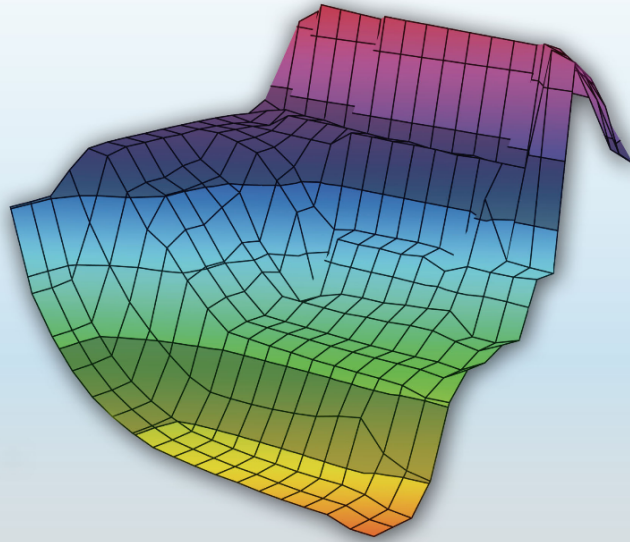


Second Edition

COMPUTER SYSTEMS

A Programmer's Perspective



Bryant • O'Hallaron

SYLLABUS

INF2270 - spring 2013

Contents

I	Computer architecture	4
1	CONVERSION OF NUMBERS	4
1.1	Hex to bin	4
1.2	Dec to bin	4
1.3	Oct to bin	4
1.4	Hex to dec	5
2	TWO'S COMPLEMENT (signed)	5
2.1	Negative numbers	5
2.2	Bin to dec	5
3	BINARY MATHEMATICAL OPERATIONS	6
3.1	Binary addition	6
3.2	Binary subtraction	6
3.3	Binary multiplication	6
3.4	Binary division	7
4	BOOLEAN ALGEBRA	7
4.1	Gates	7
4.2	Truth table	8
4.3	Sum of products (SOP)	8
4.4	Minterm	8
4.5	Product of sums (POS)	8
4.6	Maxterm	8
4.7	Special implementation	9
5	SIMPLIFICATION OF BOOLEAN EXPRESSIONS	10
6	KARNAUGH DIAGRAM	10
7	COMBINATIONAL LOGIC CIRCUITS	11
7.1	Decoder	11
7.1.1	Other variants	12
7.2	Encoder	13
7.2.1	Other variants	14
7.3	Multiplexer (MUX)	15
7.4	Demultiplexer (DEMUX)	15
7.5	Adders (half/full)	16
7.5.1	Half Adder	16
7.5.2	Full Adder	17
8	SEQUENTIAL LOGIC CIRCUITS	18
8.1	Synchronous Flip-flops	18
8.1.1	D-Latch	18
8.1.2	D-Flip-Flop	19
8.1.2.1	Practical examples	20
8.1.3	JK-Flip-Flop	21

8.1.4	T-Flip-Flop	22
8.2	Asynchronous Latches	23
8.2.1	Gated D-latch/transparent latch	23
8.2.2	SR-Latch	24
8.3	Finite State Machine (FSM)	25
8.3.1	State Transition Graphs	25
8.3.1.1	Example	26
8.3.2	Moore FSM	27
8.3.3	Mealy FSM	27
8.3.4	Synchronous FSM	27
8.3.5	Asynchronous FSM	27
8.4	Registers	28
8.5	Standard Sequential Logic Circuits	29
8.5.1	Counter	29
8.5.2	Shift register	30
9	VON NEUMAN ARCHITECTURE	31
9.1	Data Path and Memory Bus	31
9.2	Arithmetic and Logical Unit (ALU)	31
9.3	Memory	33
9.3.1	Terminology	33
9.3.2	Typical signals of a RAM	33
9.3.3	Static Random Access Memory (SRAM)	34
9.3.4	Dynamic Random Access Memory (DRAM)	35
9.4	Control Unit (CU)	35
9.4.1	Register Transfer Language (RTL)	35
9.4.2	Execution of Instructions	36
9.4.3	Microarchitecture	37
9.4.4	Complex and reduced instruction sets (CISC/RISC)	37
9.5	Input/Output	38
9.5.1	Modes of transfer	38
10	OPTIMIZING HARDWARE PERFORMANCE	39
10.1	Memory Hierarchy	39
10.1.1	Applications	40
10.1.2	Storage Capacity	40
10.1.3	access speed	40
10.1.4	Cache	40
10.1.4.1	Cache hit	41
10.1.4.2	Cache miss	42
10.1.5	Cache mapping strategies	42
10.1.6	Cache coherency	44
10.1.7	Cache block replacement strategies	44
10.1.8	Cache Architecture	45
10.1.9	Virtual Memory	46
10.2	Speed up the single-cycle design	47
10.2.1	Multi-cycle	47
10.2.2	Pipelining	48
10.2.2.1	Speed-up	49
10.2.2.2	Pipelining Hazards	49

10.2.2.2.1	Resource Hazards	50
10.2.2.2.2	Data Hazards	50
10.2.2.2.3	Control Hazards	51
10.3	Superscalar CPU	51
 II Low-level programming		52
 11 Assembly		52
11.1	Registers	52
11.2	Instructions	53
11.3	ASCII-table	56
 12 C		57
12.1	Inline-code	57
12.1.1	Assembly	57
12.1.2	Parameters	57

Part I

Computer architecture

1 CONVERSION OF NUMBERS

1.1 Hex to bin

Each hexadecimal consists of four binary numbers.

Example:

$(123)_{HEX}$
1 \rightarrow 0001
2 \rightarrow 0010
3 \rightarrow 0011
 $\rightarrow (0001\ 0010\ 0011)_{BIN}$

1.2 Dec to bin

Divide on two, keep remainder.

Example:

$(123)_{DEC}$
123 : 2 \rightarrow 1 LSB
61 : 2 \rightarrow 1
30 : 2 \rightarrow 0
15 : 2 \rightarrow 1
7 : 2 \rightarrow 1
3 : 2 \rightarrow 1
1 : 2 \rightarrow 1
 $\rightarrow (0111\ 1011)_{BIN}$

1.3 Oct to bin

Each octadecimal consists of three binary numbers.

Example:

$(123)_{OCT}$
1 \rightarrow 001
2 \rightarrow 010
3 \rightarrow 011
 $\rightarrow (001\ 010\ 011)_{BIN}$

1.4 Hex to dec

Like every conversion to decimal, we do the following;

Example:

$$\begin{aligned}
 (123,123)_{HEX} \\
 1 &\rightarrow 1 \cdot 16^2 \rightarrow 256 \\
 2 &\rightarrow 2 \cdot 16^1 \rightarrow 32 \\
 3 &\rightarrow 3 \cdot 16^0 \rightarrow 3 \\
 , \\
 1 &\rightarrow 1 \cdot 16^{-1} \rightarrow \frac{1}{16^1} \rightarrow 0.0625 \\
 2 &\rightarrow 2 \cdot 16^{-2} \rightarrow \frac{2}{16^2} \rightarrow 0.0078125 \\
 3 &\rightarrow 3 \cdot 16^{-3} \rightarrow \frac{3}{16^3} \rightarrow 0.000732421875 \\
 &\rightarrow 256 + 32 + 3 + 0.0625 + 0.0078125 + 0.000732421875 \\
 &\rightarrow (291.0710449)_{DEC}
 \end{aligned}$$

General;

$$(\dots a_2 a_1 a_0, a_{-1} a_{-2} \dots)_r = \dots + a_2 \cdot r^2 + a_1 \cdot r^1 + a_0 \cdot r^0 + a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} + \dots$$

2 TWO'S COMPLEMENT (signed)

2.1 Negative numbers

To convert to negative numbers, you just have to invert the binary number and add one.

Example:

-5

$$5 = 0101$$

$$\begin{array}{rcl}
 & 1010 & \text{(inverted)} \\
 + & 0001 & \text{(add one)} \\
 \hline
 = & 1011 & \text{(-5)}
 \end{array}$$

If you want more bits in your answer, add ones (instead of zeros)

2.2 Bin to dec

To convert from binary to decimal, we'll use same approach;

$$11011$$

$$\begin{array}{rcl}
 & 00100 & \text{(inverted)} \\
 + & 00001 & \text{(add one)} \\
 \hline
 = & 00101 & \text{(\cdot -1)}
 \end{array}$$

$$\begin{aligned}
 &\rightarrow -(1 \cdot 2^2 + 1 \cdot 2^0) \\
 &\rightarrow -5
 \end{aligned}$$

3 BINARY MATHEMATICAL OPERATIONS

3.1 Binary addition

$$\begin{array}{r}
 \text{1 1 1} \\
 00101 \\
 + 01101 \\
 \hline
 = 10010
 \end{array}
 \qquad
 \begin{array}{r}
 05 \\
 + 13 \\
 \hline
 = 18
 \end{array}$$

Figure 1: Binary addition

3.2 Binary subtraction

$ \begin{array}{r} 0011 \\ + 1000 \\ \hline = 1011 \end{array} $ <p>Negative</p>	$ \begin{array}{r} 3 \\ + -8 \\ \hline = -5 \end{array} $	$ \begin{array}{r} \text{1 1} \\ 0110 \\ + 1110 \\ \hline = (1)0100 \end{array} $ <p>Remove</p>	$ \begin{array}{r} 6 \\ + -2 \\ \hline = 4 \end{array} $
---	--	---	---

Figure 2: Binary subtraction

3.3 Binary multiplication

Multiplication with a factor of two of a binary number is simply achieved by shifting the individual bits by one position to the left and inserting a '0' into the LSB (rightmost bit).

