw $t0,hanoi_off_size($fp)   # same size
  sw $t0,hanoi_off_size($sp)
  lw $t0,hanoi_off_free($fp)   # new dest=free
  sw $t0,hanoi_off_dest($sp)
  lw $t0,hanoi_off_dest($fp)   # new free=dest
  sw $t0,hanoi_off_free($sp)
  lw $t0,hanoi_off_source($fp) # source same
  sw $t0,hanoi_off_source($sp)
  la $t0,back1                 # return address
  sw $t0,hanoi_off_ra($sp)
  j hanoi
  back1:
  #don't deallocate yet, we are calling another in a sec
  #
  # store "source>dest\nNull"
  lw $t1,hanoi_off_ans($sp)
  lw $t0,hanoi_off_size($sp)
  add $t1,$t1,$t0
  lw $t2,hanoi_off_source($sp)
  sb $t2,0($t1)
  li $t2,arrow
  sb $t2,1($t1)
```

```
  lw $t2,hanoi_off_dest($sp)
  sb $t2,2($t1)
  li $t2,newline
  sb $t2,3($t1)
  sb $zero,4($t1)
  addi $t0,$t0,4
  sw $t0,hanoi_off_size($sp)
  #
  # move stack-1 to dest
  lw $t0,hanoi_off_dest($fp)     # same dest
  sw $t0,hanoi_off_dest($sp)
  lw $t0,hanoi_off_source($fp)   # new free=source
  sw $t0,hanoi_off_free($sp)
  lw $t0,hanoi_off_free($fp)     # new source=free
  sw $t0,hanoi_off_source($sp)
  la $t0,back2                   # return address
  sw $t0,hanoi_off_ra($sp)
  j hanoi
  back2:
  addiu $sp,$sp,hanoi_deallocate
  lw $ra,hanoi_off_ra($sp)
  jr $ra

done:
  # store "source>dest\nNull"
  lw $t1,hanoi_off_ans($sp)
  lw $t0,hanoi_off_size($sp)
  add $t1,$t1,$t0
  lw $t2,hanoi_off_source($sp)
  sb $t2,0($t1)
  li $t2,arrow
  sb $t2,1($t1)
  lw $t2,hanoi_off_dest($sp)
  sb $t2,2($t1)
  li $t2,newline
  sb $t2,3($t1)
  sb $zero,4($t1)
  addi $t0,$t0,4
  sw $t0,hanoi_off_size($sp)
  lw $ra,hanoi_off_ra($sp)
  jr $ra
```

9.6.2 Tracing Code

The code that follows, implements the algorithm

$$n_{k+1} = \begin{cases} 3n_k + 1 & \text{if } n_k \text{ is odd} \\ \frac{n_k}{2} & \text{if } n_k \text{ is even} \end{cases}$$

in MIPS. Trace the code by showing how the register values change. What is the value that is returned? Note: this code is a somewhat famous problem in number theory. The problem is to prove that starting at any number, the algorithm will bring you to 1.

```
!   code                    $t0   |  $a0   |  $v0
!                                 |   3    |
!-------------------------------------------
secret:                    !      |        |
  bgtz $a0, ok             !      |        |
  break $zero              !      |        |
ok:                        !      |        |
  addi $v0,$zero,1         !      |        |
  subi $t0,$a0,1           !      |        |
  beqz $t0, end            !      |        |
loop:                      !      |        |
  addi $v0,$v0,1           !      |        |
  andi $t0,$a0,1           !      |        |
  beqz $t0, even           !      |        |
  sll  $t0,$a0,1           !      |        |
  add  $a0,$a0,$t0         !      |        |
  addi $a0,$a0,1           !      |        |
  b loop                   !      |        |
even:                      !      |        |
  sra  $a0,$a0,1           !      |        |
  subi $t0,$a0,1           !      |        |
  bgtz $t0, loop           !      |        |
end:
```

I will show changes on successive loops by placing a comma and then the new value

```
#   code                    $t0             |  $a0             |  $v0
#                                           |   3              |
#-----------------------------------------------------------------------------
secret: bgtz $a0, ok      #                | 3                |
        break $zero       #                |                  |
ok:     addi $v0,$zero,1  #                | 3                | 1
        subi $t0,$a0,1    # 2              | 3                | 1
        beqz $t0, end     # 2              | 3                | 1
loop:   addi $v0,$v0,1    # 2,6,4 ,10,7,3,1| 3,10,5 ,16,8,4,2| 2,3,4,5,6,7,8
        andi $t0,$a0,1    # 1,0,1 ,0 ,0,0,0| 3,10,5 ,16,8,4,2| 2,3,4,5,6,7,8
        beqz $t0, even    # 1,0,1 ,0 ,0,0,0| 3,10,5 ,16,8,4,2| 2,3,4,5,6,7,8
        sll  $t0,$a0,1    # 6   ,10        | 3   ,5           | 2  ,4
        add  $a0,$a0,$t0  # 6   ,10        | 9   ,15          | 2  ,4
        addi $a0,$a0,1    # 6   ,10        | 10  ,16          | 2  ,4
        b loop            # 6   ,10        | 10  ,16          | 2  ,4
even:   sra  $a0,$a0,1    #   0   ,0 ,0,0,0|   5    ,8 ,4,2,1|   3  ,5,6,7,8
        subi $t0,$a0,1    #   4   ,7 ,3,1,0|   5    ,8 ,4,2,1|   3  ,5,6,7,8
        bgtz $t0, loop    #   4   ,7 ,3,1,0|   5    ,8 ,4,2,1|   3  ,5,6,7,8
end:
```

Returns 8.

Chapter 10

Data Transfer

10.1 I/O

Transmission of data from one device to another is the essence of I/O. Usually, I/O is accomplished by defining registers to hold the information necessary to transmit the data. The registers that handle the transmission are called the I/O port. At least three registers are used, one for the data, one for the control, and one for the Status.

Data the codes to be transmitted. These can be traditional codes, such as ASCII, or even an address of data being requested.

Control the commands specifying what is to be done.

Status a series of bits specifying what is going on with the bus and the current transaction.

Accessing the registers (reading from or writing to) can be accomplished in two ways.

Memory Mapped the registers of the I/O port, have addresses in regular memory, and thus can be treated as a regular memory location for access purposes.

Isolated the registers are in a separate (isolated) memory address scheme, and thus the memory must be access through special commands.

10.2 Busses

Internal vs. External (relative to cpu)
Master/Slave (initiator/target)

(Transaction) Master the initiator of a transaction.

(Transaction) Slave the target of a transaction.

Bus Master any device that can be a (transaction) master.

Burst Mode Transaction transaction which transmits several values.

Bus Transaction data transfer on an external bus.

Synchronous Bus Lines

Line/Signal	Num	Owner
Clock	1	Bus
Start	1	Master
Address	k	Master
$R/\overline{W}$	1	Master
Data	n	Master/Slave
Done	1	Slave

Arbitration is usually overlapped

10.2.1 Synchronous/Asynchronous Transfer

Busses have to have a way to specify when to transfer and if data has been received. The two basic schemes for transfer is synchronous and asynchronous.

Synchronous transfers uses a clock signal to coordinate communication, and is thus very fast. For a data request, we only need to spend one bus cycle to sent the request, the access time to find the data, and one bus cycle to send the answer. The time to transmit the data is thus

$$T_{transmit} = \frac{2}{f_{bus}} + T_a,$$

were T_a is the time to access the data, and f_{bus} is the bus clock rate[1]. The faster the clock the less time to transmit the data. The bandwidth of the bus in terms of transactions is

$$BW_{transaction} = \frac{W_{bus}}{T_{transmit}},$$

where W_{bus} is the width of the bus[2]. Frequently however, buses are measured not by an actual transaction but by what a one way message would be

$$\begin{aligned} BW &= \frac{W_{bus}}{T_{bus}} \\ &= W_{bus} f_{bus}. \end{aligned}$$

Let's consider a few examples. Note that we will be reporting bandwidth in megabytes per second (MB/s). A byte is 8 bits, and a megabyte is 2^{20} bytes. Bus frequencies (sometimes called speeds) are reported in megaHertz (MHz), but here mega is in base 10 not base 2, so it is 10^6 Hertz. Recall a Hertz is a reciprocal second. Sometimes this distinction is ignored to simplify calculations.

Example 16 (PCI) *A basic PCI bus is 32 bits wide (4 bytes) and runs at 33.3 MHz. Thus the bandwidth is*

$$\begin{aligned} BW &= W_{bus} f_{bus} && (10.1) \\ &= \left(4[B]\frac{1[MB]}{2^{20}[B]}\right)\left(33.3[MHz]\frac{10^6[Hz]}{1[MHz]}\right) && (10.2) \\ &= \left(\frac{1}{2^{18}[MB]}\right)(3.33 \times 10^7[Hz]) && (10.3) \\ &= \left(\frac{1}{2^{18}[MB]}\right)(3.33 \times 10^7[Hz]) && (10.4) \\ &\approx 127[MB/s] && (10.5) \end{aligned}$$

[1] A one way transmission must finish in this time.

[2] How much data can be sent simultaneously, i.e. the number of wires measured in bits or bytes. A bus that has 32 data wires is 32 bits wide or 4 bytes wide.

Clock signals take time to transfer down the wire and thus is subject to clock skew. To understand clock skew, consider a simple example of two clocks 3 kilometers apart. The clocks are synchronized by a beam of light, which travels at 3×10^5 km/s, and thus it takes 10μs for the synchronization pulse to arrive from the master clock. If the clocks were only synchronized once per second the fraction of the synchronization time used to transmit the pulse would be $\frac{10\mu s}{1s} = .001\%$, which is basically insignificant. What if we wanted to synchronize the clocks every tenth of a milisecond (.1ms)? The fraction of time to transfer now is $\frac{10\mu s}{.1ms} = 10\%$, which is very significant. When the clock pulse arrives it is off by 10%! That is called clock skew, when the transmission time of the clock pulse takes a significant portion of the clock frequency. Clock skew is effected by the distance (d) and the clock rate (f). If the clock skew is some fraction (s) and we assume that the clock signal is carried at the speed of light (c) then the relation between the variables is

$$\frac{d}{c} = \frac{s}{f}$$

Assuming we want the skew to be less than a third ($s = .33\ldots$), the distance is measured in meters and the bus clock will be measured in megahertz, then

$$df = 100.$$

In other words a 100MHz bus (f=100) can only be 1 meter long (d=1) to keep clock skew under 33.3%! Given that bus speeds of 400MHz are very reasonable, this would limit bus length to about 9in. Thus we see that clock skew limits bus length, and thus synchronous buses are fast but short.

Asynchronous transfers get around the problem of clock skew by doing a procedure called handshaking. Basically two units that want to talk send messages back and forth letting each other know what is going on. A basic handshaking protocol between a sender (S) and a receiver (R) to request data from R is

1. S to R: Here is the address of the data I want.
2. R to S: I got your request and will look it up.
3. S: Drop request when recieve
4. R: looking up data.
5. R to S: Here is your data.
6. S to R: I got it.
7. S: Wait till see data signal drop then drop acknowledgement.

Call the time for the signal to travel from sender to receiver or vice versa T_h (for handshake time), and the time to get the data as T_a(for access time). If we are clever we can overlap items 2,3 with item 4, so that we will only take the longer of $2T_h$ or T_a rather than $2T_h + T_a$. The total time for one transfer is thus

$$T_{transfer} = 4T_h + \max(2T_h, T_a).$$

The bandwidth of the bus is the rate at which data can be sent, and thus

$$BW = \frac{W_{bus}}{T_{transfer}},$$

where W_{bus} is the width of the bus.

10.2.2 Polling and Interrupts

There are two basic ways to handle bus communication with the CPU: polling, interrupts. Direct Memory Access (DMA) is a special case of interrupts.

Polling - CPU Controlled Data Transfer

$$\begin{aligned}
\text{Fraction of CPU Time} &= \frac{\text{Cycles Per Second used on Polls}}{\text{Clock Frequency}} \\
&= \frac{\frac{\text{Polls}}{\text{Sec}}\frac{\text{Cycles}}{\text{Poll}}}{\text{Clock Frequency}} \\
&= \frac{\frac{\text{Data Rate}}{\text{Poll Size}}\frac{\text{Cycles}}{\text{Poll}}}{\text{Clock Frequency}}
\end{aligned}$$

Interrupt Driven - CPU Controlled Data Transfer

$$\begin{aligned}
\text{Fraction of CPU Time} &= \frac{\text{Cycles Per Second used on Interrupts}}{\text{Clock Frequency}} \\
&= \frac{\frac{\text{Interrupts}}{\text{Sec}}\frac{\text{Cycles}}{\text{Interrupts}}}{\text{Clock Frequency}} \\
&= \frac{\frac{\text{Data Rate}}{\text{Packet Size}}\frac{\text{Cycles}}{\text{Interrupt}}}{\text{Clock Frequency}}
\end{aligned}$$

Interrupt Driven - Direct Memory Access (DMA)

$$\begin{aligned}
T_{\text{Transfer}} &= \frac{\text{Size Transfer}}{\text{Speed Transfer}} \\
&= \frac{\text{Data Size}}{\text{Data Rate}} \\
\text{Cycles to Handle} &= C_h \\
&= \frac{\text{Cycles to Start} + \text{Cycles to Complete} + f_e \times \text{Cycles to handle errors}}{1 - f_e} \\
\text{Fraction of CPU Time} &= \frac{\text{Cycles Per Second used to handle DMA}}{\text{Clock Frequency}} \\
&= \frac{\frac{C_h}{T_{\text{Transfer}}}}{\text{Clock Frequency}} \\
&= \frac{C_h}{T_{\text{Transfer}}\text{Clock Frequency}}
\end{aligned}$$

Example

You are given a 32-bit **asynchronous** bus with a handshaking time of 15 ns. Your computer has the following equipment attached:

Hard Drive	RAM
Total Latency: 7.2 ms	Access Time: 40ns
Disk Transfer Rate: 10MB/s	No Burst Mode
Number of Disks: 4	

Showing all work calculate the following:

1. the band width of the bus,
2. the percent of the bus utilized by continuous paging of a virtual memory system with 32KB pages,

3. the number of cache to RAM transfers that can occur if: The bus is continuously paging and 10% of the bandwidth must be left for other transactions (Hint: calculate the available bandwidth for the RAM transactions and use the size of the transactions).

The bandwidth of the bus is:

$$\begin{aligned}
\text{BW} &= \frac{\text{Data Transfered}}{\text{Time to Transfer}} \\
&= \frac{\text{Bus Width}}{4T_{Hand} + \max 2T_{Hand}, T_{RAM}} \\
&= \frac{4B}{4(15ns) + \max 2(15ns), 40ns} \\
&= \frac{4B}{100ns} \\
&= 40MB/s
\end{aligned}$$

The effective transfer rate of the pages from the disks is:

$$\begin{aligned}
\text{Rate}_{\text{Disk}} &= \frac{\text{Data Transfered}}{\text{Time to Transfer}} \\
&= \frac{\text{Data Transfered}}{\text{Total Latency} + \frac{\text{Data Transfered}}{\text{Combined Disk Transfer Rate}}} \\
&= \frac{32KB}{7.2ms + \frac{32KB}{4\times 10MB/s}} \\
&= \frac{32KB}{7.2ms + .8ms} \\
&= 4MB/s
\end{aligned}$$

Thus the bandwidth available to RAM is $40 - 4 - 4 = 32$ MB/s. Since each transfer is 4 B, the transfers per second is 8×10^6 transfers/sec or 1 cache miss every 125 ns.

Chapter 11

Memory and Cache

11.1 Memory

2D

2.5D

A synchronous memory bus for a system with 2^k addresses of n bit words would require at least:

- k address lines
- n data lines
- 4+ control lines

or a total of $k + n + 4$ parallel lines. See Section 10.2

Memory is usually byte-addressable, but I don't just load it one byte at a time. In a typical 2D or 2.5D RAM configuration though, if I had all of memory in one large module/array, I would only be able to access one byte at a time. To allow access to more than one byte at a time, memory is interleaved: the first byte is stored in the first location of the first module/array, the second byte in the first location of the second module/array, and so on. When all the module/arrays have their first location addressed, the second locations are specified, see Table 11.1.

Module Address	Module 1	Module 2	$\ldots$	Module N
0	0	1	$\ldots$	$N-1$
1	N	$N+1$	$\ldots$	$2N-1$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
2^k-1	$(2^k-1)N$	$(2^k-1)N+1$	$\ldots$	2^kN-1

Table 11.1: Mapping Memory Module's Addresses to the Computer's Memory Addresses

A number of potential problems can arise. Consider the four byte integer, $0x12345678$, stored starting in address 2 on a machine with four modules. In the easiest and fastest way to implement the hardware, the first byte of the returned number comes from the first module, the second byte from the second module and so on. By examining Table 11.2 you will notice that this means the value sent back is $0x56781234$ or even $0xABCD1234$ depending on how the addresses are selected!

To prevent such problems, systems adopt standards of how memory must be stored. The simplest method is justified, in which the first byte of any new memory item must start in the first module. Justified can obviously lead to some inefficiencies in memory utilization. A more sophisticated method is aligned, in which

First Byte Address	Module 1	Module 2	Module 3	Module N
0	$0xAB$	$0xCD$	$0x12$	$0x34$
1	$0x56$	$0x78$	$0x00$	$0x00$

Table 11.2: Memory Contents of Non-Aligned Integer

a new memory item must start at an address that is divisible by the number of bytes in the memory item (e.g.: a 4 byte integer can start at any address that can be expressed as $4i$ for i a non-negative integer).

11.1.1 Endian

Big (LR) and little (RL) endian

Consistent (same for bits)

Sparc is inconsistent big-endian.

Endian	Consistent				Inconsistent			
	0	1	...	n	0	1	...	n
Big	0...7	0...7	...	0...7	7...0	7...0	...	7...0
	n	...	1	0	n	...	1	0
Little	7...0	7...0	...	7...0	0...7	0...7	...	0...7

11.2 Cache Design

In general DRAM has a cycle-time of about 50ns to 80ns, and SRAM has a cycle-time of 5ns to 20ns. Main memory is almost exclusively DRAM due to size and cost, so access will be slow. Strategies must be used to speed up access to main memory. Several common techniques are:

Wide Memory memory that passes multiple words at a time.

Interleaving memory that has successive addresses stored in different components that can be accessed simultaneously.

Prefetching buffer that fetches most likely instructions (or sometimes data) when memory is idle.

Cache data and instructions that have been accessed are stored in fast memory (SRAM) that is close to the CPU often as well as in main memory.

Usually, a variety of techniques are used, and often multiple levels of cache (l1, l2, and even l3).

Cache can be:

fully associative any main memory location can be stored in any cache location.

2^k-way set associative each main memory location must be stored in one of n prescribed cache locations. Usually, $16 \geq k \geq 1$.

direct mapping each main memory location must be stored in a particular cache location. This is the same as 1-way set associative.

Let's introduce some formalisms. Let 2^k be the associativity of the cache, 2^l be the size of a cache location (block size, usually less than 16 words), 2^m be the number of cache locations, and 2^n be the size of main memory.

Then

$$
\begin{aligned}
\text{number of sets} &= m-k \\
\text{size of the cache} &= 2^{(l\times m)} \\
\#\text{ address bits inferred by location} &= m-k+l \\
\#\text{ tag address bits} &= n-(m-k+l)
\end{aligned}
$$

n-(m-k+l)	m-k	l
tag address bits	set address bits	offset in block

Example: Cache for Toy Stack

Design a 4 way associative, 8 byte cache for a 64 byte system (i.e.: the Toy Stack). Show an example of how your system would do a cache lookup (ie: through all the steps for a lookup, you may pick memory and cache to have any values you want)

The numbers of our design are as follows.

- 64 bytes means 6 bit addresses
- 8 byte cache means 3 bit addresses
- 4 way associative means the high two bits of each cache address do not need to match the corresponding bits in main memory, but the least bit does.
- 5 bits of address from main memory need to be identified for each cache location, with the valid bit, this makes 6 tag bits for each cache location.
- the least significant bit of the main memory address to be checked for is used as a lookup on the cache to provided the 4 specific locations in cache that must be checked
- the 5 address tag bits of each of the 4 cache locations is compared with the high 5 bits of the main memory address.
- if any of them match and the corresponding valid bit is set then we have a cache hit and the data is sent
- if there is no match or the match is not valid main memory is accessed.

lookup

Let the address to be checked for be 010111, and let the cache be

Tag Bits						Address			Contents
High Address					Valid Bit				
0	0	1	1	0	1	0	0	0	11011101
0	1	0	1	0	1	0	0	1	11010110
0	0	0	0	0	0	0	1	0	00011100
1	0	0	0	0	0	0	1	1	10010100
1	1	0	1	1	1	1	0	0	11101101
1	0	1	0	1	0	1	0	1	11011110
1	0	0	0	0	1	1	1	0	11111111
0	1	0	1	1	1	1	1	1	11010000

First, the low bit (a 1) of the address tells us to look at the 4 odd addresses in cache:

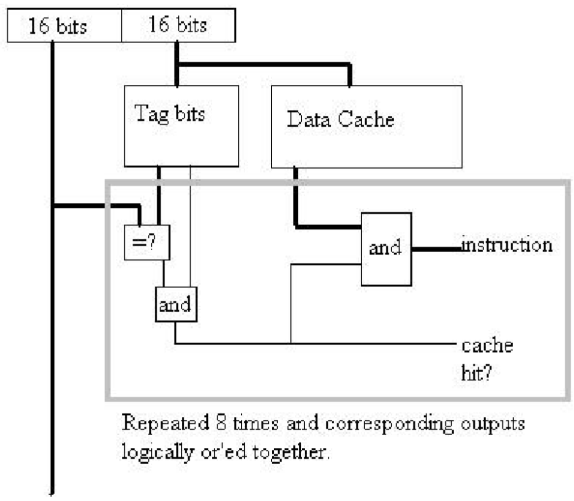


Figure 11.1: 8-Way Set Associative Cache

Tag Bits						Address			Contents
High Address					Valid Bit				
0	1	0	1	0	1	0	0	1	11010110
1	0	0	0	0	0	0	1	1	10010100
1	0	1	0	1	0	1	0	1	11011110
0	1	0	1	1	1	1	1	1	11010000

The 5 address tag bits are checked against the high five bits of the address (01011):

Tag Bits						Address			Contents
High Address					Valid Bit				
0	1	0	1	1	1	1	1	1	11010000

The address matches and the valid bit is set so 11010000 is sent as the contents.

Example: 8-way set associative

Consider a machine with 32 bit addressing (up to 4GB of RAM) and 512k (2^{19}) of data cache with 1 byte blocks. To define the 8-way set association, it will be required that main memory addresses must have the same last 16 bits (19-3=16) as a cache location to be stored in that cache location. Every cache location has 17 extra bits, 16 for addressing, and one for validity. Eight location in cache must be checked for each main memory access (it is 8-way for a reason). The main memory address to be checked is split into the upper and lower 16 bits. The lower 16 bits are used to identify the eight cache locations, whose 16 address tag bits are then compared to the 16 high bits of the main memory address, see Figure 11.1. This generates eight signals (true if match was found) that are then logically and'ed together with the corresponding 8 validity bits (might have the same address but might not be current). If any generates a hit (is true) then its contents are sent as the data.

Replacement policies

LRU Least Recently Used

FIFO First-in First-out

LFU Least Frequently Used

Random Random

11.2.1 Neat Little LRU Algorithm

Let the number of cache slots (locations) be 2^k, then we create a matrix of bits that is $2^k \times 2^k$ (so we can associate the cache address with both a row and column). Initially they are all cleared. When a cache slot, say address p, is accessed:

1. 1's are placed in every bit of the matrix row p,
2. 0's are placed in every bit of the matrix column p.

Note that the second step will delete one of the 1's you placed in the first step.