Or else:

$$\begin{array}{r}
 101 \\
 \times 10 \\
 \hline
 000 \\
 + 101 \\
 \hline
 = 1010
 \end{array}$$

Figure 3: Binary multiplication

3.4 Binary division

A division with a factor of 2 is a shift of all the bits by one position to the right. Note that if the MSB (leftmost bit) is filled in with a copy of its state before the shift (This is known as arithmetic right shift), this works for both unsigned and signed (two's complement) binary numbers, but note that the result is rounded towards $-\infty$ and not towards zero, e.g., right-shifting -3 results in -2.

Or else:

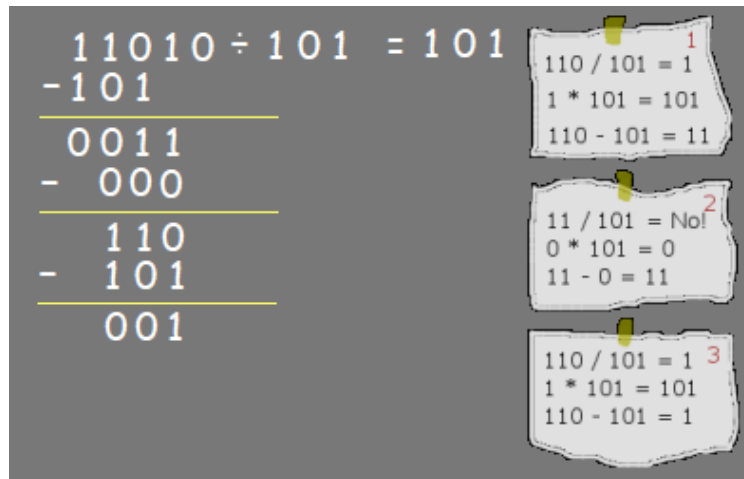


Figure 4: Binary division

4 BOOLEAN ALGEBRA

4.1 Gates

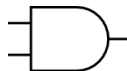


Figure 5: AND, True if all input-signals are true.

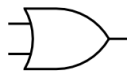


Figure 6: OR, True if one of input-signals are true.

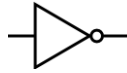


Figure 7: NOT, Inverts signal.



Figure 8: XOR, True if odd number of true input-signals.

4.2 Truth table

abc	AND	OR	NAND	NOR	XOR
000	0	0	1	1	0
001	0	1	1	0	1
010	0	1	1	0	1
011	0	1	1	0	0
100	0	1	1	0	1
101	0	1	1	0	0
110	0	1	1	0	0
111	1	1	0	0	1

4.3 Sum of products (SOP)

- Only OR (sum) operations at the "outermost" level
- Each term that is summed must be a product of literals

Example:

$$f(x, y, z) = y' + x'yz' + xz$$

4.4 Minterm

A minterm is a special product of literals, in which each input variable appears exactly once. A function with n variables has 2^n minterms (since each variable can appear complemented or not).

4.5 Product of sums (POS)

- Only AND (product) operations at the "outermost" level
- Each term must be a sum of literals

Example:

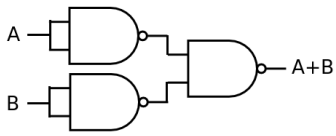
$$f(x, y, z) = y'(x' + y + z')(x + z)$$

4.6 Maxterm

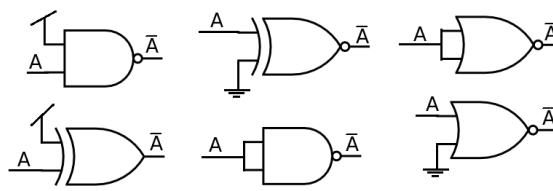
A maxterm is a sum of literals, in which each input variable appears exactly once. A function with n variables has 2^n maxterms.

4.7 Special implementation

OR



INVERTER

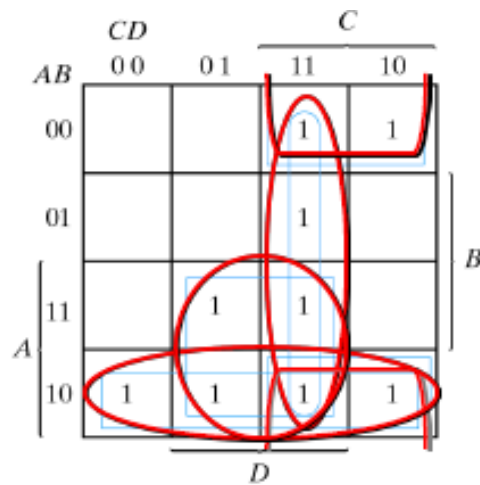


5 SIMPLIFICATION OF BOOLEAN EXPRESSIONS

P2	$x + 0 = x$	P2	$x \cdot 1 = x$
P5	$x + x' = 1$	P5	$xx' = 0$
P3	$x + y = y + x$	P3	$xy = yx$
	$x + (y + z) = (x + y) + z$		$x(yz) = (xy)z$
P4	$x(y + z) = xy + xz$	P4	$x + (yz) = (x + y)(x + z)$
T1a	$x + x = x$	T1b	$x \cdot x = x$
T2a	$x + 1 = 1$	T2b	$x \cdot 0 = 0$
	$x + xy = x$		$x(x + y) = x$
DeMorgans	$(x + y)' = x'y'$	DeMorgans	$(xy)' = x' + y'$

Figure 9: Theorems and postulates

6 KARNAUGH DIAGRAM

Figure 10: $F = AD + AB' + CD + B'C$

7 COMBINATIONAL LOGIC CIRCUITS

Combinational logic circuits are logic/digital circuits composed of feed-forward networks of logic gates with no memory that can be described by Boolean functions.

7.1 Decoder

- takes a binary word, and returns all minterms.

It decodes n inputs to 2^n outputs.

Example:

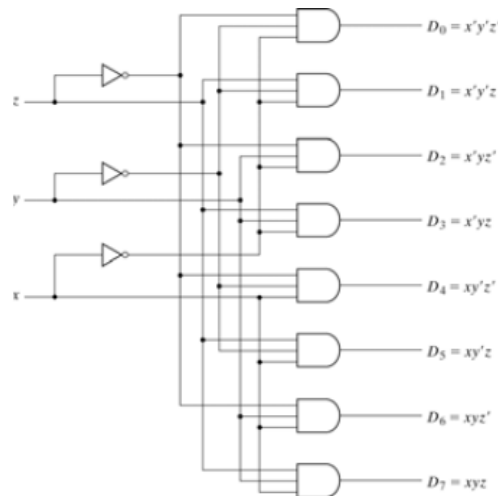


Figure 11: 3 bit in / 8 bit out

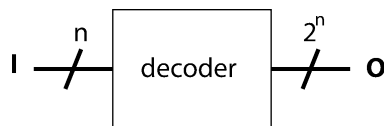


Figure 12: Decoder symbol

input			output							
x	y	z	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figure 13: Truth table for 3 bit decoder

7.1.1 Other variants

- Enable input
 Enable active \rightarrow normal operation.
 Enable inactive \rightarrow all outputs disabled.

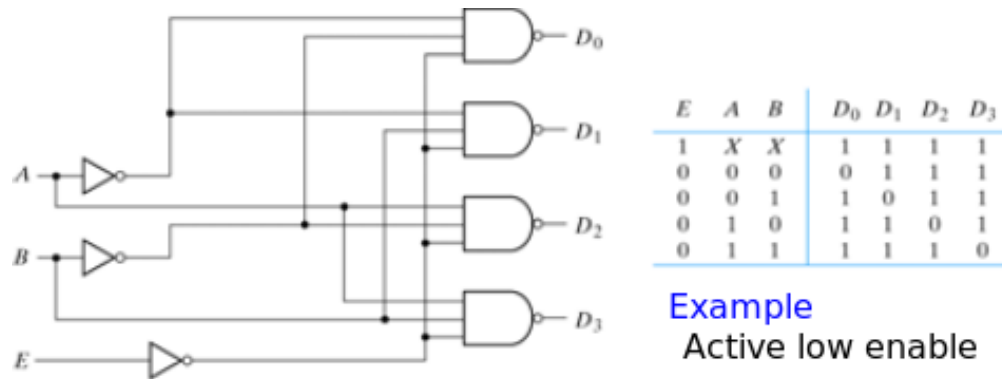


Figure 14: NAND logic: inverse output

- Parallel
 Example:
 Makes an 4x16 decoder from two 3x8 decoders with enable input.