The the address that was least recently used corresponds to the number of the row that has a sum of zero. Equivalently, the address that was least recently used corresponds to the number of the column with the largest sum.

Example: Fully Associative Cache With 4 Slots

For simplicity we will assume main memory has 256 (2^8) bytes, and the data length is 1 byte. The cache starts empty.

NLLRU				Tag Bits			Data
0	1	2	3	V	D	Address	
0	0	0	0	0	0	0x00	0x00
0	0	0	0	0	0	0x00	0x00
0	0	0	0	0	0	0x00	0x00
0	0	0	0	0	0	0x00	0x00

Address 0x1A, which contains 0x49, is accessed.

NLLRU				Tag Bits			Data
0	1	2	3	V	D	Address	
0	1	1	1	1	0	0x1A	0x49
0	0	0	0	0	0	0x00	0x00
0	0	0	0	0	0	0x00	0x00
0	0	0	0	0	0	0x00	0x00

Address 0x05, which contains 0x11, is accessed.

NLLRU				Tag Bits			Data
0	1	2	3	V	D	Address	
0	0	1	1	1	0	0x1A	0x49
1	0	1	1	1	0	0x05	0x11
0	0	0	0	0	0	0x00	0x00
0	0	0	0	0	0	0x00	0x00

Address 0x25, which contains 0xFF, is accessed.

NLLRU				Tag Bits			Data
0	1	2	3	V	D	Address	
0	0	0	1	1	0	0x1A	0x49
1	0	0	1	1	0	0x05	0x11
1	1	0	1	0	0	0x25	0xFF
0	0	0	0	0	0	0x00	0x00

The value 0x33 is stored to address 0x05.

NLLRU				Tag Bits			Data
0	1	2	3	V	D	Address	
0	0	0	1	1	0	0x1A	0x49
1	0	1	1	1	1	0x05	0x33
1	0	0	1	0	0	0x25	0xFF
0	0	0	0	0	0	0x00	0x00

The value 0xF5 is stored to address 0x06.

NLLRU				Tag Bits			Data
0	1	2	3	V	D	Address	
0	0	0	0	1	0	0x1A	0x49
1	0	1	0	1	1	0x05	0x33
1	0	0	0	0	0	0x25	0xFF
1	1	1	0	1	1	0x06	0xF5

The value 0x07 is stored to address 0x07.

NLLRU				Tag Bits			Data
0	1	2	3	V	D	Address	
0	1	1	1	1	1	0x07	0x07
0	0	1	0	1	1	0x05	0x33
0	0	0	0	0	0	0x25	0xFF
0	1	1	0	1	1	0x06	0xF5

11.2.2 Implementing LRU Algorithm

NLLRU is a nice algorithm to learn off, but it is not a good one to build. First off it requires over twice as many bits as is needed. Second, it can become inconsistent if a bit flip occurs. To understand these problems notice the LRU square is skew symmetric:

1. The main diagonal is always zero.
2. The lower triangular elements (lower left triangle of the LRU square) are the negated transpose (each bit is the logical not of the bit on the opposite side of the main diagonal) of the upper triangular elements (upper right triangle of the LRU square.

11.2.3 Cache Performance

We will be concerned with some basic numbers

Hit Ratio (HR) The number of cache hits over the number of lookups.

Miss Ratio (MR) The number of cache misses over the number of lookups.

Effective Access Time (EAT or T_{eff}) The average time spent in a memory access.

First let us consider the hit and miss ratios. For a series of lookups, the number of hits was "Hit" and the number of misses was "$Miss$", thus $Hit + Miss = lookups$. Given this,

$$\begin{aligned} HR &= \frac{Hit}{Hit + Miss} \\ MR &= \frac{Miss}{Hit + Miss} \\ 1 &= HR + MR \end{aligned}$$

thus,

$$\begin{aligned} T_{eff} &= \frac{Hit \times T_{Hit} + Miss \times T_{Miss}}{Hit + Miss} \\ &= HR \times T_{Hit} + MR \times T_{Miss}. \end{aligned}$$

Usually, the miss time is the access time (T_{Hit}), plus a miss penalty (say $T_{Penalty}$).

$$\begin{aligned} T_{Miss} &= T_{Hit} + T_{Penalty} \\ T_{eff} &= HR \times T_{Hit} + MR \times T_{Miss} \\ &= HR \times T_{Hit} + MR \times (T_{Hit} + T_{Penalty}) \\ &= (HR + MR) \times T_{Hit} + MR \times T_{Penalty} \\ &= T_{Hit} + MR \times T_{Penalty} \end{aligned}$$

Example

Use the following chart to show the state of a 4 location, 2-Way associative cache, that uses LRU. If a location has a number printed in it, the address is valid, if no number appears the contents are invalid. For simplicity the computer only has 16 locations in memory. If the cache takes 5ns to access and RAM takes 60ns, what is the effective access time given the sequence?

Time	0	1	2	3	4	5	6	7	8	9	10
Lookup Address	-	2	5	6	B	5	2	2	B	C	5
Cache location 00	A										
Cache location 01	B										
Cache location 10											
Cache location 11											

Time	0	1	2	3	4	5	6	7	8	9	10
Lookup Address	-	2	5	6	B	5	2	2	B	C	5
Cache location 00	A	A	A	6	6	6	6	6	6	C	C
Cache location 01	B	B	B	B	B	B	B	B	B	B	B
Cache location 10		2	2	2	2	2	2	2	2	2	2
Cache location 11			5	5	5	5	5	5	5	5	5

MR=.4

$$\begin{aligned} T_{eff} &= T_{cache} + MR(T_{RAM}) \\ &= 5ns + .4(60ns) \\ &= 29ns \end{aligned}$$

11.3 Virtual Memory

A 32-bit virtual memory system has a 64KB page size, and 1 GB of RAM. How large is the physical page number in bits? Assuming that the each entry in the table is word aligned, how large is the lookup table in bytes?

64KB $= 2^{16}$

1GB = 2^{30}

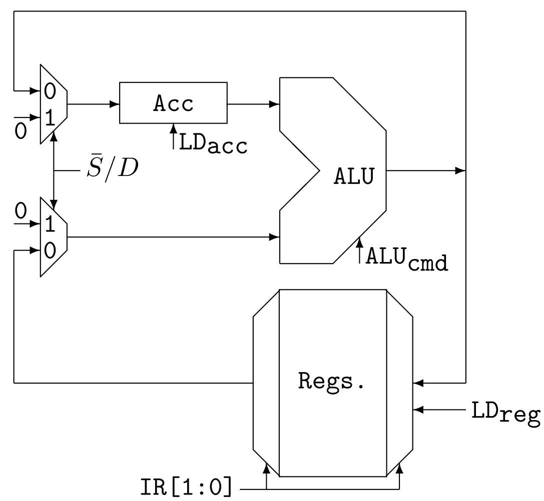
So the physical page number takes 30-16=14 bits or almost 2B to store in the table. We also need to add memory protection, ownership, validity, location, etc. I will assume that I can fit all this in 4B.

The table size is $2^{(32-16)} \times 4B = 2^{18}B = 256KB$

Chapter 12

CPU Control

12.1 Tiny Accumulator



The tiny accumulator has four commands

Mach. Code	Assem. Lang.	Description.
00MN	STC MN	Store Acc to location MN and clear Acc
01MN	ADD MN	Add Acc and location MN placing result in Acc
10MN	SUB MN	Sub location MN from Acc, placing result in Acc
11MN	BRL MN	if Acc is negative, Branch to $nPC + MN\bar{N}$

STC MN The store and clear command not only allows storage, but due to the clear, allows a load if it is followed by adding the desired value to load. The instruction is implemented as follows. The signal $\bar{S}/D$ is set to 1, which puts a zero both on the accumulator and the second input of the ALU. The ALUop is set to add, which thus does ACC plus zero, and so the value of the ACC is placed on the answer line. Both the ACC and the register file is told to read, which results in the ACC loading zero, and register M loading the value that had been in the ACC.

ADD MN This instruction makes it easy to load the ACC as mentioned in STC MN, as well as providing an arithmetic command. The instruction is implemented as follows. The signal $\bar{S}/D$ is set to 0, which allows the selected register to go to the second input of the ALU and allows the result of the ALU to go to the ACC input. the ALUop is set to add, and finally the ACC is told to load, so the result becomes stored.

SUB MN This instruction is very similar to ADD. The instruction is implemented as follows. The signal $\bar{S}/D$ is set to 0, which allows the selected register to go to the second input of the ALU and allows the result of the ALU to go to the ACC input. the ALUop is set to sub, and finally the ACC is told to load, so the result becomes stored.

BRL MN This instruction allows loops and conditional executions to be handled. The offset is taken to be a three bit, two's compliment number, of which the first two are MN and the last bit is the flip of N. While this may sound strange it makes the displacements to be

MN	$MN\bar{N}$	displacement
11	110	-2
10	101	-3
01	010	2
00	001	1

The negative numbers allow loops which include one or two instructions besides the branch, and the positive numbers allow for conditional statements of one or two instructions. Note the negative numbers are larger in magnitude by one to include the branch statement.

This gives us a full architecture that can be programmed, but is small enough to be built by hand.

12.2 GST ISA

Gomez-Schubert-Tafas Instruction Set Architecture.

My thought is to implement 1k-word of memory for each processor, and to do memory mapped IO so we don't need special commands. The word size is 16 bits and this is the smallest addressable size, again for simplicity. The "network" port should have a buffer of, say, 16 words. Initially there will not be a cache because since this will be a SOC there is no access time advantage.

The ISA is load-store. I have broken the 16 bit instruction into 4 nibbles for different purposes as seen below. I have tried to pair commands by opcode to make for easier control. I left two unused in case there is anything you want to add.

We only use register, immediate, and indexed addressing, to keep things simple and still provide flexibility. These three modes allow us to do anything.

I am only considering two's complement numbers, so no unsigned numbers. While this is a limitation for real computers, I don't think it will matter for this test architecture.

12.2.1 R Type Commands

FEDC	BA98	7654	3210
Opcode	RD	RS1	RS2

or

FEDC	BA98	7654	3210
Opcode	RD	RS1	Imm1

12.2.2 I Type commands

FEDC	BA98	76543210
Opcode	RD	Imm2

12.2.3 B Type commands

FEDC	BA9876543210
Opcode	Imm3

12.2.4 Commands

Opcode	Assembly	Comments
0000	load RD(RS1+RS2)	$RD \leftarrow M[RS1 + RS2]$
0001	store RD(RS1+RS2)	$RD \rightarrow M[RS1 + RS2]$
0010	ldi RD,Imm2	$RD[F:8] \leftarrow Imm2$
0011		
0100	add RD,RS1,RS2	$RD \leftarrow RS1 + RS2$
0101	sub RD,RS1,RS2	$RD \leftarrow RS1 - RS2$
0110		
0111		
1000	sll RD,RS1,Imm1	$RD \leftarrow RS1 << Imm$
1001	sra RD,RS1,Imm1	$RD \leftarrow RS1 >> Imm$
1010	nand RD,RS1,RS2	$RD \leftarrow (RS1 \cdot RS2)'$
1011	nor RD,RS1,RS2	$RD \leftarrow (RS1 + RS2)'$
1100	brlt RD,RS1,Imm1	$(RD < RS1) \Rightarrow (PC \leftarrow nPC + \{Imm1[3:0], \not{I}mm1[0]\})$
1101	brle RD,RS1,Imm1	$(RD \leq RS1) \Rightarrow (PC \leftarrow nPC + \{Imm1[3:0], \not{I}mm1[0]\})$
1110	br Imm3	$PC \leftarrow PC + Imm3$
1111	j RD	$PC \leftarrow PC + RD$

Note: SE is sign extend.

12.2.5 Registers

0	R0	Zero	8	L0	Local Register 0
1	R1	General Purpose Register 1	9	L1	Local Register 1
2	R2	General Purpose Register 2	10	L2	Local Register 2
3	R3	General Purpose Register 3	11	L3	Local Register 3
4	R4	General Purpose Register 4	12	L4	Local Register 4
5	R5	General Purpose Register 5	13	L5	Local Register 5
6	R6	General Purpose Register 6	14	SP	Stack Pointer
7	R7	General Purpose Register 7	15	RA	Return Address

Part III

Performance

Chapter 13

Performance

13.1 Cost

$$\text{Cost of IC} = \frac{\text{Cost of die} + \text{Cost of Testing} + \text{Cost of Packaging}}{\text{Final Yield}}$$

$$\text{Cost of Die} = \frac{\text{Cost of Wafer}}{\text{Dies per Wafer} \times \text{Die Yield}}$$

$$\text{Die Yield} = \frac{WaferYield}{\left(1 + \frac{\text{Defects per Area} \times \text{Die Area}}{\alpha}\right)^{\alpha}}$$

$$\begin{aligned}
\text{List Price} &= \frac{4}{3}\text{Average Selling Price} \\
&= \frac{4}{3}\frac{4}{3}\text{Production Cost} \\
&= \frac{4}{3}\frac{4}{3}\frac{6}{5}\text{Component Cost} \\
&= \frac{32}{15}\text{Component Cost} \\
&\approx 2\text{Component Cost}
\end{aligned}$$

13.2 Power, Energy, and Heat

These are probably the most misused terms in computers (and many other fields as well). They are not synonyms and should not be used as such.

Work Electrical work is electrical force applied on a charge over a distance. Usually Electrical force is calculated by the the charge times the electrical field. For computers a computation involves moving charges from one place to another by applying a voltage, i.e.: electrical work. The work done does not change with the time it takes to do the computation. Think of it as this is what you want to do.

Energy The ability to do work. You can also consider this the cost of doing work. In a computer Energy use is primarily due to dynamic operations (switching transistors), so

$$E_d = \frac{1}{2}CV^2$$

, where E_d is the dynamic energy, C is the capacitive load of the computer (consider it constant for a computer design), and V is the voltage of the computer. Energy for laptops are stored in batteries, and since this is a fixed source energy is a major issue to laptops (i.e. we care about the work done which is proportional to the computations we do).

Power The rate at which energy is used (and thus work done). Total power is the sum of dynamic power and static power. We are primarily concerned with dynamic power (again from switching transistors), so assuming the capacitance does not change,

$$P_d = \frac{d}{dt}E_d \tag{13.1}$$

$$= CV(t)\frac{dV(t)}{dt}, \tag{13.2}$$

where P_d is dynamic power, C is capacitive load, and V is voltage. A standard assumption is that the voltage is an ideal square wave with a duty cycle of $\frac{1}{2}$ with a switching frequency of f_s, which is proportional to the clock frequency of the processor, thus

$$P_d = \frac{1}{2}CV^2 f_s. \tag{13.3}$$

Static power loss is caused primarily from leakage current in the transistors and thus is constant even for inactive circuits (the computer must be on of course though). Static power, P_s is given by $P_s = i_c \cdot V$, where i_c is the static current (leakage current in one transistor time the number of transistors), and V is still voltage. Static power accounts for more than 25% in current computers. Computers that have a continuous power source are more concerned with power, as power also tells us the rate of heat production. We are at the limits of air cooling, so this is a major issue.

13.3 Dependability

MTTF mean time to fail

MTTR mean time to repair (detect + fix)

MTBF mean time between failures

$MTBF = MTTF + MTTR$

$$MTTF(AorB) = \frac{1}{\frac{1}{MTTF(A)} + \frac{1}{MTTF(B)}} \tag{13.4}$$

$$= \frac{MTTF(A)MTTF(B)}{MTTF(A) + MTTF(B)} \tag{13.5}$$

For an identical device this becomes:

$$MTTF(2) = \frac{MTTF}{2} \tag{13.6}$$

13.4 Performance

Response Time (aka execution time) the time between the start and completion of a task.

Throughput The number of task completed in a period of time.

There are four tasks (a, b, c, and d) which are composed of four subparts (1, 2, 3, 4 for each of a, b, c, and d) that are independent (i.e. you can do a1 and a2 simultaneously). You are to run them on a four processor machine. Ignoring memory and overhead, we can schedule the processes as:

Time

Processor

	1	2	3	4
1	a1	a2	a3	a4
2	b1	b2	b3	b4
3	c1	c2	c3	c4
4	d1	d2	d3	d4

or

Time

Processor

	1	2	3	4
1	a1	b1	c1	d1
2	a2	b2	c2	d2
3	a3	b3	c3	d3
4	a4	b4	c4	d4

13.5 Time

Time can be different things. There is time that we exist in, sometimes called "wall time" due to measurements by wall clocks. There is the CPU time of the program, but even here do we mean the total time from start to finish, or just the time spent on the program without counting system functions or other programs (execution time). We will in general speak of only the execution time or CPU Time (CPUT, T_{CPU}) of the program, for simplicity.

The longer a process takes to run the worse the performance, this should be obvious as who wants a slower machine. We could also say, the less time a process takes the better the performance. Execution time and performance are thus inversely related:

$$\text{Perf} = \frac{1}{\text{Execution Time}}$$

If the performance of system A is n times better than system B then

$$\begin{aligned} \text{Perf}_A &= n\text{Perf}_B \\ \frac{\text{Perf}_A}{\text{Perf}_B} &= n. \end{aligned}$$

Alternately we note

$$\begin{aligned} \text{Perf}_A &= n\text{Perf}_B \\ \frac{1}{\text{Execution Time}_A} &= n\frac{1}{\text{Execution Time}_B} \\ \frac{\text{Execution Time}_B}{\text{Execution Time}_A} &= n. \end{aligned}$$

Putting all this together we obtain:

$$\frac{\text{Perf}_A}{\text{Perf}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A}.$$

13.6 Measuring CPU Time

$$\begin{aligned} CPUT &= \#\ \text{cycles} \times \text{cycle time} \\ &= \#\ \text{cycles} \times \frac{1}{\text{cycle rate}} \end{aligned}$$

Cycle rate is easily known for a machine so only the # cycles is needed.

13.6.1 First Approximation

$$\begin{aligned} \#\ \text{cycles} &= \#\ \text{instruct} \times \frac{\#\ \text{cycles}}{\#\ \text{instruct}} \\ &= IC \times \text{CPI} \end{aligned}$$

CPI is the cycles per instruction, and IC is the instruction count. It can be measured on average for a running program, and theoretical predictions of it can be made fairly easily.

13.6.2 Second Approximation

CPI for different types of instructions are different. For instance, arithmetic instructions like addition are usually much faster than memory access instructions.

$$\begin{aligned} \#\ \text{cycles} &= IC_{total} CPI_{avg} \\ &= IC_{total} \sum_{i=1}^{n} f_i \times \text{CPI}_i \\ &= IC_{total} \sum_{i=1}^{n} \frac{IC_i}{IC_{total}} \times \text{CPI}_i \\ &= \sum_{i=1}^{n} IC_i \times \text{CPI}_i \end{aligned}$$

where f_i is the frequency of instruction type i. These frequencies can be measured for a large number of software packages to give typical results.

Consider, for example, a program that executes 50,000 instructions running on a machine that is typified by

	ALU	Branch	Memory
CPI	1	3	4
freq	0.5	0.2	0.3

In this case the average CPI of the machine would be given by

$$
\begin{aligned}
CPI_{avg} &= \sum_{i=1}^{n} f_i \times \mathrm{CPI}_i \\
&= .5 \times 1 + .2 \times 3 + .3 \times 4 \\
&= .5 + .6 + 1.2 \\
&= 2.3
\end{aligned}
$$

It is interesting to note that memory accounts for more of the CPI than the other two combined, and branching accounts for more than ALU operations even though there are over twice as many ALU operations.

13.7 Amdahl's Law

The performance difference between two machines, or two configurations of the same machine for that matter, can be compared by setting them as a ratio as we have seen. Let's refer to the performance difference of the two machines as the speedup (S). From what we have seen we can write for two machines a and b that

$$
\begin{aligned}
S &= \frac{P_a}{P_b} \\
&= \frac{T_b}{T_a} \\
&= \frac{IC_b CPI_b \frac{1}{\text{cycle rate}_b}}{IC_a CPI_a \frac{1}{\text{cycle rate}_a}} \\
&= \frac{IC_b CPI_b \text{cycle rate}_a}{IC_a CPI_a \text{cycle rate}_b}
\end{aligned}
$$

Now, let's assume that we are dealing with two versions of the same machine, one enhanced and one not enhanced. If the time of the original code was $T_{original}$, and the instructions that would be speed up by the enhancement took up a fraction, f of the original time and resulted in that portion be completed in $\frac{1}{S_{enhanced}}$ the time, then

$$T_{enhanced} = T_{original}\left((1-f) + f\frac{1}{S_{enhanced}}\right).$$

The speedup, per the second form above is

$$
\begin{aligned}
S_{overall} &= \frac{T_{original}}{T_{enhanced}} \\
&= \frac{T_{original}}{T_{original}\left((1-f) + f\frac{1}{S_{enhanced}}\right)} \\
&= \frac{1}{(1-f) + \frac{f}{S_{enhanced}}}
\end{aligned}
$$

This result can be extended to cover many enhancements, say n of them.

$$S = \frac{1}{(1 - \sum_{i=1}^{n} f_i) + \sum_{i=1}^{n} \frac{f_i}{S_i}}$$

13.7.1 Alternate Approach

We could have assumed that the enhanced time took $T_{enhanced}$, and that the instructions using the enhanced mode took up a fraction g of the enhanced time. If the speedup of the enhanced mode was still $S_{enhanced}$ then

$$T_{original} = T_{enhanced}\left((1-g) + gS_{enhanced}\right)$$

We can relate f and g by noting that

$$\begin{aligned} T_{enhanced}\, g S_{enhanced} &= T_{original} f \\ gS_{enhanced} &= f S_{overall} \end{aligned}$$

By observing that $S_{overall} \leq S_{enhanced}$, with strict inequality if $S_{enhanced} > 1$, we find that $g \leq f$, with strict inequality for the same condition. Alternately, we could note that

$$\begin{aligned} T_{enhanced}(1-g) &= T_{original}(1-f) \\ 1-g &= (1-f)S_{overall} \\ 1-g &= S_{overall} - gS_{enhanced} \\ S_{overall} &= (1-g) + gS_{enhanced} \end{aligned}$$

An alternate way of finding the overall speedup is by using the formula for speedup directly.

$$\begin{aligned} S_{overall} &= \frac{T_{original}}{T_{enhanced}} \\ &= \frac{T_{enhanced}\left((1-g) + gS_{enhanced}\right)}{T_{enhanced}} \\ &= (1-g) + gS_{enhanced} \end{aligned}$$

Since the speedup must be the same, we can also find a formula to calculate the speedup for the enhanced portion in terms of just f and g.

$$\begin{aligned} (1-g) + gS_{enhanced} &= \frac{1}{(1-f) + \frac{f}{S_{enhanced}}} \\ ((1-g) + gS_{enhanced})\left((1-f) + \frac{f}{S_{enhanced}}\right) &= 1 \\ 1 - g - f + fg + (1-g)\frac{f}{S_{enhanced}} + (1-f)gS_{enhanced} + fg &= 1 \\ g(S_{enhanced} - 1) + f\left(\frac{1}{S_{enhanced}} - 1\right) &= fg\left(S_{enhanced} - 1 + \frac{1}{S_{enhanced}} - 1\right) \\ &= fg(S_{enhanced} - 1) + fg\left(\frac{1}{S_{enhanced}} - 1\right) \\ g(1-f)(S_{enhanced} - 1) &= f(1-g)\left(1 - \frac{1}{S_{enhanced}}\right) \\ g(1-f)(S_{enhanced} - 1) &= f(1-g)\frac{S_{enhanced} - 1}{S_{enhanced}} \\ g(1-f)S_{enhanced} &= f(1-g) \\ S_{enhanced} &= \frac{f}{1-f}\frac{1-g}{g} \\ S_{enhanced} &= \frac{f}{g}S_{overall} \end{aligned}$$

We can thus calculate the overall speedup a number of ways

$$\begin{aligned} S_{overall} &= S_{enhanced}\frac{g}{f} \\ &= \frac{1-g}{1-f} \\ &= (1-g) + gS_{enhanced} \\ &= \frac{1}{(1-f) + \frac{f}{S_{enhanced}}} \end{aligned}$$

Consider, for example, that on an unenhanced machine a piece of code runs in 10 seconds, and the instructions that could have used the enhanced mode (were it available) took up 6 seconds of that time. On an enhanced machine the same code uses the enhanced mode for a total of 1 second of the time. What is f and g? What is the speedup of the enhancement and the overall system?

We can find f directly.

$$\begin{aligned} f &= \frac{6sec}{10sec} \\ &= 0.6 \end{aligned}$$

We can find g by noting that the original code has 4 seconds that are not speed up, so the total time after must be 5 seconds.

$$\begin{aligned} g &= \frac{1sec}{5sec} \\ &= 0.2 \end{aligned}$$

If you did not make this observation you could have first found the speedup of the enhanced mode and used it to find g. The speedup of the enhancement is simple, given this information.

$$\begin{aligned} S_{enhanced} &= \frac{6sec}{1sec} \\ &= 6 \end{aligned}$$

Using this, we could have found

$$\begin{aligned} S_{enhanced} &= \frac{f}{1-f}\frac{1-g}{g} \\ 6 &= \frac{0.6}{0.4}\frac{1-g}{g} \\ 4 &= \frac{1-g}{g} \\ 5g &= 1 \\ g &= 0.2 \end{aligned}$$

The same we found before. The overall speedup is equally easy to get, by a bunch of ways.

$$\begin{aligned} S_{overall} &= \frac{T_{original}}{T_{enhanced}} \\ &= \frac{10sec}{5sec} \\ &= 2 \end{aligned}$$

Or

$$\begin{aligned} S_{overall} &= S_{enhanced}\frac{g}{f} \\ &= 6\frac{.2}{.6} \\ &= 2 \end{aligned}$$

Or

$$\begin{aligned} S_{overall} &= \frac{1-g}{1-f} \\ &= \frac{1-.2}{1-.6} \\ &= \frac{.8}{.4} \\ &= 2 \end{aligned}$$

Or

$$\begin{aligned} S_{overall} &= (1-g) + gS_{enhanced} \\ &= (1-0.2) + 0.2 \times 6 \\ &= 0.8 + 1.2 \\ &= 2 \end{aligned}$$

Or

$$\begin{aligned} S_{overall} &= \frac{1}{(1-f) + \frac{f}{S_{enhanced}}} \\ &= \frac{1}{(1-0.6) + \frac{0.6}{6}} \\ &= \frac{1}{0.4 + 0.1} \\ &= \frac{1}{0.5} \\ &= 2 \end{aligned}$$

As you can see, it doesn't matter which formula you use, they all give the same answer. You should also notice that if you improve the enhanced mode more, you will gain almost nothing in the overall speedup. For example consider allowing $S_{enhanced} = \infty$, then

$$\begin{aligned} S_{overall} &= \frac{1}{(1-f) + \frac{f}{S_{enhanced}}} \\ &= \lim_{x\to\infty} \frac{1}{(1-0.6) + \frac{0.6}{x}} \\ &= \frac{1}{0.4} \\ &= 2.5 \end{aligned}$$

In this case $g = 0$ so some of the equations have the indeterminate form $0 \times \infty$, which we avoid by using a form that does not have this problem. The really big thing to see though is that even a huge increase in the speedup of the enhanced mode made little difference, because the non-enhanced portions are dominating. This brings up one of the most basic interpretations of Amdahl's Law, always improve the most common case.

13.7.2 Relating the CPIs

Assuming we are dealing with enhancements to a machine, it is thus reasonable that the code length would not change, so $IC_a = IC_b$. Additionally we will assume it is not a trivial improvement of increasing the clock speed, so cycle rate$_a$ = cycle rate$_b$. Thus

$$\begin{aligned} S &= \frac{CPI_{original}}{CPI_{enhanced}} \\ CPI_{enhanced} &= CPI_{original}\left((1 - \sum_{i=1}^{n} f_i) + \sum_{i=1}^{n} \frac{f_i}{S_i}\right) \end{aligned}$$

Without changing the clock or reducing instructions, we can then find that the maximum speedup possible for a single issue system is $CPI_{original}$, since the ideal CPI for a single issue system is 1.

13.8 Putting It All Together

Example

You are to select a compiler to develop applications for a company with two types of computers. The company wants the best average performance with both machines. Assume all the machines are 1GHz machines.

Type	CPI 1	CPI 2	Compiler 1	Compiler 2
Arithmetic	1	1	35%	30%
Branch	6	3	25%	20%
Memory	3	5	40%	50%

If the code is 10000 lines (for either compiler) when assembled how long does it take to run on each machine?

	Compiler 1	Compiler 2
Machine 1	$1 \times .35 + 6 \times .25 + 3 \times .4 = 3.05$	$1 \times .3 + 6 \times .2 + 3 \times .5 = 3$
Machine 2	$1 \times .35 + 3 \times .25 + 5 \times .4 = 3.1$	$1 \times .3 + 3 \times .2 + 5 \times .5 = 3.4$
Average	3.075	3.2

Since time is the inverse of performance, we want the lowest average and ergo pick compiler 1. If each command runs only once (a bad assumption in reality but we will use it for now), the code will run in:

machine 1: $\frac{10000 \times 3.05}{10^9} = 3.05 \times 10^{-4}$ seconds.

machine 2: $\frac{10000 \times 3.1}{10^9} = 3.1 \times 10^{-4}$ seconds.

Chapter 14

Instruction Level Parallelism

14.1 Trouble In Paradise

There are three types of hazards we can encounter.

Structural hardware cannot support the instruction combo. Big problem in multi-cycle execution, out of order execution, and superscalar, but it can also happen in simple pipelines with things like memory access. Fixing this requires hardware design.

Data data is not available to proceed. Typical solutions fall into two categories, wait till the answer is here or send the answer from where it is now. These are discussed more below.

Control at branch, which do I take and how can I rearrange code around branches in dynamic execution?

14.1.1 Data Hazards

Dependence	Hazard	Example	When
True (data)	RAW	add r2,r3,r4 add r5,r2,r6	When: read happens before the write can finish Requires: pipelining (without forwarding), multi-cycle units, out of order execution, etc.

Output
(name)
	WAW	add r2,r3,r4
		brgtz r7, label
		add r2,r5,r6
			When: instructions finish out of order.
			Requires: out of order execution or multiple can
			multi-cycle execution units.

Antidependence (name)	WAR	add r3,r2,r4 add r2,r5,r6	When: instructions start out of order. Requires: out of order execution
None	RAR	add r3,r2,r4 add r5,r2,r6	There is no problem here, and it is not a hazard. I put it in because people kept asking.

Read after write (RAW) data hazards are also called true dependence or data dependence, because the second instruction actually needs the result from the first. It is the strongest dependence in the sense that it cannot be broken - the second instruction must have the result of the first instruction. Since it is so fundamental, it is the easiest to have happen. RAW occurs when the second instruction tries to access a result before it has been written by the first instruction. This commonly occurs in pipelines, as there are typically multiple cycles after the execute cycle completes till the result is updated in the registers. Each cycle of delay till the update could cause an instruction being decoded to access the wrong value. The two most common solutions to this problem are slips and register forwarding, though register renaming will also handle it (explained in subsection 14.1.2.

Write after write (WAW) hazards is the second most easy data hazard to generate, but the last most people think about. Usually people look at this and wonder if this can ever be a problem. This is actually the most dangerous data hazard in terms of potential to harm your results. Most machines today allow instructions to finish out of order, either by starting out of order, or because some instructions are slower and the fast ones are allowed to pass. If two instructions finish out of order and are writing to the same register, then we have a WAW hazard. The severity of the problem is caused by the number of instructions that are impacted. Normally, the first instruction would finish and its result would be available for use till the second one finished in which case the second answer would be available from then on. When a WAW hazard occurs, the second one finishes first and its result is available in the intermediate time, then the first ones result is available from then on. Unlike a RAW hazard which impacts one instruction (and those dependent on it), WAW can effect many instructions (and those dependent on them). The entire problem is based on the output so it is often called an output dependence. The problem is also due to the reuse of a register for different values, so it is called name dependence (it depends on the register name you picked). It can be fixed by a reorder buffer or register renaming.

Write after read (WAR) hazards are the hardest to occur, and have a small impact, but seem to make reasonable sense to most people. They occur when instructions start out of order causing one instruction to read the result of an instruction that was supposed to happen after it. It can only happen with out of order execution units, and it only effects the instruction that did the read (and those which use its results - but this is true of all data hazards). The dependence is in reverse order so it is sometimes called anti-dependence, but it is also based on reuse of a register so it is also considered a name dependence as WAW is. Both reorder buffers and register renaming will work to solve WAR hazards. The most commonly known algorithm for solving this problem is by Tomasulo and is covered in chapter 16.

14.1.2 Hazard Solutions

What can we do with data hazards. Remove all performance measures and execute single instructions slowly. I'm not kidding, it will work for all problems. The problems are challenges that come from performance improvements, so if you are willing to run non-pipelined, single threaded, non-superscalar processors at a few hundred megahertz you will never hit one of these problems. Your performance will stink, you won't be able to play modern games or movies, but you won't have any problems. Most people want speed, and so we have to come up with other solutions. Here are some of the most famous.

- register interlocking

  This is basically a stop until the data is available. Two variety exists

  Stall Entire processor is held for an instruction (or more), particularly important for structural hazards such as multi-cycle units or memory operations, since the units between the pipeline buffer registers keep running, and thus can finish what they are doing. Essentially this is like slowing the clock down when you need to. This tends to kill performance, but it avoids errors. Stalling will not solve the problems register forwarding will. It is the easiest method to implement.

  Slip only the held-up instruction and those after must wait, others can proceed. Note it could be one of these that produces the desired answer, so this handles the same problems as forwarding, and can handle the problems that stalling does. Overall it is the most versatile (it handles everything stalling and forwarding does), but it is not the fastest solution (same as stalling on performance). It is the second easiest to implement.

- register forwarding

  Often the value exists, it is just not in the final destination yet. This technique sends the value that is missing, to the execution unit. There is no delay if you can do it. It cannot handle multi-cycle execution or memory accesses, and it adds cost and complexity to the design (though not bad for what you get). This is straightforward to implement, but does add several multiplexors, wires, and control circuits to track where the result is (comparators or counters are common).

- register renaming

  Used to solve WAR and WAW hazards. Register renaming adds a status field to each register, which contains the address of the instruction that is calculating its current value or 0, which means it has the correct value. Instructions are fetched and issued in order, so the registers have the correct values in the status field, but are then buffered and executed when the system is ready (kind of like giving them a number and sticking them in a waiting room). It can do almost anything (it can't handle control hazards). The most basic (and famous) of these algorithms is Tomasulo's algorithm, see chapter 16.

- reorder buffer

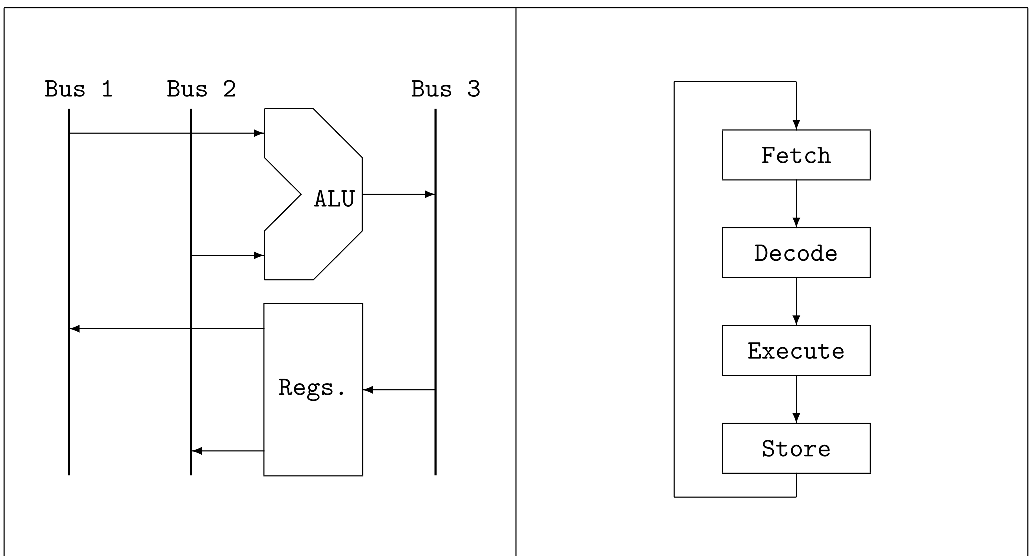
  Instructions are held in a buffer for writing to the register files, then they are written in the order of the original code. These are different buffers than the pipeline buffers. This preserves the order of the writes and thus solves WAR and WAW hazards, but increases the latency of the instruction execution. On the bright side it can handle control hazards (the only one listed that can).

Chapter 15

Pipelining

15.1 Basic Architecture

Consider the following architecture.



Architecture

Fetch/Execute

The architecture and Fetch/Execute loop, lend themselves to a four stage pipeline. We will make each of the stages in the Fetch/Execute loop to be a stage in our pipeline.

Use registers at boundaries of hardware portions that do the stages of the IFetch (more fully to separate the clock cycles).

15.1.1 Calculating efficiency

Our basic equations of pipeline performance are

$$\text{speedup} = \frac{T_{\text{original}}}{T_{\text{modified}}}$$

$$\text{efficiency} = \frac{\text{actual speedup}}{\text{ideal speedup}}$$

Consider m instructions running on a computer with n stages. If this is not pipelined then the time of execution will take $T_{nopipe} = m \times n \times T_{clock}$. To get this we just used that $T = \#\, cycles \times T_{clock}$. If it is pipelined then the execution will take $T_{pipe} = (m + n - 1) \times T_{clock}$. To see why consider this for m¿¿n (the usual case)

	Cycle								
Instruction	0	1	...	n-1	n	...	m-1	...	m+n-1
Inst 1	x	x	...	x					
Inst 2		x	...	x	x				
⋮			⋱	⋱	⋱	⋱			
⋮						⋱	⋱	⋱	
Inst m							x	...	x

Using this we can find that as the speedup of pipelining for m instructions in an n stage machine as m gets very large (long program run) is

$$\begin{aligned}
\text{speedup} &= \frac{T_{nopipe}}{T_{pipe}} \\
&= \lim_{m\to\infty} \frac{mnT_{clock}}{(m+n-1)T_{clock}} \\
&= \lim_{m\to\infty} \frac{mn}{m+n-1} \\
&= n
\end{aligned}$$

Yielding the famous result that the ideal speedup is the number of stages in a pipeline. If a stall were to happen a finite number of times it would not effect the asymptotic speedup, however if a stall happened a fraction of the time that is a different matter. For instance, assume the pipeline stalls P_{err} cycles in $f_{T,err}$ of all instructions of type T ($m \times fT$ total instructions) then the time of the pipelined machine would be $T_{pipe} = (m + n - 1 + mf_T f_{err} P_{err}) \times T_{clock}$. The non-ideal speedup would be

$$\begin{aligned}
\text{speedup} &= \frac{T_{nopipe}}{T_{pipe}} \\
&= \lim_{m\to\infty} \frac{mnT_{clock}}{(m+n-1+mf_T f_{err} P_{err})T_{clock}} \\
&= \lim_{m\to\infty} \frac{mn}{m+n-1+mf_T f_{T,err} P_{err}} \\
&= \frac{n}{1+f_T f_{T,err} P_{err}} \\
&= \frac{n}{1+f_{err} P_{err}}
\end{aligned}$$

where $f_{err} = f_T f_{T,err}$. Note that the numerator is the CPI of the non-pipelined machine and the denominator is the CPI of the non-ideal pipelined machine. Thus we see that CPI for a pipelined machine is

$$CPI = 1 + \sum_{i=1}^{n} f_i P_i.$$

If there are no errors the ideal CPI is thus 1. Consider an example of this with branches incurring a penalty when they taken (i.e. the machine assumes branch not taken).

$$CPI_{avg} = (1 - P_b)CPI_{no\,branch} + P_b((1 - P_{take})CPI_{no\,branch} + P_{take}(1 + b))$$

CPI Cycles per instruction. The smaller the better. Nominally for a RISC machine this will be 1, but bubbles will increase it and pipelining will decrease it.

P Probability that something will happen (the event is indicated by the subscript).

b Branch penalty, which indicates how large the bubble in the pipeline is, that is caused by taking a branch.

15.1.2 Branch Prediction

Normally branches are assumed to be not taken but this is a simplistic assumption. A more sophisticated choice is to do what was done most recently. So for instance if the second instruction is a branch, and last time I was there I took it, I would have:

Address	Taken
0	0
1	0
2	1
3	0

This would require an extra bit for every memory location, most of which would be unused.

Performance

A pipelined RISC computer has 8 stages, and runs at 1.25 GHz. The cache has a miss rate of 1% for data and instructions, and a miss penalty of 24 ns. The system has a dynamic branch predictor that is wrong only 10% of the time. Branch errors cost 5 cycles.

1. What is the ideal (no stalls) speedup over a non-pipelined machine?
2. What is the impact to the CPI due to cache misses on a non-memory operation?
3. What is the impact to the CPI due to cache misses on a memory operation?
4. What is the impact to the CPI due to branch errors on branching instructions?
5. If memory operation make up 20% of the commands in a typical program and branching make up 15% of the commands, what is the average CPI?

1. $n = \frac{\text{Time Without Pipeline}}{\text{Time With Pipeline}} = \frac{I \times 8}{I+8} \approx 8$ for large I (number of instructions).
2. $\Delta CPI = \text{Miss Rate} \times \text{Miss Penalty} \times \text{Clock Frequency} = (.01)(24ns)(1.25GHz) = .3$
3. Twice above or (0.6).
4. $\Delta CPI = \text{Branch Error Rate} \times (BranchPenalty) = .1 \times 5 = .5$
5. $CPI_{avg} = .2(1 + .6) + .15(1 + .3 + .5) + .65(1 + .3) = .32 + .27 + .845 = 1.435$

Superscalar

Superscalar pipelines have multiple pipelines to execute commands on (for example the latest pentium has 2). The advantage is that a machine with n pipelines could have a CPI of $\frac{1}{n}$. They have their own challenges in programming though.

Consider the following section of a program:

```
loop:  lw $t3,0($t1)       # first data
       add $t5, $t5, $t3   # running sum
       addi $t1, $t1, 4    # increment counter
       brne $t0, $t1, loop # check if done
exit:
```

And place the commands to be scheduled on two pipelines in the most obvious way.

Pipeline 1	Pipeline 2
lw $t3,0($t1)	Nop
add $t5, $t5, $t3	addi $t1, $t1, 4
brne $t0, $t1, loop	Nop

Granting myself a perfect branch predictor, so I have no stalls due to branching (in class we considered stalls), I still only get:

$$CPI = \frac{3}{4} = .75$$

Now consider a clever rearrangement:

Pipeline 1	Pipeline 2
lw $t3,0($t1)	addi $t1, $t1, 4
add $t5, $t5, $t3	brne $t0, $t1, loop

Granting myself a perfect branch predictor, I get:

$$CPI = \frac{2}{4} = .5$$

Can I always do such a rearrangement? Sorry but no. Consider the following:

```
loop:  lw $t3,0($t1)        # first data
       mult $t3, $t1        # multiplication
       mflo $t3             # get the product
       add $t5, $t5, $t3    # running sum
       addi $t1, $t1, 4     # increment counter
       brne $t0, $t1, loop  # check if done
exit:
```

And place the commands to be scheduled on two pipelines in the most obvious way.

Pipeline 1	Pipeline 2
lw $t3,0($t1)	Nop
mult $t3, $t1	addi $t1, $t1, 4
mflo $t3	Nop
add $t5, $t5, $t3	brne $t0, $t1, loop

Granting myself a perfect branch predictor, so I have no stalls due to branching, I still only get:

$$CPI = \frac{4}{6} = .66$$

And note that the second pipeline is only half used.

15.2 Unrolling

Now let us unroll the loop, by considering two runs through at once. Note that on the second run through the data accessed is at four bytes higher than the first run.