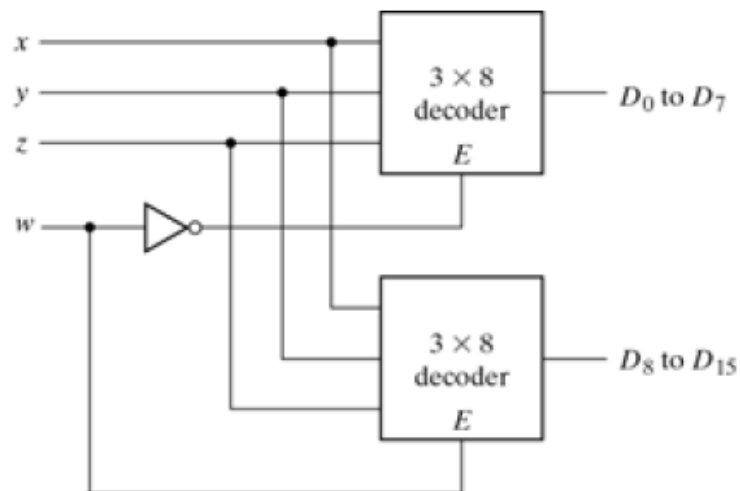


Figure 15: 4x16 decoder

7.2 Encoder

- inverse function of a decoder.

It decodes 2^n inputs to n outputs.

Example:

$$x = D_4 + D_5 + D_6 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$z = D_1 + D_3 + D_5 + D_7$$

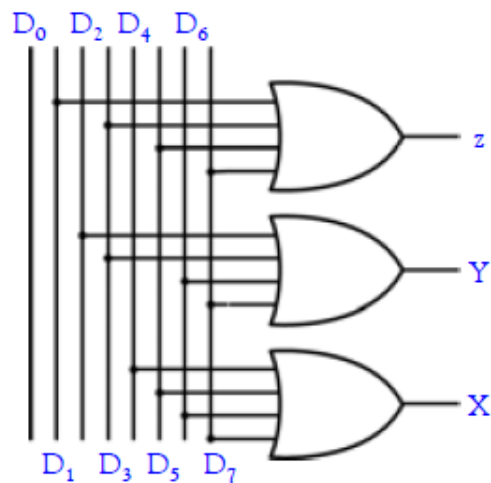


Figure 16: 8 bit in / 3 bit out

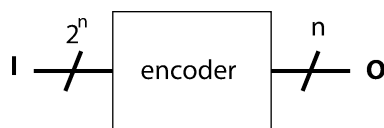


Figure 17: Encoder symbol

input								output		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Figure 18: Truth table

7.2.1 Other variants

- Priority encoder

Problem: what if we get multiple high input-signals?

Solution: Priority encoder

If multiple high input-signals, the signal with the highest index applies.

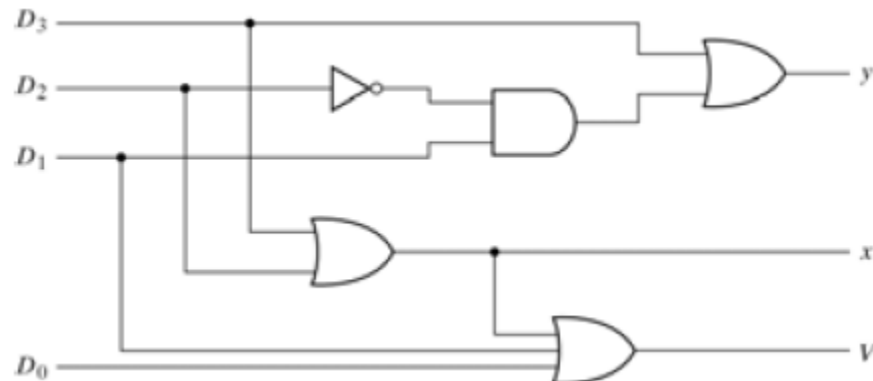


Figure 19: Priority encoder, "V" (valid) gives that at least one signal is high

input								output		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
x	1	0	0	0	0	0	0	0	0	1
x	x	1	0	0	0	0	0	0	1	0
x	x	x	1	0	0	0	0	0	1	1
x	x	x	x	1	0	0	0	1	0	0
x	x	x	x	x	1	0	0	1	0	1
x	x	x	x	x	x	1	0	1	1	0
x	x	x	x	x	x	x	1	1	1	1

Figure 20: Priority encoder, truth table

7.3 Multiplexer (MUX)

A *multiplexer* routes one of 2^n input signals as defined by the binary control number S to the output.

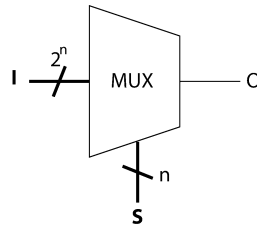


Figure 21: Multiplexer symbol

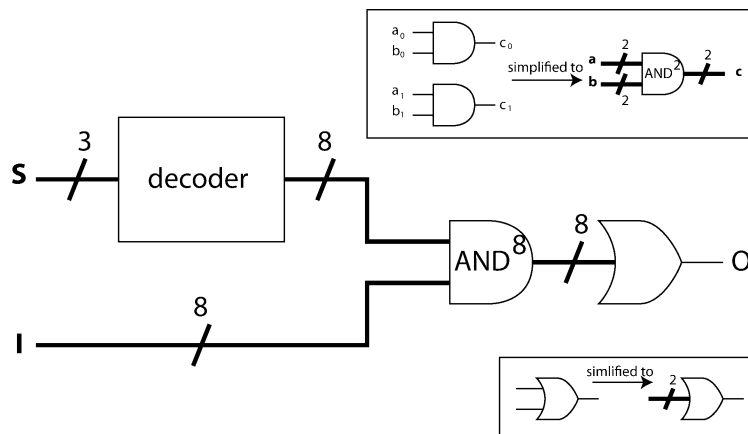


Figure 22: Possible multiplexer implementation

7.4 Demultiplexer (DEMUX)

A *demultiplexer* performs the inverse function of a multiplexer, routing one input signal to one of 2^n outputs as defined by the binary control number S .

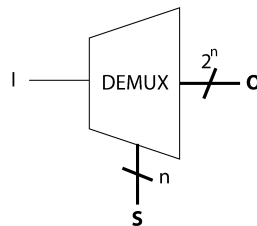


Figure 23: Demultiplexer symbol

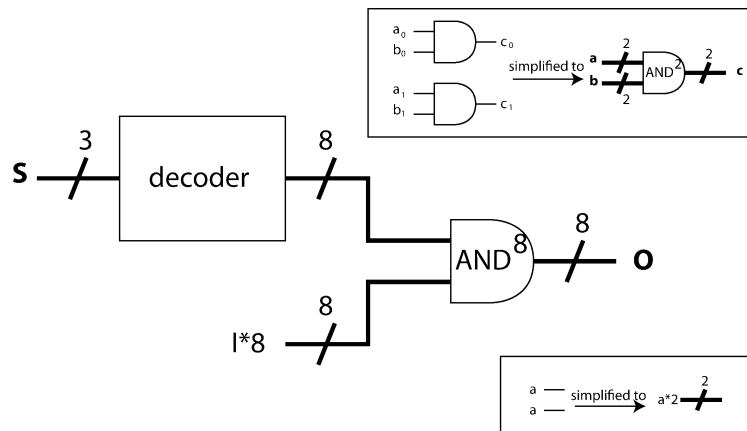


Figure 24: Possible demultiplexer implementation

7.5 Adders (half/full)

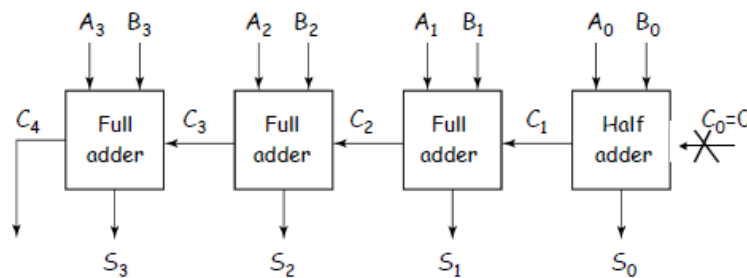


Figure 25: Adder system

7.5.1 Half Adder

A *half adder* can add two 1-bit binary numbers, and output sum (S) and carry (C)

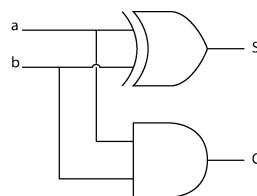


Figure 26: Implementation of half adder

a	b	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 27: Truth table, half adder

7.5.2 Full Adder

A *full adder* can add two 1-bit binary numbers and carry input. This gives an output sum (S) and carry (C)

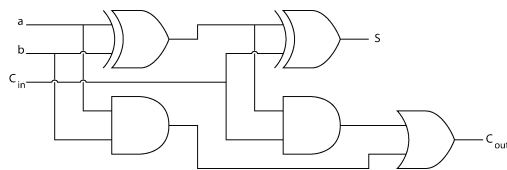


Figure 28: Implementation of full adder

C _{in}	a	b	S	C _{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 29: Truth table, full adder

8 SEQUENTIAL LOGIC CIRCUITS

Sequential logic circuits go beyond the concept of a Boolean function: they contain internal memory elements and their output will also depend on those internal states, i.e., on the input history and not just the momentary input.