```
loop:  lw $t3,0($t1)        # first data
       lw $t4,4($t1)        # second data
       mult $t3, $t1        # multiplication
       mflo $t3             # get the product
       add $t5, $t5, $t3    # running sum
       addi $t1, $t1, 4     # increment counter
       breq $t0, $t1, exit  # check if done
       mult $t4, $t1        # multiplication
       mflo $t4             # get the product
       add $t5, $t5, $t4    # running sum
       addi $t1, $t1, 4     # increment counter
       brne $t0, $t1, loop  # check if done
exit:
```

Pipeline 1	Pipeline 2
lw $t3,0($t1)	lw $t4,4($t1)
mult $t3, $t1	addi $t1, $t1, 4
mflo $t3	mult $t4, $t1
add $t5, $t5, $t3	breq $t0, $t1, exit
mflo $t4	addi $t1, $t1, 4
add $t5, $t5, $t4	brne $t0, $t1, loop

Granting myself a perfect branch predictor, so I have no stalls due to branching, I now get:

$$CPI = \frac{6}{12} = .5$$

As a general rule you unroll n copies of the loop for a machine with n pipelines. In this case I unrolled 2 copies because I had two pipes to fill.

15.3 Unrolling, Part II

Consider the following code to calculate the Fibonacci numbers.

```
top: add r4, r3, r2
     mov r2, r3
     mov r3, r4
     addi r1, r1, -1
     brgtz r1, top
```

The first three instructions are the data manipulations, and the last two are loop overhead (indexing and branching). There is a large amount of wasted effort spent in moving data around. Consider two loops worth of just the data manipulation portions.

```
add r4, r3, r2
mov r2, r3
mov r3, r4
add r4, r3, r2
mov r2, r3
mov r3, r4
```

Note that the "mov" commands are only to set up the problem for the next loop. In particular the contents of r2 are removed and the contents of r3 and r4 are shuffled. Consider the following change.

```
add r2, r3, r2
add r4, r3, r2
mov r3, r4
```

The contents of the registers are the same at the end of the loop, as the original, but considerable savings have been achieved. by noting the last mov command only shifts the results of the second add, we note that it is equivalent to the following

```
add r2, r3, r2
add r3, r3, r2
```

Thus by unrolling we can see the loop is equivalent to

```
 top: add r2, r3, r2
      add r3, r3, r2
      addi r1, r1, -2
      brgtz r1, top
      mov r4, r3
      breqz r1, exit
      mov r4, r2
exit:
```

Note the last three commands are cleanup only, so two iterations of the original loop can be done in less instructions than the unoptimized code. The loop can be scheduled efficiently on a two pipeline machine as

top:	add r2, r3, r2	addi r1, r1, -2
	add r3, r3, r2	bgtqz r1, top
	mov r4, r3	breqz r1, exit
	mov r4, r2	
exit:		

15.4 Software Pipelining

Returning to the original code

```
top: add r4, r3, r2
     mov r2, r3
     mov r3, r4
     addi r1, r1, -1
     brgtz r1, top
```

And let us again consider two iterations of the Fibonacci number loop.

```
add r4, r3, r2
mov r2, r3
mov r3, r4
add r4, r3, r2
mov r2, r3
mov r3, r4
```

First note that each pair of moves can be done simultaneously.

```
add r4, r3, r2
mov r2, r3         mov r3, r4
add r4, r3, r2
mov r2, r3         mov r3, r4
```

Now we will move the second add ahead in the scheduling so it is simultaneous with the first moves.

```
add r4, r3, r2
mov r2, r3         mov r3, r4         add r4, r4, r3
mov r2, r3         mov r3, r4
```

Now note that the `mov r2, r3` commands are useless and can be dropped.

```
add r4, r3, r2
mov r3, r4         add r4, r4, r3
mov r3, r4
```

This suggests the following parallel exececuation

mov r2, r3	add r3, r3, r2	addi r1, r1, -1	brgtz r1, top

time	r3	r2	r1
0	1	1	3
1	2	1	2
2	3	2	1
3	5	3	0

15.4.1 Example

Consider the following code

```
top: ld r2, 0(r1)
     addi r3, r2, 1
     st r3, 0(r1)
     addi r1, r1, 4
     brlt r1, r4, top
```

st r3, 0(r1)	addi r3, r2, 1	ld r2, 8(r1)

Chapter 16

Tomasulo

16.1 Multiple Issue Tomasulo

To illustrate the method we will consider a simple piece of code.