8.1 Synchronous Flip-flops

Synchronous digital circuits have a dedicated input signal called clock (CLK). State changes of synchronous circuits will only occur in synchrony with a change of this clock signal, i.e., either at the rising or falling edge of the clock (figure 30). A clock signal toggles back and forth between 0 and 1 with a regular frequency, the inverse of which is the clock period or clock cycle. In circuit symbols the clock signal is often marked with a triangle just inside of the clock pin. If the pin is connected to the symbol with a circle in addition the falling edge of the clock will be used for synchronization, otherwise it's the rising edge.

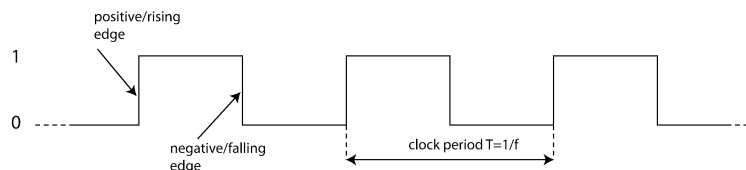


Figure 30: Clock signal

8.1.1 D-Latch

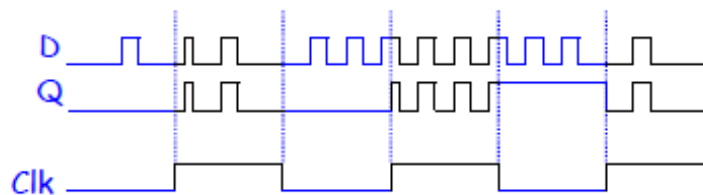


Figure 31: Positive-edge-triggered D-Latch

The *D-Latch* has the following characteristics:

- Clk=1: D-Latch is transparent
- Clk=0: Locked output

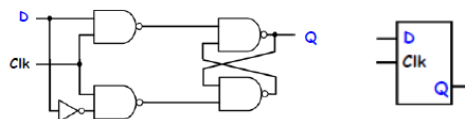


Figure 32: D-Latch

8.1.2 D-Flip-Flop

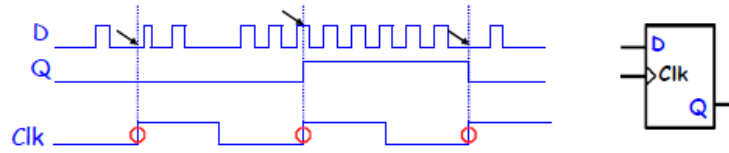


Figure 33: Positive-edge-triggered D-Flip-Flop

The *Postive-edge-triggered D-Flip-Flop* samples the value on D for rising clock signal.

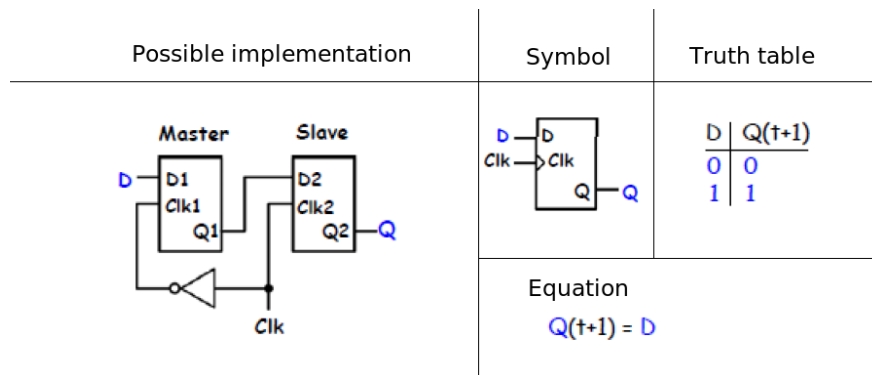


Figure 34: D-Flip-Flop

The *D-Flip-Flop* has the following characteristics:

- Clk=0: First D-Latch (master) is transparent
- Clk=1: Last D-Latch (slave) is transparent

A more compact version;

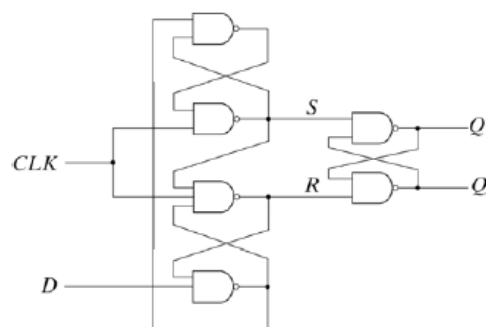


Figure 35: Compact implementation of D-Flip-Flop

8.1.2.1 Practical examples

- Ripple-carry adder

A ripple-carry adder will, in a short window of time, give wrong output.

If we use D-Flip-Flops to control the signal flow, we can prevent this.

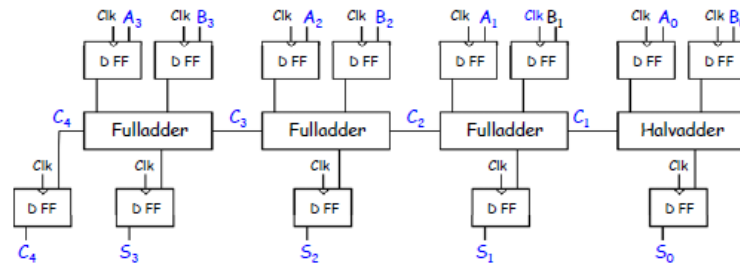


Figure 36: Controlled ripple-carry adder

- Serial adder

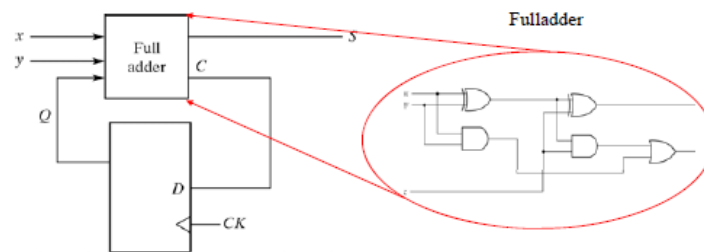


Figure 37: Serial adder

8.1.3 JK-Flip-Flop

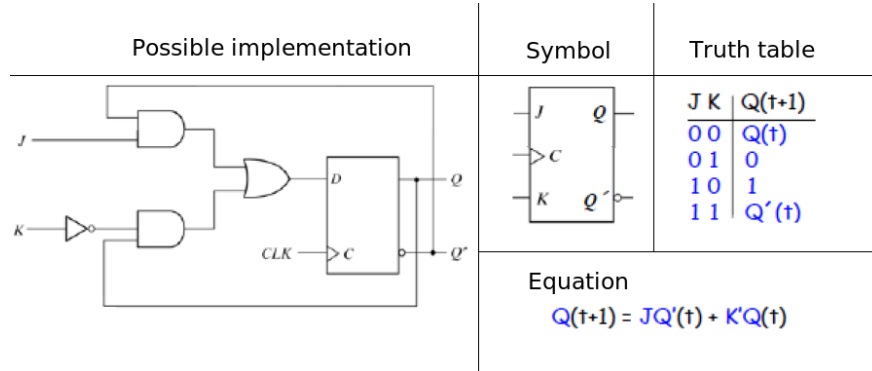


Figure 38: JK-Flip-Flop

The *JK-Flip-Flop* has the following characteristics:

- J=0, K=0: Locked output
- J=0, K=1: Reset output to "0"
- J=1, K=0: Set output to "1"
- J=1, K=1: Inverts output; $Q \rightarrow \overline{Q}$

The output can only change values when the clock signal 'C' rises.

The JK flip-flop is the mosr general flip-flop.

8.1.4 T-Flip-Flop

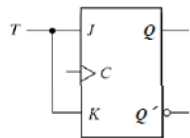
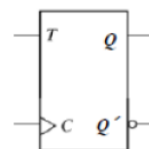
Possible implementation	Symbol	Truth table						
		<table><tr><th>T</th><th>Q(t+1)</th></tr><tr><td>0</td><td>Q(t)</td></tr><tr><td>1</td><td>Q'(t)</td></tr></table>	T	Q(t+1)	0	Q(t)	1	Q'(t)
T	Q(t+1)							
0	Q(t)							
1	Q'(t)							
<p>Equation</p> $Q(t+1) = T \oplus Q(t)$								

Figure 39: T-Flip-Flop

The *T-Flip-Flop* has the following characteristics:

- T=0: Locked output
- T=1: Inverts output; $Q \rightarrow \overline{Q}$

The output can only change values when the clock signal 'C' rises.

The T-Flip-Flop can easily be used to create counters.

8.2 Asynchronous Latches

Asynchronous digital circuits in general, are circuits whose state changes are not governed by a dedicated clock signal, i.e., they can change state any time as an immediate consequence of a change of an input. A more concise definition of 'asynchronous' is 'not synchronous'.

8.2.1 Gated D-latch/transparent latch

The *D-latch* is the simplest flip-flop type.

Gated (in contrast to clocked) means that the output state may change with an input signal while a gating signal ('E' in equation 2) is high and does no more change when the gating signal is low (or vice versa).

The D-latch's behaviour is defined by;

$$Q_{next} = D \cdot E + \overline{E} \cdot Q_{present} \quad (1)$$

Note that often the subscripts 'present' and 'next' are not explicitly written but it is assumed that the left hand side of the equation refers to the next state and the right hand to the present.

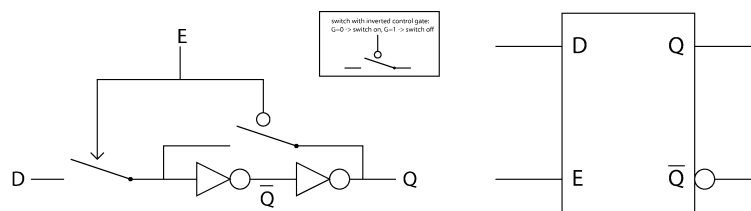


Figure 40: Possible implementation of gated D-latch and symbol

8.2.2 SR-Latch

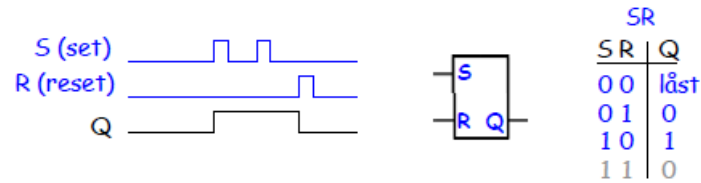


Figure 41: SR-latch

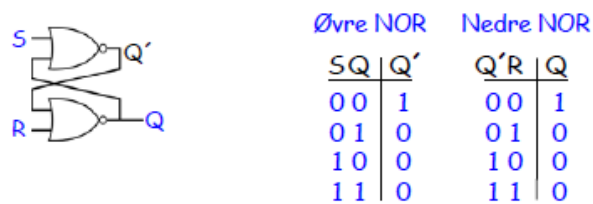


Figure 42: Possible implementation of SR-Latch

The *SR-Latch* has the following characteristics:

- $S=1 \rightarrow Q=1$: when 'S' goes back to '0', 'Q' will stay at '1'
- $R=1 \rightarrow Q=0$: resets 'Q'
- $S=1, R=1$: This state isn't normally used.

8.3 Finite State Machine (FSM)

Finite State Machines are a formal model suited to describe sequential logic, i.e., logic circuits whose output does not only depend on the present input but on internal memory and thus, on the history or sequence of the inputs. They describe circuits composed of combinational logic and flipflops.

8.3.1 State Transition Graphs

A common way to describe/define a FSM is by a state transition graph: a graph consisting of the possible states of the FSM represented as bobbles and state transitions represented as arrows that connect the bobbles. The state transitions are usually labeled with a state transition condition, i.e., a Boolean function of the FSM inputs.

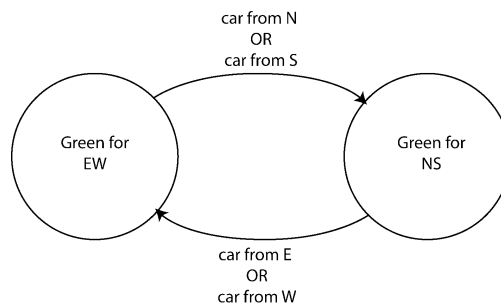


Figure 43: State transition graph

Consider the simple example in figure 43.

It defines a controller for a traffic light, where pressure sensors in the ground are able to detect cars waiting coming from either of the four roads. There are two states of this system, either north-south or east-west traffic is permitted. This FSM is governed by a slow clock cycle, let's say of 20 seconds. Equipped with sensors, the controller's behaviour is somewhat more clever than simply switching back and forth between permitting east-west and north-south traffic every cycle: it only switches, if there are cars waiting in the direction it switches to and will not stop the cars travelling in the direction that is green at present otherwise.

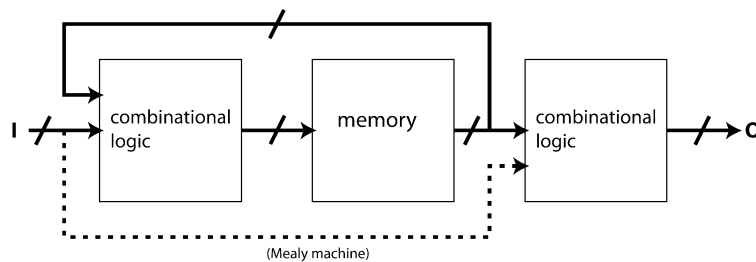
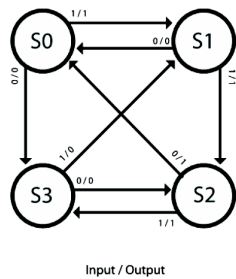


Figure 44: FSM

Figure 44 shows the principle block diagram of a Moore FSM. If the dashed connection is included, it becomes a Mealy FSM.

8.3.1.1 Example



Truthtable:

s_1	s_0	in	out	s_{1next}	s_{0next}
0	0	0	0	1	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	0	1	0
1	1	1	0	0	1

Show next state with Karnaugh;

out	IN	$s_1 s_0$	00	01	11	10
0	0		0	0	0	1
1	1		1	1	0	1

s_{1next}	IN	$s_1 s_0$	00	01	11	10
0	0		1	0	1	0
1	1		0	1	0	1

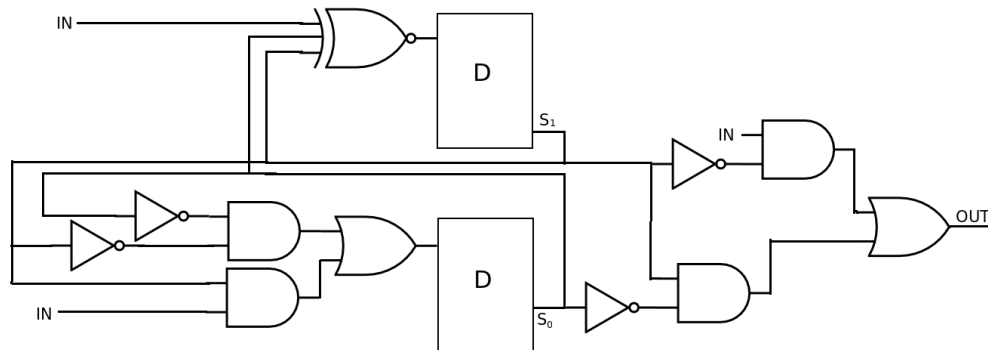
s_{0next}	IN	$s_1 s_0$	00	01	11	10
0	0		1	0	0	0
1	1		1	0	1	1

$$\text{out} = \bar{s}_1 IN + s_1 \bar{s}_0$$

$$s_{1next} = s_1 \oplus s_0 \oplus IN$$

$$s_{0next} = \bar{s}_1 \bar{s}_0 + s_1 IN$$

Implement by using D-flipflop;



8.3.2 Moore FSM

In a *Moore machine* the output depends solely on the internal states. In the traffic light example here, the traffic lights are directly controlled by the states and the inputs only influence the state transitions, so that is a typical Moore machine.

8.3.3 Mealy FSM

In a *Mealy machine* the outputs may also depend on the input signals directly. A Mealy machine can often reduce the number of states (naturally, since the 'state' of the input signals is exploited too), but one needs to be more careful when designing them. For one thing: even if all memory elements are synchronous the outputs too may change asynchronously, since the inputs are bound to change asynchronously.

8.3.4 Synchronous FSM

In general, it is simpler to design fully synchronous FSMs, i.e., with only synchronous flip-flops that all receive the same global clock signal. The design methods and especially the verification methods of the design are much more formalized and, thus, easier to perform.