```
loop:
  mul  $t4,$t2
  mflo $t4
  subi $t3,$t3,1
  bgtz $t3,loop
```

This code will calculate $\$t4 = \$t2^{\$t3}$, assuming $\$t4 = 1$ initially and $\$t2 > 0$ and $\$t3 > 1$.

Further lets assume add/sub/move takes 1 cycle of execution, multiply takes 2 cycles, and branches take 2 cycle. The branch predictor will always predict branch taken in this example. Let's schedule this for our machine.

Cycle 1

Reorder Buffer					
Entry	Busy	Instruction	State	Destination	Value
1	yes	mul $t4,$t2	Issue	$Hi, $Lo	
2	yes	mflo $t4	Issue	$t4	
3					
4					
5					
6					
7					
8					
9					
10					

Registers			
Field	Data	Reorder	Busy
$t0			
$t1			
$t2	5		
$t3	2		
$t4	1	#2	yes
$t5			
$t6			
$t7			
$t8			
$t9			

Reservation Station								
Name	Busy	Op	V_1	V_2	S_1	S_2	Dest	A
Add1		mflo			#1		#2	
Add2								
Add3								
Add4								
Mul1		mul	1	5			#1	
Mul2								
Br1								
Br2								

Cycle 2

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
1	yes	mul $t4,$t2	Exec	$Hi, $Lo	
2	yes	mflo $t4	Issue	$t4	
3	yes	subi $t3,$t3,1	Issue	$t3	
4	yes	bgtz $t3,loop	Issue		
5					
6					
7					
8					
9					
10					

Registers

Field	Data	Reorder	Busy
$t0			
$t1			
$t2	5		
$t3	2	#3	yes
$t4	1	#2	yes
$t5			
$t6			
$t7			
$t8			
$t9			

Reservation Station

Name	Busy	Op	V_1	V_2	S_1	S_2	Dest	A
Add1		mflo			#1		#2	
Add2		subi	2	1			#3	
Add3								
Add4								
Mul1	yes	mul	1	5			#1	
Mul2								
Br1		bgtz			#3		#4	
Br2								

Cycle 3

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
1	yes	mul $t4,$t2	Exec	$Hi, $Lo	
2	yes	mflo $t4	Issue	$t4	
3	yes	subi $t3,$t3,1	Exec	$t3	
4	yes	bgtz $t3,loop	Issue		
5	yes	mul $t4,$t2	Issue	$Hi, $Lo	
6	yes	mflo $t4	Issue	$t4	
7					
8					
9					
10					

Registers

Field	Data	Reorder	Busy
$t0			
$t1			
$t2	5		
$t3	2	#3	yes
$t4	1	#6	yes
$t5			
$t6			
$t7			
$t8			
$t9			

Reservation Station

Name	Busy	Op	V_1	V_2	S_1	S_2	Dest	A
Add1		mflo			#1		#2	
Add2	yes	subi	2	1			#3	
Add3		mflo			#5		#6	
Add4								
Mul1	yes	mul	1	5			#1	
Mul2		mul		5	#2		#5	
Br1		bgtz			#3		#4	
Br2								

Cycle 4

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
1	no	mul $t4,$t2	Commit	$Hi, $Lo	5
2	yes	mflo $t4	Exec	$t4	
3	no	subi $t3,$t3,1	done	$t3	1
4	yes	bgtz $t3,loop	Exec		
5	yes	mul $t4,$t2	Issue	$Hi, $Lo	
6	yes	mflo $t4	Issue	$t4	
7	yes	subi $t3,$t3,1	Issue	$t3	
8	yes	bgtz $t3,loop	Issue		
9					
10					

Registers

Field	Data	Reorder	Busy
$t0			
$t1			
$t2	5		
$t3	1	#7	yes
$t4	1	#6	yes
$t5			
$t6			
$t7			
$t8			
$t9			

Reservation Station

Name	Busy	Op	V_1	V_2	S_1	S_2	Dest	A
Add1	yes	mflo	5				#2	
Add2		subi	1	1			#7	
Add3		mflo			#5		#6	
Add4								
Mul1								
Mul2		mul		5	#2		#5	
Br1	yes	bgtz	1				#4	
Br2		bgtz		#7			#8	

Cycle 5

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
1					
2	no	mflo $t4	Commit	$t4	5
3	no	subi $t3,$t3,1	Commit	$t3	1
4	yes	bgtz $t3,loop	Exec		
5	yes	mul $t4,$t2	Exec	$Hi, $Lo	
6	yes	mflo $t4	Issue	$t4	
7	yes	subi $t3,$t3,1	Exec	$t3	
8	yes	bgtz $t3,loop	Issue		
9	yes	mul $t4,$t2	Issue	$Hi, $Lo	
10	yes	mflo $t4	Issue	$t4	

Registers

Field	Data	Reorder	Busy
$t0			
$t1			
$t2	5		
$t3	1	#7	yes
$t4	5	#10	yes
$t5			
$t6			
$t7			
$t8			
$t9			

Reservation Station

Name	Busy	Op	V_1	V_2	S_1	S_2	Dest	A
Add1		mflo			#9		#10	
Add2	yes	subi	1	1			#7	
Add3		mflo			#5		#6	
Add4								
Mul1		mul		5	#6		#9	
Mul2	yes	mul	5	5			#5	
Br1	yes	bgtz	1				#4	
Br2		bgtz		#7			#8	

Cycle 6

Reorder Buffer						Registers			
Entry	Busy	Instruction	State	Destination	Value	Field	Data	Reorder	Busy
1	yes	subi \$t3,\$t3,1	Issue	\$t3		\$t0			
2	yes	bgtz \$t3,loop	Issue			\$t1			
3						\$t2	5		
4	no	bgtz \$t3,loop	Commit			\$t3	1	#1	yes
5	yes	mul \$t4,\$t2	Exec	\$Hi, \$Lo		\$t4	5	#10	yes
6	yes	mflo \$t4	Issue	\$t4		\$t5			
7	no	subi \$t3,\$t3,1	Done	\$t3	0	\$t6			
8	yes	bgtz \$t3,loop	Issue			\$t7			
9	yes	mul \$t4,\$t2	Issue	\$Hi, \$Lo		\$t8			
10	yes	mflo \$t4	Issue	\$t4		\$t9			

Reservation Station

Name	Busy	Op	V_1	V_2	S_1	S_2	Dest	A
Add1		mflo			#9		#10	
Add2		subi	0	1			#1	
Add3		mflo			#5		#6	
Add4								
Mul1		mul		5	#6		#9	
Mul2	yes	mul	5	5			#5	
Br1		bgtz			#2		#2	
Br2	yes	bgtz	0				#8	

Cycle 7

Reorder Buffer						Registers			
Entry	Busy	Instruction	State	Destination	Value	Field	Data	Reorder	Busy
1	yes	subi \$t3,\$t3,1	Issue	\$t3		\$t0			
2	yes	bgtz \$t3,loop	Issue			\$t1			
3						\$t2	5		
4						\$t3	1	#1	yes
5	no	mul \$t4,\$t2	Commit	\$Hi, \$Lo	25	\$t4	5	#10	yes
6	yes	mflo \$t4	Exec	\$t4		\$t5			
7	no	subi \$t3,\$t3,1	Done	\$t3	0	\$t6			
8	yes	bgtz \$t3,loop	Exec			\$t7			
9	yes	mul \$t4,\$t2	Issue	\$Hi, \$Lo		\$t8			
10	yes	mflo \$t4	Issue	\$t4		\$t9			

Reservation Station

Name	Busy	Op	V_1	V_2	S_1	S_2	Dest	A
Add1		mflo			#9		#10	
Add2		subi	0	1			#1	
Add3	yes	mflo	25				#6	
Add4								
Mul1		mul		5	#6		#9	
Mul2								
Br1		bgtz			#2		#2	
Br2	yes	bgtz	0				#8	

Cycle 8

Reorder Buffer					
Entry	Busy	Instruction	State	Destination	Value
1	yes	subi $t3,$t3,1	Exec	$t3	
2	yes	bgtz $t3,loop	Issue		
3	yes	mul $t4,$t2	Issue		
4	yes	mflo $t4	Issue		
5					
6					
7					
8	no	bgtz $t3,loop	Flush		
9	yes	mul $t4,$t2	Exec	$Hi, $Lo	
10	yes	mflo $t4	Issue	$t4	

Registers			
Field	Data	Reorder	Busy
$t0			
$t1			
$t2	5		
$t3	0	#1	yes
$t4	25	#4	yes
$t5			
$t6			
$t7			
$t8			
$t9			

Reservation Station								
Name	Busy	Op	V_1	V_2	S_1	S_2	Dest	A
Add1		mflo			#9		#10	
Add2	yes	subi	0	1			#1	
Add3		mflo			#3		#4	
Add4								
Mul1	yes	mul	25	5			#9	
Mul2		mul		5	#10		#3	
Br1		bgtz			#2		#2	
Br2								

At this point the buffers and stations will be flushed, the executions cancelled, and the registers not updated (they are at the right point). New commands will be loaded from after the branch, and execution proceeds normally.

Chapter 17

Thread Level Parallelism

17.1 Taxonomy

Flynn

SISD Single Instruction Single Data (Modern uniprocessors)

SIMD Single Instruction Multiple Data (Vector machines, and some multimedia)

MISD Multiple Instruction Single Data (No commercial, possible in special applications)

MIMD Multiple Instruction Multiple Data (Modern multiprocessors)

MIMD is broken into two groups based on memory configuration. Memory is either shared equally by all processors or distributed among the processors.

17.2 Shared Memory

The first group centralizes the memory and has each processor with its cache connect via a shared memory bus.

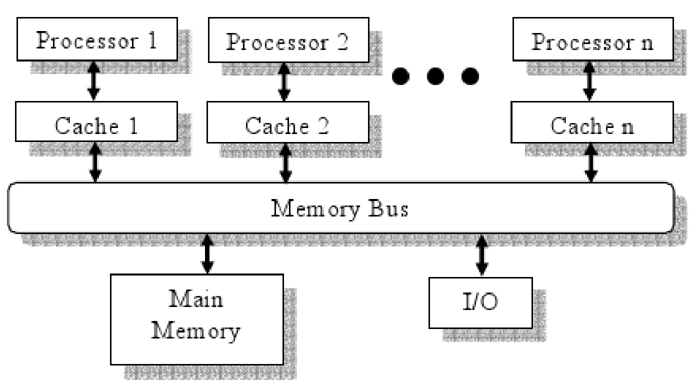


Figure 17.1: Centralized shared memory multiprocessor

The first group is also referred to by

- Centralized Shared Memory
- Symmetric Multiprocessors (SMP)
- Uniform Memory Access (UMA)

These alternate titles are used since the the memory is central and shared, it is thus symmetric to all, and thus the access for each processor is uniform. The main problem here is that as the number of processors grows, the need for memory bandwidth grows. Without the needed bandwidth, requests will have to be scheduled resulting in increased latency.

Example 17 *Using Figure 6.10 in the book, fill in the table, assuming all events are for an address relative to a cache in a SMP system.*

Event	*Source*	*State*
Startup	-	*Invalid*
Read Miss	*CPU*	
Read Miss	*Bus*	
Write Hit	*CPU*	
Write Miss	*Bus*	
Write Miss	*CPU*	
Read Miss	*Bus*	

Event	*Source*	*State*
Startup	-	*Invalid*
Read Miss	*CPU*	*Shared*
Read Miss	*Bus*	*Shared*
Write Hit	*CPU*	*Exclusive*
Write Miss	*Bus*	*Invalid*
Write Miss	*CPU*	*Exclusive*
Read Miss	*Bus*	*Shared*

17.3 Distributed Memory

The second group distributes the memory to each processor so the memory bandwidth grows with the need. This results in the problem of data sharing and communications between the nodes. We could just treat the distributed memories like one big memory, giving each an address (shared address space). This would allow the memories to be shared. Access to different parts of memory is no longer uniform (addresses corresponding to "local" memory will be fast and the addresses corresponding to "remote" memory will be slow). This scheme is referred to as

- Distributed Shared Memory (DSM)
- Nonuniform Memory Access (NUMA)

Alternately we could keep each address space separate (local addresses) and pass messages between nodes containing the data or communications. This scheme makes each machine look like an individual computer (multi-computers) and often each processor is a separate machine (clusters).

Shades of grey exist between the two, for instance a network OS can use message passing to pass a page of memory and implement what looks like shared address space by utilizing paging capabilities.

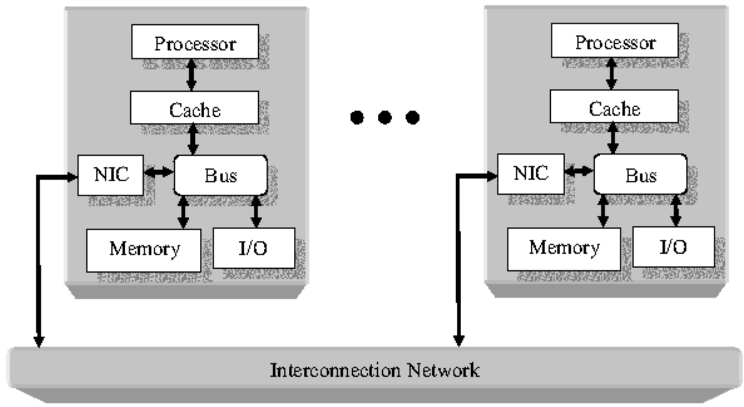


Figure 17.2: Distributed memory multiprocessor

17.4 Performance

Amdahl's Law, for n processors is

$$S = \frac{1}{\sum_{i=1}^{n} \left(\frac{f_i}{i}\right)}, \tag{17.1}$$

where f_i is the fraction of time when i processors are busy. Note that

$$\sum_{i=1}^{n} f_i = 1. \tag{17.2}$$

Example 18 *Consider a 4 processor machine. What must the fractions be to ensure a speedup of at least 3.*

$$\begin{aligned} 3 &= \frac{1}{\frac{f_1}{1} + \frac{f_2}{2} + \frac{f_3}{3} + \frac{f_4}{4}} \\ 1 &= 3\left(\frac{f_1}{1} + \frac{f_2}{2} + \frac{f_3}{3} + \frac{f_4}{4}\right) \\ 4 &= 12f_1 + 6f_2 + 4f_3 + 3f_4 \end{aligned}$$

Note that if the least common multiple of the numbers 1 *through* n *is denoted* LCM*, then for an* n *processor system trying to achieve a speedup of* s *we can say*

$$\frac{LCM}{s} = \sum_{i=1}^{n} \frac{LCM}{i} f_i$$

is the equation describing this situation that has integer coefficients. We also know

$$1 \quad = \quad f_1 + f_2 + f_3 + f_4.$$

Combining yields

$$\begin{bmatrix} 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 12 & 6 & 4 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}.$$

This is indefinite (more unknowns than equations), but we can solve for the fractions in terms of f_1 and f_2.

$$\begin{aligned} 8f_1 + 2f_2 &= f_4 \\ 1 - 9f_1 - 3f_2 &= f_3 \end{aligned}$$

The second equation implies that individually $f_1 < \frac{1}{9} \approx .11$ and $f_2 < \frac{1}{3} \approx .33$ and together $3f_1 + f_2 \leq \frac{1}{3}$. Further, if f_3 is negligible then $.67 \leq f_4 \leq .88$ is the minimum range to ensure a speedup of 3.

The last example shows how great the required thread level parallelism is to achieve a reasonable speedup. The lack of thread level parallelism is one of the two great problems/challenges in multiprocessing. The other great problem/challenge is the latency of remote accesses, which effectively adds a fixed penalty to the CPI of each processor thereby limiting performance.

The efficiency is given by

$$E = \sum_{i=1}^{n} \left(f_i \frac{i}{n} \right) \leq 1 \tag{17.3}$$

Scientific programs are often used to benchmark multiprocessor performance. For the following table n is the problem size, p is the number of processors, and the α numbers are the scaling factors.

Application	$\alpha_{compute}$	$\alpha_{communicate}$
FFT	$\frac{n \log n}{p}$	$\frac{n}{p}$
LU/Ocean	$\frac{n}{p}$	$\sqrt{\frac{n}{p}}$
Barnes-Hut	$\frac{n \log n}{p}$	$\sqrt{\frac{n}{p}} \log n$

Example 19 *An Ocean application takes 1 hour to run on a uniprocessor, and 33 minutes on a dual processor. How long will it take to run the Ocean application on a problem 16 times the original size on a 128 processor machine?*

Processors(p)	*Size(n)*	*Time*	*Computation*	*Communication*
1	*1*	*1 hour*	1	0
2	*1*	*33 min*	$\frac{1}{2}$	$\frac{\sqrt{1}}{\sqrt{2}}$
128	*16*		$\frac{16}{128}$ $\frac{1}{8}$	$\frac{\sqrt{16}}{\sqrt{128}}$ $\frac{1}{2\sqrt{2}}$

From the table, if we call the base time of computation x, and the base time of communication y then we get (using minutes to be consistent):

$$60 = 1x + 0y \tag{17.4}$$

$$33 = .5x + \frac{1}{\sqrt{2}}y \tag{17.5}$$

from the first equation, $x = 60min$ and from the second equation $y = 3\sqrt{2}min$. Thus the time for the third case (problem solution) is

$$\begin{aligned} T &= \frac{1}{8}x + \frac{1}{2\sqrt{2}}y \\ &= \frac{1}{8}60 + \frac{1}{2\sqrt{2}}3\sqrt{2} \\ &= 7.5 + 1.5 \\ &= 9min \end{aligned}$$

Example 20 *An LU application runs in 4000 seconds on a uniprocessor. The same problem runs in 1020 seconds on a four processor machine. How long will it take to run on a 16 processor machine? How long will it take to run on a 64 processor machine?*

$$\begin{aligned} 4000 &= 1x + 0y \\ 1020 &= .25x + .5y \end{aligned}$$

So $x = 4000$ *and* $y = 40$*. Using this,*

$$\begin{aligned} \frac{1}{16}4000 + \frac{1}{4}40 &= 250 + 10 \\ &= 260 \end{aligned}$$

So a 16 processor machine runs it in 260 seconds.

$$\begin{aligned} \frac{1}{64}4000 + \frac{1}{8}40 &= 62.5 + 5 \\ &= 67.5 \end{aligned}$$

Note that communication takes up $\frac{20}{1020} \approx .0196$ *or just under 2% of the time on a four processor. When we have a sixteen processor machine it takes up* $\frac{10}{260} \approx .0385$ *or about 3.85% of the time. On the 64 processor machine the communication took up* $\frac{5}{67.5} \approx .0385$ *or about 7.41% of the time. Communication takes ever increasing fraction of the time, and becomes a limit to performance. Consider running this on a 4096 processor machine. It would take*

$$\begin{aligned} \frac{1}{4096}4000 + \frac{1}{64}40 &\approx .977 + .625 \\ &\approx 1.6 \end{aligned}$$

Thus communication takes $\frac{.625}{1.6} \approx .391$ *or almost 40% of the time!*

Appendix A

Sample Computers

A.1 32 Bit Pipelined Computer

Consider a 32 bit pipelined computer with a 1.0 GHz clock and an ISA that has three categories of commands:

	Freq
Branch	.2
Memory	.3
Other	.5

The computer has a 64 bit memory bus that operates at 500 MHz. The bus requires that requests and responses take 1 cycle. The memory takes 40ns to respond to a request and can do burst sends with a delay of 10ns. The bus requires 3 cycles to initiate a request and 2 cycles to transmit the response. The bus is DMA and requires 710 CPU cycles to set up a transfer, 275 cycles to complete, 500 cycles to handle errors (1% of the time).

The machine has two disks that have a combined transfer rate of 20MB/s, and a total latency of 6.8 ms. The computer has virtual memory with a page size of 64KB.

1. What is the bandwidth of the bus?

   We have been assuming the installed RAM to be integral in the bus design, so the answer would be:

$$\begin{aligned}
\text{Bandwidth} &= \frac{\text{Data Transferred}}{\text{Time of Transfer}} \\
&= \frac{\text{Data Transferred}}{\text{Number of Cycles} \times \text{Time of 1 Cycle}} \\
&= \frac{\text{Data Transferred} \times \text{Bus Clock Frequency}}{\text{Cycles to Initiate} + \text{Cycles to Respond} + \text{Cycles to Get Data}} \\
&= \frac{8\text{Bytes} \times 500\text{MHz}}{3 + 2 + (40\text{ns} \times 500\text{MHz})} \\
&= \frac{4000\text{MB/s}}{25} \\
&= 160\text{MB/s}
\end{aligned}$$

   You might have noted that be RAM supports a burst transfer mode. As the size of the burst increases the effective time to get the data approaches the burst time of 10 ns (down from 40 ns). If you assumed this you would have found the bandwidth to be 400 MB/s.

2. If the computer had to continually page, how much of the CPU's time and the bus's bandwidth would it use?

   Note that in memory KB = 2^{10} bytes, but in networks KB = 10^3 bytes. As they are similar, we will ignore the difference as the book does. Additionally, we will assume the pages are spread across both disks so as to maximize the transfer.

   The time it takes to transfer one page is given by:

$$\begin{aligned} T_{\text{transfer}} &= \text{time to get the data} + \text{time to send the data} \\ &= \text{total latency} + \frac{\text{Data Sent}}{\text{Transmission Rate}} \\ &= 6.8\text{ms} + \frac{64\text{KB}}{20\text{MB/s}} \\ &= 6.8\text{ms} + 3.2\text{ms} \\ &= 10\text{ms}. \end{aligned}$$

   The data rate for the transfer is:

$$\begin{aligned} R_{\text{Data}} &= \frac{\text{Data Sent}}{T_{\text{transfer}}} \\ &= \frac{64\text{KB}}{10\text{ms}} \\ &= 6.4\text{MB/s}. \end{aligned}$$

   Using the figure of 160 MB/s for the bus's bandwidth we find:

$$\begin{aligned} \text{Percent Utilization of Bus} &= \frac{\text{Bandwidth Used}}{\text{Bandwidth Available}} \times 100\% \\ &= \frac{6.4\text{MB/s}}{160\text{MB/s}} \times 100\% \\ &= 4\%. \end{aligned}$$

   Now let's look at the impact on the CPU. We need to find the number of cycles the CPU must use to handle the transfer.

$$\begin{aligned} \text{Cycles Per Transfer} &= \frac{\text{Cycles to Set Up} + \text{Cycles to Finish} + \text{error rate} \times \text{Cycle to Handle Errors}}{1 - \text{error rate}} \\ &= \frac{710 + 275 + .01 \times 500}{1 - .01} \\ &= \frac{990}{.99} \\ &= 1000 \end{aligned}$$

   The utilization of the CPU is thus:

$$\begin{aligned} \text{Percent Utilization of CPU} &= \frac{\frac{\text{Cycles Per Transfer}}{T_{\text{transfer}}}}{\text{CPU Clock Frequency}} \times 100\% \\ &= \frac{\frac{1000}{10\text{ms}}}{1\text{GHz}} \times 100\% \\ &= 0.01\% \end{aligned}$$

   Thus we have a negligible impact.

3. What block size of the cache would cause the least impact on the CPI of the computer due to misses, assuming the instruction and data miss rate are equal?

Block Size	2 words	4 words	8 words	16 words
Miss Rate	4%	2%	1.2%	1%

The average increase to a command's CPI due to cache misses depends on if the command accesses memory just for the instruction fetch or also for the commands implementation. We will therefor assess the impact to memory commands separate from branch and other commands. The average increase for branch and other commands is given by:

$$\begin{aligned}\Delta\text{CPI} &= \text{miss rate} \times \text{Bus Cycles to Transfer} \times \frac{\text{CPU Clock Rate}}{\text{Bus Clock Rate}} \\ &= \text{miss rate} \times \text{Bus Cycles to Transfer} \times 2.\end{aligned}$$

The average impact for branch commands is twice the increase of branch and other commands.

The bus cycles to transfer $2N$ words is given by:

$$\begin{aligned}\text{Cycle to Transfer} &= \text{Cycles to Initiate} + \text{Cycles to Get First 2 Words} \\ &\quad +(N-1) \times \text{Cycle to Burst Get 2 Words} \\ &\quad +N \times \text{Cycles to Send 2 Words} \\ &= 3 + (40\text{ns} \times 500\text{MHz}) + (N-1) \times (10\text{ns} \times 500\text{MHz}) + N \times 2 \\ &= 18 + 7N\end{aligned}$$

Block Size	2 words	4 words
Miss Rate	4%	2%
Bus Cycles to Transfer	18+7(1)=25	18+7(2)=32
ΔCPI Not Memory	(.04)(25)(2)=2	(.02)(32)(2)=1.28
ΔCPI Memory	(2)(2)=4	(2)(1.28)=2.56

Block Size	8 words	16 words
Miss Rate	1.2%	1%
Bus Cycles to Transfer	18+7(4)=46	18+7(8)=74
ΔCPI Not Memory	(.012)(46)(2)=1.104	(.01)(74)(2)=1.48
ΔCPI Memory	(2)(1.104)=2.208	(2)(1.48)=2.96

The least impact is given by a cache with blocks of 8 words in this case.

4. Design a dynamic branch predictor for the computer.

A good estimate of whether a branch will be taken is to remember whether it was taken last time. Remembering if a branch was taken or not requires 1 bit per instruction tracked. To keep the problem realistic we will add an additional 32-bit register. Each bit in the register will indicate if the branch was taken for the last instruction whose address modulo 32 corresponds to the bit's location. An easy way to implement this would be to take the outputs of the 32 bits and pass them into a 32×1 MUX, whose address select lines are given the last five bits of the command's address (from PC for instance). The branch taken signal could be sent from the control to the particular bit by using a 1×32 DeMUX.

5. For this system, 60% of the branch instructions make loops and the rest are for conditional execution. On average, the code in a loop is executed 10 times. What fraction of the time does your dynamic branch predictor, correctly predict the branch taken?

Loops occur 60% of the time, conditional execution occurs 40% of the time. The dynamic branch selected above does not likely do anything for conditional execution branches, so it is most likely correct on 50% of the conditional execution branch instructions. In the loops, the method designed would be correct in all but the first and last execution of the loop, so 80% on loops.

The net effect is $(.4)(.5) + (.6)(.8) = .68$ or 68% of the time it is right.

6. Using the best cache and your dynamic branch predictor, calculate the average CPI and the performance of the computer in MIPS.

   I forgot to give you base CPI and the penalty to CPI for missing a branch. I wanted the base for all instructions to be 1 (ideal for piplined) and the penalty to be 3. Sorry about that.

   Average CPI is given by:

$$\begin{aligned}
\mathrm{CPI}_{\mathrm{avg}} &= \sum_i (\mathrm{CPI}_i \times \mathrm{frequency}_i) \\
&= \mathrm{CPI}_{\mathrm{Memory}} \times \mathrm{freq}_{\mathrm{Memory}} + \mathrm{CPI}_{\mathrm{Correct\ Branch}} \times \mathrm{freq}_{\mathrm{Correct\ Branch}} \\
&\quad + \mathrm{CPI}_{\mathrm{Incorrect\ Branch}} \times \mathrm{freq}_{\mathrm{Incorrect\ Branch}} + \mathrm{CPI}_{\mathrm{Other}} \times \mathrm{freq}_{\mathrm{Other}} \\
&= (1 + 2.208)(.2) + (1 + 1.104)(.3 \times .68) \\
&\quad + (1 + 1.104 + 3)(.3 \times .32) + (1 + 1.104)(.5) \\
&= 2.6128
\end{aligned}$$

   MIPS is given by:

$$\begin{aligned}
\mathrm{MIPS} &= \frac{\mathrm{CPU\ Clock\ Freq}}{\mathrm{CPI} \times 10^6} \\
&= \frac{10^9 \mathrm{Cycles/s}}{2.6128 \mathrm{Cycle/Million\ Inst} \times 10^6} \\
&\approx 383
\end{aligned}$$

A.2 One Command Computer

Consider a computer which has only one command, subtract and branch if negative (SBrN D, S1, S2, Jump). Which does:

```
D = S1 - S2
if D < 0 goto Jump
```

Since there is only one command there is no need to include the opcode in the machine language instruction. The system is to have 1K of memory divided into 256 words of 4 bytes each. Since memory requires 1 bytes to specify the address of a memory location the instructions will have four fields of 1 byte each:

Destination	Source 1	Source 2	Jump Address

1. Design a CPU that implements this.

   Sol:

   See Figure 1

2. Alter your design to make it a four stage pipeline with forwarding.

   Sol:

   See Figure 2. Note that the control to the forwarding MUXs can come from tag bits on the RAM (first idea) or comparators on the destination (better solution).

3. Design the control for the CPU (hardwired or microcoded)

   Sol:

   In this case, most of the control signals are already handled. All that remains undone is the load commands to the program counter and instruction register, and the read and write commands to memory. The ifetch loop has only four states so the resulting logic table is:

S_1S_0	S_1S_0	Rpc	Rs1/Rs2	Wd
00	01	1	0	0
01	10	0	1	0
10	11	0	0	0
11	00	0	0	1

   Next $S_1 = S_1' \cdot S_0 + S_1 \cdot S_0'$

   Next $S_0 = S_0'$

   $Rpc = S_1' \cdot S_0'$

   $Rs1Rs2 = S_1' \cdot S_0$

   $Wd = S_1 \cdot S_0$

4. Show the tag bits (with their size), data field, and address of a 2-way associative write-back cache that uses NLLRU for this machine that has 8 locations. How many total bits must be stored?

   Sol:

   Main Memory has 2^8 locations (n=8)

   Cache has 2^3 locations (m=3)

   Associativity is 2^1 (k=1)

   Each location in cache has a total of 42 bits

   (a) Address tag bits: n-(m-k) = 8-(3-1) = 6

   (b) Valid tag bit: 1

   (c) Dirty tag bit: 1

   (d) NLLRU tag bits: 2 (the associativity)

   (e) Data bits: 32

   The entire cache has $8 \times 42 = 336$ bits

5. Show the cache accesses and calculate the hit ratio for the following memory values, assuming execution begins at location 0 and terminates when location 5 is reached. If a location is not specified below, the contents are not important. All values are in hex.

Address	D	S1	S2	J
00	87	88	80	01
01	86	80	8B	02
02	87	87	86	03
03	01	01	83	04
04	82	82	81	01

Address	Data			
80	00	00	00	00
81	FF	FF	FF	FF
82	FF	FF	FF	FE
83	00	00	01	00

   Sol:

   (I have grouped my cache table so the associated portions of cache are on the same row.)

   Initial condition (hits=0, misses=0)

NLLRU	V	D	Address	Loc	Data	NLLRU	V	D	Address	Loc	Data
00	0	0	000000	000	0x00000000	00	0	0	000000	100	0x00000000
00	0	0	000000	001	0x00000000	00	0	0	000000	101	0x00000000
00	0	0	000000	010	0x00000000	00	0	0	000000	110	0x00000000
00	0	0	000000	011	0x00000000	00	0	0	000000	111	0x00000000

command=0x87888001 (hits=0, misses=3) (read in 0 then 88, then 80 overwrote 0)

NLLRU	V	D	Address	Loc	Data	NLLRU	V	D	Address	Loc	Data
01	1	0	100000	000	0x00000000	00	1	0	100010	100	0x????????
00	0	0	000000	001	0x00000000	00	0	0	000000	101	0x00000000
00	0	0	000000	010	0x00000000	00	0	0	000000	110	0x00000000
01	1	1	100001	011	0x????????	00	0	0	000000	111	0x00000000

command=0x86808B02 (hits=1, misses=5)

LRU	V	D	Address	Loc	Data	LRU	V	D	Address	Loc	Data
01	1	0	100000	000	0x00000000	00	1	0	100010	100	0x????????
01	1	0	000000	001	0x86808B02	00	0	0	000000	101	0x00000000
01	1	1	100001	010	0x????????	00	0	0	000000	110	0x00000000
00	1	1	100001	011	0x????????	10	1	0	100010	111	0x????????

command=0x87878603 (hits=3, misses=6)

LRU	V	D	Address	Loc	Data	LRU	V	D	Address	Loc	Data
01	1	0	100000	000	0x00000000	00	1	0	100010	100	0x????????
01	1	0	000000	001	0x86808B02	00	0	0	000000	101	0x00000000
01	1	1	100001	010	0x????????	00	1	0	000000	110	0x87878603
01	1	1	100001	011	0x????????	00	1	0	100010	111	0x????????

command=0x01018304 (hits=4, misses=8)

LRU	V	D	Address	Loc	Data	LRU	V	D	Address	Loc	Data
01	1	0	100000	000	0x00000000	00	1	0	100010	100	0x????????
01	1	1	000000	001	0x86808A02	00	0	0	000000	101	0x00000000
01	1	1	100001	010	0x????????	00	1	0	000000	110	0x87878603
01	1	1	100000	011	0x00000100	00	1	0	000000	111	0x01018304

command=0x82828101 (hits=4, misses=11) (NOTE: 82 is 0xFFFFFFFF at end of command then jumps to 01)

LRU	V	D	Address	Loc	Data	LRU	V	D	Address	Loc	Data
00	1	0	100000	000	0x00000000	10	1	0	000001	100	0x82828101
00	1	1	000000	001	0x86808A02	10	1	0	100000	101	0xFFFFFFFF
00	1	1	100001	010	0x????????	10	1	0	100000	110	0xFFFFFFFF
01	1	1	100000	011	0x00000100	00	1	0	000000	111	0x01018304

command=0x0x86808A02 (hits=6, misses=12)

LRU	V	D	Address	Loc	Data	LRU	V	D	Address	Loc	Data
00	1	0	100000	000	0x00000000	10	1	0	000001	100	0x82828101
01	1	1	000000	001	0x86808A02	00	1	0	100000	101	0xFFFFFFFF
00	1	1	100001	010	0x????????	10	1	0	100010	110	0x????????
01	1	1	100000	011	0x00000100	00	1	0	000000	111	0x01018304

command=0x0x87878603 (hits=7, misses=14)

LRU	V	D	Address	Loc	Data	LRU	V	D	Address	Loc	Data
00	1	0	100000	000	0x00000000	10	1	0	000001	100	0x82828101
01	1	1	000000	001	0x86808A02	00	1	0	100000	101	0xFFFFFFFF
01	1	0	100001	010	0x87878603	00	1	0	100010	110	0x????????
00	1	1	100000	011	0x00000100	10	1	1	100001	111	0x????????

command=0x0x01018304 (hits=8, misses=16)

LRU	V	D	Address	Loc	Data	LRU	V	D	Address	Loc	Data
00	1	0	100000	000	0x00000000	10	1	0	000001	100	0x82828101
01	1	1	000000	001	0x86808902	00	1	0	100000	101	0xFFFFFFFF
01	1	0	100001	010	0x87878603	00	1	0	100010	110	0x????????
01	1	0	000000	011	0x01018304	10	1	0	100003	111	0x00000100

command=0x0x82828101 (hits=10, misses=17) (loaded 82 as 0xFFFFFFFF then -(-1) to get 0 which ends)

LRU	V	D	Address	Loc	Data	LRU	V	D	Address	Loc	Data
00	1	0	100000	000	0x00000000	10	1	0	000001	100	0x82828101
01	1	1	000000	001	0x86808902	00	1	0	100000	101	0xFFFFFFFF
00	1	0	100001	010	0x87878603	10	1	1	100000	110	0x00000000
01	1	0	000000	011	0x01018304	10	1	0	100003	111	0x00000100

6. Assuming the cache has an access time of 4ns and the memory has an access time of 60ns, calculate the effective access time of the memory.

Sol:

$hr = \frac{\text{hit}}{\text{hit+miss}} = \frac{10}{27} \approx .37$

$mr = 1 - hr \approx 1 - .37 = .63$

$T_{\text{eff}} = hr \times T_{\text{cache}} + mr \times T_{\text{RAM}} \approx .37 \times 4 + .63 \times 60 \approx 39ns$

A.3 Multiple Issue Machine

You have a 1.5 GHz computer which can issue 2 instructions per cycle and a dynamic branch predictor that reduces the branch penalty from 4 cycles to 1 cycle, 90% of the time. Branch instructions are 15% of all instructions, loads are 20%, and stores are 5%.

The cache is split into 4k instruction cache and 4k data cache. The cache takes 2 ns to access. The instruction cache has a block size of 2 words, has an associativity of 4, and a miss rate of 2%. The data cache has a block size of 4 words, an associativity of 2, is write-back, is not write-allocate, has a read miss rate of 5%, a write miss rate of 2%, and 10% of the blocks are dirty.

The RAM is 8MB takes 50ns to access and can burst write subsequent accesses at 10ns.

1. How many cycles on average is the branch penalty?

$$\begin{aligned} Penalty_{branch} &= f_{\text{pred. correct}} Cost_{\text{pred. correct}} + f_{\text{pred. error}} Cost_{\text{pred. error}} \\ &= .9 \times 1 + .1 \times 4 \\ &= 1.3 \end{aligned}$$

2. How long does an instruction read miss take?

   Two words have to be loaded on a miss, which takes 70ns.

3. How long does a data read miss take?

   Four words have to be loaded on a miss, which takes 90ns. Now 10% of the time we also have to write four words, which takes the same as a read thus we have: $(1 + .1) \times 90ns = 99ns$

4. How long does a data write miss take?

   On a write miss, four words have to be written, which takes 90ns.

5. What is the effective access time for instruction loads?

$$\begin{aligned} T_{Inst} &= 2ns + .02 \times 70ns \\ &= 3.4ns \end{aligned}$$

6. What is the effective access time for data reads?

$$\begin{aligned} T_{read} &= 2ns + .05 \times 99ns \\ &= 6.95ns \end{aligned}$$

7. What is the effective access time for data writes?

$$\begin{aligned} T_{write} &= 2ns + .02 \times 90ns \\ &= 3.8ns \end{aligned}$$

8. What is the CPI of this machine?

$$\begin{aligned} CPI &= \frac{(1 + f_{branch} Penalty_{branch} + Penalty_{inst} + f_{read} \times Penalty_{read} + f_{write} \times Penalty_{write})}{\#\text{ inst per cycle}} \\ &= \frac{(1 + f_{branch} Penalty_{branch} + Clock_{rate}(T_{inst} + f_{read} \times T_{read} + f_{write} \times T_{write}))}{\#\text{ inst per cycle}} \\ &= \frac{1 + .15 \times 1.3 + 1.5GHz(3.4ns + .2 \times 6.95ns + .05 \times 3.8ns)}{2} \\ &= 1.0830625 \end{aligned}$$

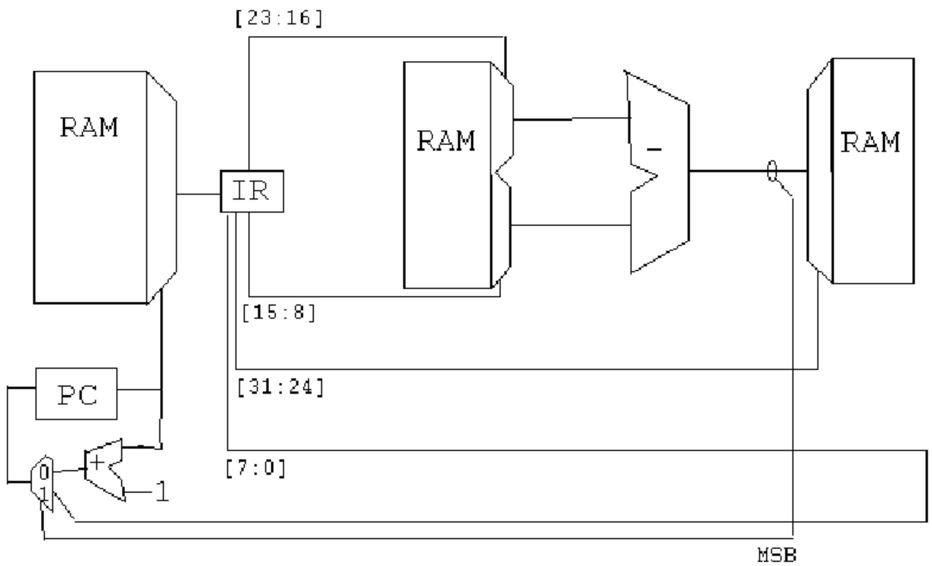


Figure 1

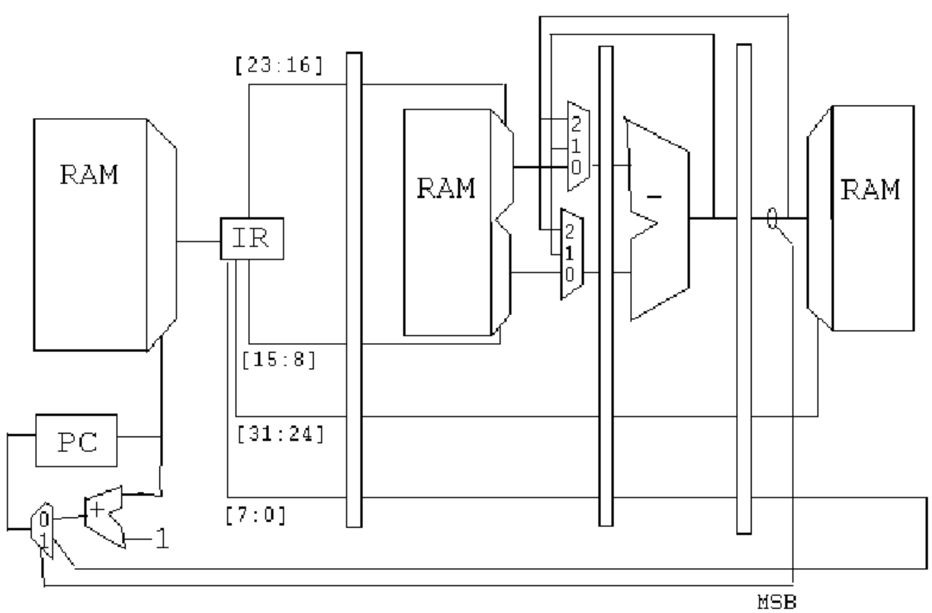


Figure 2

Figure A.1: One Command Computer

Appendix B

Encryption

B.1 Modular Arithmetic

B.1.1 Congruence

We say a is congruent to b modulus n when $a - b$ is divisible by n. In mathematical notation, we write $a \equiv b \pmod{n} \quad \Leftrightarrow \quad a - b = kn$ for some integer k. Several important properties of congruence are

1. $a \equiv a \pmod{n}$
2. $a \equiv b \pmod{n} \quad \Rightarrow \quad b \equiv a \pmod{n}$
3. $\{\{a \equiv b \pmod{n}\} \cdot \{b \equiv c \pmod{n}\}\} \quad \Rightarrow \quad a \equiv c \pmod{n}$

Example 21

$$\begin{aligned} 8 &\equiv 29 \pmod{7} \\ 8 - 29 &= -21 \\ &= (-3)7 \end{aligned}$$

$$\begin{aligned} 9 &\equiv -15 \pmod{6} \\ 9 - (-15) &= 24 \\ &= (4)6 \end{aligned}$$

B.1.2 Modulus

Invariably confusion happens with integer division, modulus, and remainder involving negative numbers. The problem arises in the basic definition. For a dividend, $a \in \mathbb{Z}$ and a divisor, $b \in \mathbb{Z}$, the quotient, q and remainder r must satisfy

1. $\{r, q\} \in \mathbb{Z}$,
2. $a = b * q + r$,
3. $|r| < |d|$.

The problem comes with the last requirement, because many choices can be made. The three most justifiable definitions are below[1]

1. Truncate division preserves the magnitudes of the quotient and remainder, independent of the signs of the dividend and divisor. This forces the remainder to have the same sign as the dividend.

2. Floor division forces the remainder to have the same sign as the divisor.

3. Euclidean division defines $r \geq 0$ and thus ensures $b \times q \leq a$.

Each is defensible.

Truncate

Remainder's definition is based off the definition of integer division. Integer division, a/b, is defined for positive a and b to be the number q such that

1. $b \times q \leq a$,

2. $b \times (q+1) \geq a$.

When negative numbers are allowed the following requirement is added

3 $(-a)/b = a/(-b) = -(a/b)$,

still for a and b positive. One could summarize this as:

$$c/d \quad = \quad \mathrm{sgn}(c)\mathrm{sgn}(d)(|c|/|d|)$$

Given we now have quotient or integer division defined we can then define remainder such that

$$\begin{aligned} a &= a/b + a rem b \\ a rem b &= a - a/b. \end{aligned}$$

Note that the sign of the remainder is the same as the

Example 22 *Consider the following:*

$$\begin{aligned} 5/2 &= 2 & 5 rem 2 &= 1 \\ (-5)/2 &= -2 & (-5) rem 2 &= -1 \\ 5/(-2) &= -2 & 5 rem (-2) &= 1 \\ (-5)/(-2) &= 2 & (-5) rem (-2) &= -1 \end{aligned}$$

B.1.3 Addition

$\{\{a \equiv b \pmod{n}\} \cdot \{c \equiv d \pmod{n}\}\} \quad \Rightarrow \quad a + c \equiv b + d \pmod{n}$

[1]other definitions exist such as ceiling division and rounding division, but they do not correspond to the what most people think of division for positive numbers. Note, from the requirements nothing says $5/2 = 3r - 1$ but this is hardly what most people would think of, and thus would probably not be programmed very well.

B.1.4 Additive Inverse

$$\begin{aligned} a + \bar{a} &\equiv 0 \pmod{n} \\ a + \bar{a} &= kn, \qquad k \in \mathbb{Z} \\ \bar{a} &= kn - a, \qquad k \in \mathbb{Z} \end{aligned}$$

Example 23 *Find the additive inverse(s) of 3 mod 7.*

$$\begin{aligned} \bar{a} &= kn - a, \qquad k \in \mathbb{Z} \\ &= 7k - 3, \qquad k \in \mathbb{Z} \end{aligned}$$

k	$\bar{a}$	$(3+\bar{a}) \mod 7$
1	*4*	$(3+4) \mod 7 = 0$
2	*11*	$(3+11) \mod 7 = 0$
3	*18*	$(3+18) \mod 7 = 0$
4	*25*	$(3+25) \mod 7 = 0$
$\vdots$	$\vdots$	

B.1.5 Multiplication

$$\{\{a \equiv b \pmod{n}\} \cdot \{c \equiv d \pmod{n}\}\} \quad \Rightarrow \quad ac \equiv bd \pmod{n}$$

B.1.6 Multiplicative Inverse

$$\begin{aligned} a\bar{a} &\equiv 1 \pmod{n} \\ a\bar{a} &= 1 + kn, \qquad k \in \mathbb{Z} \\ \bar{a} &= \frac{1+kn}{a}, \qquad k \in \mathbb{Z} \end{aligned}$$

Let $k_1 + ak_2 = k$ for k_1 and k_2 positive integers.

$$\begin{aligned} \bar{a} &= \frac{1+kn}{a}, \qquad k \in \mathbb{Z} \\ &= \frac{1+k_1n+ak_2n}{a}, \qquad k_1, k_2 \in \mathbb{Z}^+ \\ &= \frac{1+k_1n}{a} + k_2n, \qquad k_1, k_2 \in \mathbb{Z}^+ \end{aligned}$$

We need a to divide $1 + k_1n$, which means it divides with no remainder (aka divides evenly). Consider what would happen if $gcd(a, n) = a_1 > 1$, thus $a = a_1a_2$ and $n = a_1n_2$ for a_1, a_2, and n_2 positive integers. If a_1 is a factor of n then it is also a factor of k_1n If a_1 is a factor of k_1n then it cannot be a factor of $k_1n + 1$ (it evenly divides k_1n and $k_1n + k_1$ but nothing in between).

Now assume $gcd(a, n) = 1$. For a to divide $1 + k_1n$ implies $ak_3 = 1 + k_1n$ for some positive integer k_3.

Example 24 *Find the multiplicative inverse(s) of 3 mod 7.*

$$\begin{aligned} \bar{a} &= \frac{1+kn}{a}, \qquad k \in \mathbb{Z} \\ &= \frac{1+7k}{3}, \qquad k \in \mathbb{Z} \end{aligned}$$

k	$\bar{a}$	$(3+\bar{a}) \mod 7$
1	$\frac{8}{3}$ *no*	
2	$\frac{15}{3} = 5$	$(3 \times 5) \mod 7 = 1$
3	$\frac{22}{3}$ *no*	
4	$\frac{29}{3}$ *no*	
5	$\frac{36}{3} = 12$	$(3 \times 12) \mod 7 = 1$
$\vdots$	$\vdots$	

B.2 Affine Encryption Program

Affine encryption is one of the simplest methods for doing encryption. Let P_i be the i^{th} character in the plain text message, and let C_i be the corresponding encoded character. Let there be n possible characters to encode, then the basic idea is to pick two numbers (a, b) to encode a message such that $\gcd(a, n) = 1$ (so a has an inverse). No requirement on b is needed if your modulus function has been encoded correctly. The encoded character can then be found by

$$a \times P_i + b \quad = \quad C_i \mod n.$$

Note that the " $\mod n$" at the end says the equation holds in $\mathbb{Z}_n$, the set of integers mod n with appropriately defined arithmetic.

To decrypt the message, the equation

$$\bar{a} \times (C_i + d) \quad = \quad P_i \mod n$$

is used. The term $\bar{a}$ is the inverse of a in $\mathbb{Z}_n$, which is found by solving

$$\begin{array}{rcl} a \times \bar{a} & = & 1 \mod n \\ & \textbf{or} & \\ a \times \bar{a} & = & m \times n + 1. \end{array}$$

Note that m is any whole number. The term d is the additive inverse of b in $\mathbb{Z}_n$, which is found by solving

$$d = n - (b \mod n).$$

We can summarize this by saying an affine cipher is an encryption technique that encodes using three integers: a, b, and n. If *plain* is the character to be encoded (with 'A'=0 and 'Z'=25) then $code = (a * plain + b) \mod n$. Decoding is also done using three integers: c, d, and n. If *code* is the character to be encoded (with 'A'=0 and 'Z'=25) then $plain = (c * (code + d)) \mod n$. The requirements on (a, b, c, d, n) are:

- $\gcd(a, n) = 1$
- $(ac) \mod n = 1$
- $(b + d) \mod n = 0$

Below is C code to implement a particular case of affine cyphers.

```
char affine_encode(char plain){
    // affine codes capital letter in plain using a=5, b=12 thus this is modulo 26
    int iCode, iPlain, a=3,b=0;
```

```
    // convert char to integer and shift so A=0
    iPlain=int(plain)-65;

    // do the encoding
    iCode = (a*iPlain+b)%26;

    // return the result as a char
    return char(iCode+65);
}

char affine_decode(char code){
    // affine decodes capital letter in plain using c=21, d=8 thus this is modulo 26
    int iCode, iPlain, c=9, d=0;

    // convert char to integer and shift so A=0
    iCode=int(code)-65;

    // do the decoding
    iPlain = (c*(iCode+d))%26;

    // return the result as a char
    return char(iPlain+65);
}
```