8.3.5 Asynchronous FSM

On the other hand, asynchronous FSM implementations are potentially a good deal faster, since a state transition can occur as quickly as the state transition condition can be computed by the combinational logic, whereas a clocked FSM has to choose the clock period long enough such that the slowest of all possible state transition condition computation can be completed within a clock cycle. The design and verification, however, is tremendously more difficult and full of pitfalls.

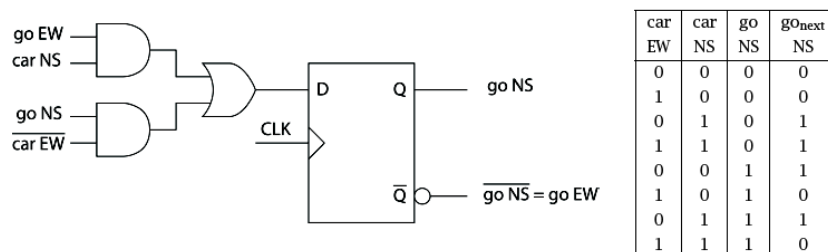


Figure 45: Traffic light example

If we go back to the traffic light example, it can be implemented as a *synchronous Moore machine* with D-flip-flops by first deriving the characteristic-/state transition table from the state transition graph. The conditions 'car from E or car from W' have been combined to 'car EW'. It has been chosen to represent the two states by a single D-flip-flop.

Note, that also the implicit conditions for a state to be maintained have to be included in the table, even if they are not explicitly stated in the graph.

8.4 Registers

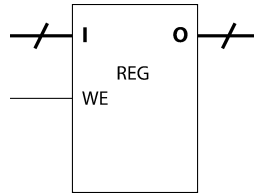


Figure 46: Symbol for a simple register

Registers are a concept that will simplify following discussions of more complex logic. They are nothing more fancy than an array of flip-flops that are accessed in parallel (e.g., as memory blocks in a CPU), controlled by shared control signals. The array is usually of a size that is convenient for parallel access in the context of a CPU/PC, e.g., one Byte or a Word. Possibly most common is the use of an array of D-Flip-Flops. A typical control signal is a 'write enable' (WE) or synchronous load (LD). In a D-Flip-Flop based register, this signal is 'and-ed' with the global clock and connected to the D-Flip-Flop clock input, such that a new input is loaded into the register only if WE is active. Other control signals might be used to control extra functionality (e.g., in shift-registers).

8.5 Standard Sequential Logic Circuits

8.5.1 Counter

Counters are a frequently used building block in digital electronics. A counter increases a binary number with each clock edge.

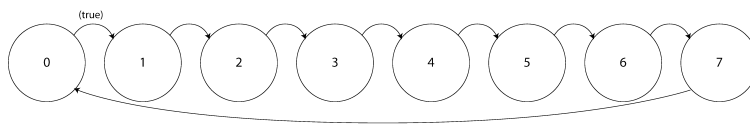


Figure 47: State transition graph of a 3-bit counter

present			in	next		
S_2	S_1	S_0	NA	S_2	S_1	S_0
0	0	0		0	0	1
0	0	1		0	1	0
0	1	0		0	1	1
0	1	1		1	0	0
1	0	0		1	0	1
1	0	1		1	1	0
1	1	0		1	1	1
1	1	1		0	0	0

		S_{2next}	
S_0	S_1	0	1
00		0	0
01		0	1
11		1	0
10		1	1

		S_{1next}	
S_0	S_1	0	1
00		0	1
01		1	0
11		1	0
10		0	1

		S_{0next}	
S_0	S_1	0	1
00		1	0
01		1	0
11		1	0
10		1	0

Figure 48: Truth table and karnaugh maps

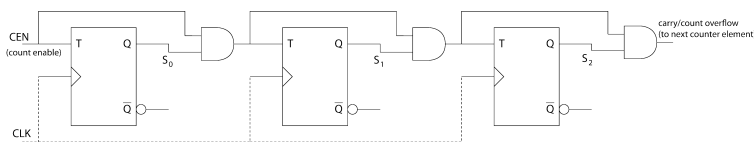


Figure 49: 3-bit counter

Counters may be equipped with even more control signals to control extra functionality such as:

- Possibility for loading an initial number (control signal LD and an input bus)
- Reset to zero (control signal RES)
- Switching between up and down counting (control signal UpDown)

8.5.2 Shift register

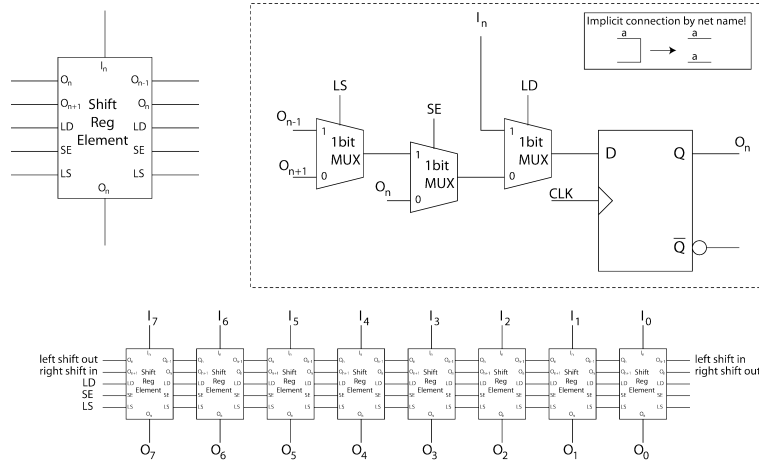


Figure 50: Shift register

Shift registers are registers that can shift the bits by one position per clock cycle. The last bit 'falls out' of the register when shifting. The first bit that is 'shifted in' can for example:

- be set to zero
- be connected to the last bit (cycling/ring counter)
- be connected to a serial input for serial loading

Typical control signals are:

- load (LD, for parallel loading)
- shift enable (SE, to enable or stop the shifting)
- left shift (LS, for controlling the direction of the shift)

Shift registers are used in:

- Parallel to serial and serial to parallel conversion
- Binary multiplication
- Ring 'one-hot' code counters/scanners
- Pseudo random number generators

9 VON NEUMAN ARCHITECTURE

The main novelty is that a single memory is used for both program *and* data.

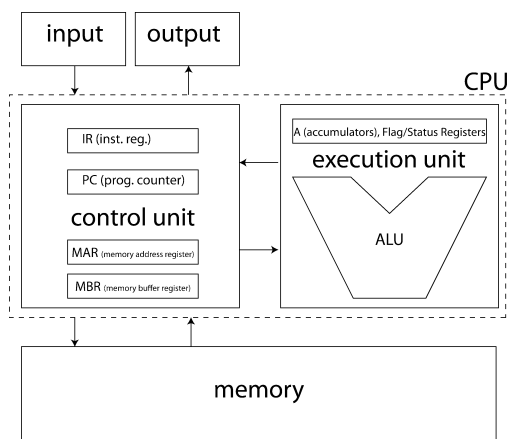


Figure 51: Von Neuman Architecture

9.1 Data Path and Memory Bus

The *data path* refers to the internal bus structure of the CPU. Buses are connections between registers, the functional units (such as the ALU), the memory, and I/O units. They are often shared by more than two of those units and usually only one unit sends data on the bus to one other at a time. Internally in the CPU there are usually several buses for data exchange among the registers and functional units, allowing parallel access to several registers at the same time, i.e., within one clock cycle. However, there is only one bus between the memory and the CPU for both instruction and data transfer in a von Neumann architecture (actually two: one for the address and one for the data), this bus can be a main speed limiting factor which is known as the von Neumann bottleneck.

9.2 Arithmetic and Logical Unit (ALU)

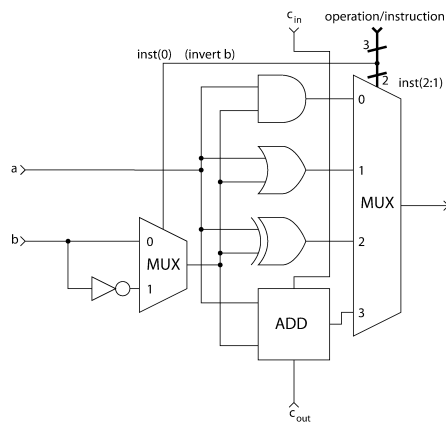


Figure 52: Example schematics of a 1-bit arithmetic logic unit (ALU)

The instruction code or operation code (inst/opcode) determines which function the ALU applies to its input as detailed in the truth table (figure 53).

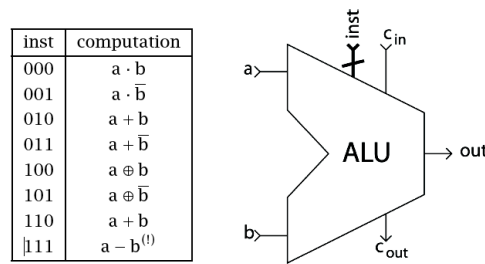


Figure 53: Truth table for and possible symbol for 1-bit ALU

Note

- Substraction will require \bar{b} and the carry in signal be set to 1 (in an n-bit ALU only for the LSB). This is to get $-b$. Invert b and add 1 (Two's complement).
- More complicated ALUs will have more functions as well as flags, e.g., overflow, divide by zero, etc.
- Modern CPUs contain several ALUs, e.g., one dedicated to memory pointer operations and one for data operations.
- TRADE-OFF: The design of the ALU is a major factor in determining the CPU performance. The complexity can be put either in the hardware or the software. Complex hardware can be expensive in power consumption, chip area and cost. Furthermore, the most complex operation may determine the maximal clock speed.

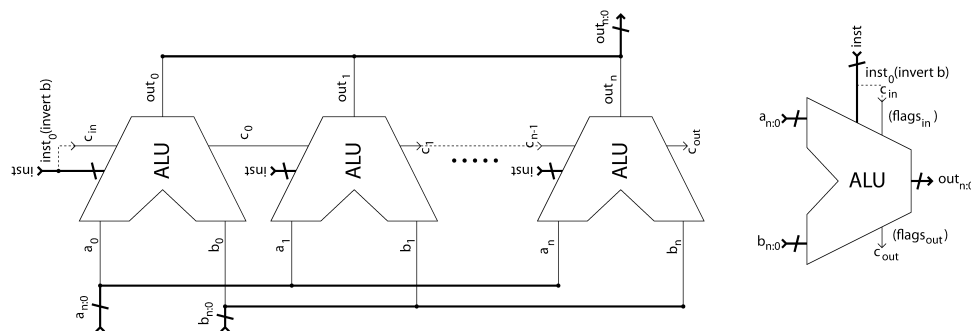


Figure 54: Example n-bit ALU schematic and symbol

9.3 Memory

The basic type of memory that is usually employed within a basic von Neumann architecture is random access memory (RAM). A RAM has an address port, and a data input/output port (combined or separately).

If the address port is kept constant, the circuit behaves like a single register that can store one word of data, i.e., as many bits as the input and output ports are wide.

Changing the address, one addresses a different word/register. 'Random access' refers to this addressed access that allows to read and write any of the words at any time, as opposed to the way that, for instance, pixels on a computer screen are accessible only in sequence.

9.3.1 Terminology

Address space

The number of words/bytes in a RAM. In figure 55 it is equivalent to the number of rows.

Word length

The number of bits or bytes that can be accessed in a single read/write operation, i.e., the number of bits addressed with a single address. In figure 55 the number of columns.

Memory size

The word length multiplied with address space.

Note that the x86 architecture (and other modern CPUs) allows instructions to address individual bytes in the main memory, despite the fact that the word length of the underlying RAM is actually 32 bits/4 bytes or 64 bits/8 bytes. Thus, the address space that a programmer sees is in fact bigger than the address space of the RAM.

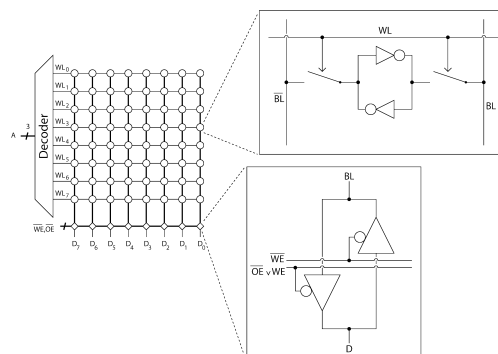


Figure 55: Static random access memory (SRAM) principle

9.3.2 Typical signals of a RAM

$A((n-1):0)$

The address port indicating which word should be accessed. 2^n is the address space.

$I/D((m-1):0)$ and $O((m-1):0)$ or $D((m-1):0)$

I (sometimes also referred to as D) and O are the input and output port respectively. Sometimes a single port D is used for both, in which case a control signal \overline{OE} is needed to distinguish between the use of port D as input or output. m is the memory word length in bits.

$\overline{\text{WE}}$, write enable (often active low)

This signal causes a write access writing I/D at address A, either immediately (asynchronous write), with a following strobe signal (see $\overline{\text{RAS}}/\overline{\text{CAS}}$) or with the next clock edge (synchronous write).

 $\overline{\text{RAS}}/\overline{\text{CAS}}$, row/column access strobe

Usually appears in DRAM that has a 3-D structure: one decoder for the row address, one for the column address and the word (conceptually) extends into the third dimension. The address bus is reused for both row and column address. First the row address is asserted on the address bus A and $\overline{\text{RAS}}$ is pulsed low, then the column address is asserted on the address bus and $\overline{\text{CAS}}$ is pulsed low. $\overline{\text{CAS}}$ is the final signal that triggers either the read (latching of the word into a read buffer) or write operation. The other control signals are asserted *before*. Several column accesses can be made for a single row access for faster access times.

 $\overline{\text{OE}}$, output enable

A signal that lets the RAM drive the data line while asserted, but lets an external source drive the data lines while deasserted. This can be used to regulate the access if there are several devices connected to a single bus: Only one of them should be allowed to drive the bus at one time. Deasserted, it can also allow a common data line to be used as input port.

 $\overline{\text{CS}}$, chip select

A control line that allows to use several RAMs instead of just one on the same address and data bus and sharing all other control signals. If $\overline{\text{CS}}$ is not asserted all other signals are ignored and the output is not enabled. Extra address bits are used to address one specific RAM and a decoder issues the appropriate CS to just one RAM at a time. This extends the address space.

9.3.3 Static Random Access Memory (SRAM)

Figure 55 shows an example block diagram of a static RAM with asynchronous WE. An active high signal on the word line (WL) will connect that latches content to the bit lines (BL and BL). The bit-lines connect the same bit in all words, but only ever one word line is active at anyone time. This is ensured by the decoder (lefthand side of the figure) that decodes the address A. Thus, a single word at a time can either be read or written to.

The tri-state buffer allow the outputs of different logic circuits to be connected to the same electrical node, e.g., a bus-line. usually, only one of these outputs at a time will be allowed to drive that line and determine its states, while all others are in principle disconnected from that line. So, the tri-state output can actually have three different states, as controlled by its control input and the input:

- input active: it conveys the usual 'high'/1 and 'low'/0 from input to output
- control input is inactive: the buffer acts like a switch that disconnects its input from the output.

The write access is somewhat less elegant than in a D-latch. The feedback loop is maintained while writing. An active low WE activates a tri-state buffer that drives the bit-lines. This buffer needs to be stronger than the feedback loop in order to overrule it. This saves space, but takes more power. The main criterion for memory cells, is to get as much memory in as little a space as possible.

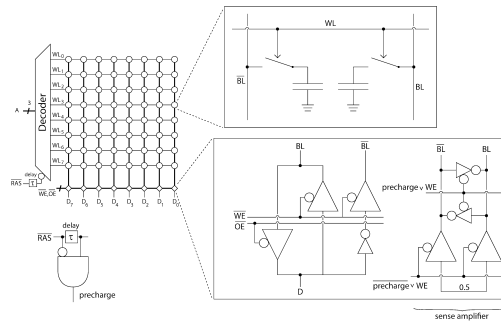


Figure 56: Dynamic random access memory (DRAM) principle

9.3.4 Dynamic Random Access Memory (DRAM)

While a SRAM needs 6 transistors per storage cell (2 per inverter and 1 per switch), a DRAM merely needs two capacitors and two transistors. Instead of an active feedback loop to maintain a state, the DRAM relies on charge stored on a capacitor. This requires a refresh on every memory cell, to this we use a sense amplifier. The sense amplifier reconstructs the digital state from the analog state that the memory cell has decayed to.

	SRAM	DRAM
access speed	+	-
memory density	-	+
no refresh needed	+	-
simple internal control	+	-
price per bit	-	+

Figure 57: SRAM vs. DRAM

9.4 Control Unit (CU)

9.4.1 Register Transfer Language (RTL)

expression	meaning
X	register X or unit X
[X]	the content of X
\leftarrow	replace/insert or execute code
M()	memory M
[M([X])]	memory content at address [X]

Figure 58: RTL grammar

To describe the control unit we use RTL to explain moving data among registers and telling the execution unit to manipulate some of these data. The syntax of RTL is illuminated in figure 58.

An example: $[IR] \leftarrow [MBR]$

(transfer the content of the MBR to the IR)

9.4.2 Execution of Instructions

IR (instruction register)

Machine code of instructions which is in order for execution.

PC (program counter / IP (instruction pointer))

Memory address for next instruction.

MAR (memory address register)

CPU register that either stores the memory address from which data will be fetched to the CPU or the address to which data will be sent and stored

MBR (memory buffer register)

Stores the data being transferred to and from the immediate access store

Every instruction consists of at least three tasks;

1. Fetch instruction

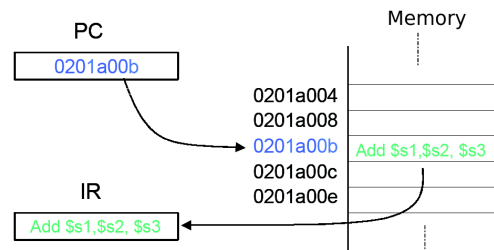


Figure 59: Gets instruction from memory, and moving it to IR

2. Decode instruction

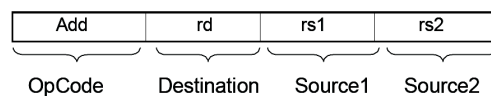


Figure 60: Decode instruction

OpCode: instruction type

Destination: result

Source 1: 1. operand for instruction

Source 2: 2. operand for instruction

Not all instructions use all the fields, and some use more fields

Additionally we need control signals to among others ALU.

3. Execute instruction

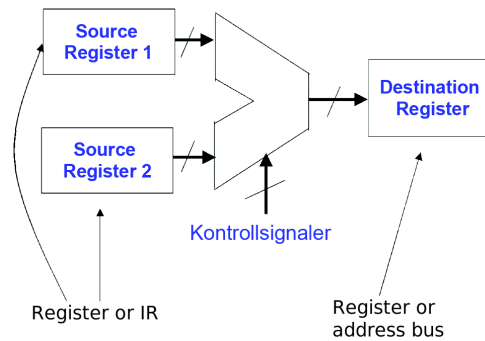


Figure 61: Execute instruction

NB! The source register can not be triggered by the same clock edge as the destination register.

9.4.3 Microarchitecture

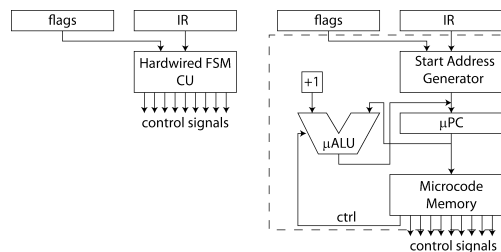


Figure 62: Hardwired and microprogrammed CU

Instead of hardwired CU architecture, where a hardwired FSM gives the control signals decoded from IR, a more flexible alternative is to use *microcode* and a simple processor which generates the control signals through micro instructions in the *microprogram memory* (typically ROM).

9.4.4 Complex and reduced instruction sets (CISC/RISC)

	Microarchitecture	Hardwired
Occurrence	CISC	RISC
Flexibility	+	-
Design Cycle	+	-
Speed	-	+
Compactness	-	+
Power	-	+

Figure 63: Microarchitecture vs. hardwired

9.5 Input/Output

A computer is connected to various devices transferring data to and from the main memory. This is referred to as Input/output (I/O).

Examples:

Keyboard, Graphics, Mouse, Network (Ethernet, Bluetooth, ...),
USB, Firewire, PCI, PCI-express, SATA, ...

Ways to communicate:

Internal communication between devices internal in the machine and between a computer and directly connected equipment.

External communication between different computers connected over network.

The performance of the I/O-system depends on various factors;

- The processor
- Memory Hierarchy
- The bus(es) that connects your computer
- Control units for I/O devices and associated bus'es
- The speed of the operating system
- The software's use of I/O

Two common benchmarks for performance for I / O are:

- Throughput: Bandwidth or throughput of data per unit of time.
- Response: Delay from start to reply

A machine usually has multiple independent bus'es that are specialized. Since a bus connects many different devices, it is often a bottleneck in the system, because many devices compete to use it. The devices share the same physical bus, which makes a need to decide which device can use the bus at what time. Therefore we use protocols. A protocol specifies the rules that apply to the use of the bus.

There are many different protocols for different bus types:

- ISA, PCI, TCP/IP, ATM, ...

9.5.1 Modes of transfer

Programmed/Polled

The processor is in full control of all aspects of the transfer. It *polls* the I/O status register in a loop to check if the controller has data to be collected from the port or is ready to receive data to the port. Polling uses up some CPU time and prevents the CPU from being used for other purposes while waiting for I/O.

Interrupt driven

The I/O controller signals with a dedicated 1bit data line (*interrupt request*(IRQ)) to the CPU that it needs servicing. The CPU is free to run other processes while waiting for I/O. If the interrupt is not masked in the corresponding CPU status register, the current instruction cycle is completed, the processor status is saved (PC and flags pushed onto stack) and the PC is loaded with the starting

address of an *interrupt handler*. The start address is found, either at a fixed memory location specific to the interrupt priority (*autovectored*) or stored in the controller and received by the CPU after having sent an interrupt acknowledge control signal to the device (*vectored*)

Direct Memory Access (DMA)

The processor is not involved, but the transfer is negotiated directly with the memory, avoiding copying to CPU registers first and the subroutine call to the interrupt handler. DMA is used for maximum speed usually by devices that write whole blocks of data to memory (e.g., disk controllers). The CPU often requests the transfer but then relinquishes control of the system bus to the I/O controller, which only at the completion of the block transfer notifies the CPU with an interrupt. (DMA poses another challenge to the cache as data can now become stale, i.e., invalid in the cache)

10 OPTIMIZING HARDWARE PERFORMANCE

10.1 Memory Hierarchy

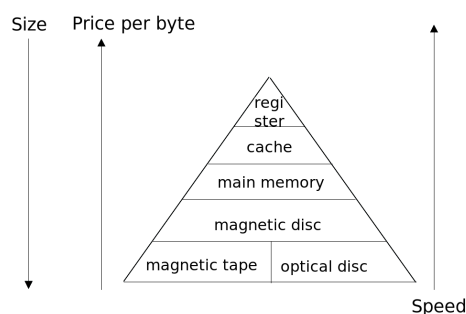


Figure 64:

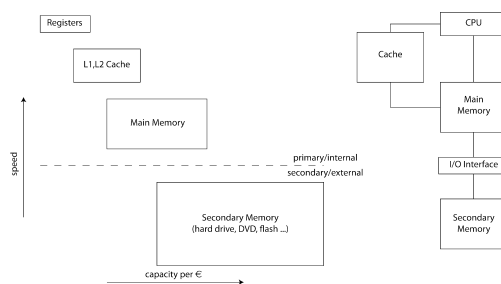


Figure 65: Memory Hierarchy

10.1.1 Applications**Register**

Internal notebook for CPU with fast access to content.

Cache

Quick cache for both instructions and data to smooth out the difference in speed between the CPU and RAM.

RAM

Buffer between the external medium and the CPU with fast read and write access.

SSD/Flash/Hard Drive

High capacity medium for program/data

10.1.2 Storage Capacity**Register**

Integrated in CPU, relatively few (32-128 pieces)

Cache

Cache internal (L1) or close (L2, L3) to the CPU, typical capacities are from 10 Kilobytes (L1) to 1 Megabyte (L2) and multiple Megabyte (L3).

RAM

Internally, the motherboard near the CPU, sizes over Gigabyte.

SSD/Flash/Hard Drive

External or internal storage device in the machine with capacity up to TeraByte.

10.1.3 access speed

Register	< 1 ns	≈ 100 Byte
L1 (on CPU) cache	≈ 1 ns	≈ 10 KB
L2, L3 (outside the CPU) cache	2-10 ns	≈ 1MB
RAM	20-100 ns	≈ 1 GB
SSD/Flash	100 ns-1μs	≈ 1 TB
Hard Drive	1 ms	≈ 1 TB

10.1.4 Cache

Cache improves the von Neumann architecture bottleneck with respect to the memory accessing. The cache is located close to the CPU to increase the speed. It is small to reduce production cost and therefore it is a trade-off discussion between size and speed. Cache contains a copy of a subset of the RAM. The CPU fetches data and/or instructions from cache instead of from the RAM.

Since the cache is smaller than the RAM, it must be determined what data/instructions must be placed in the cache. This is based on if the instructions/data is accessed;

- close in time
- close in space

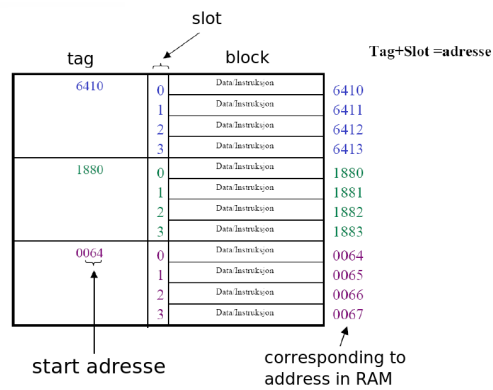


Figure 66: Cache

If the data/instruction the CPU need is in the cache, it is called a *cache hit*. If it's not in the cache, we get a *cache miss*. Therefore an essential part of the utilization of the cache is to quickly check for hit or miss.

10.1.4.1 Cache hit

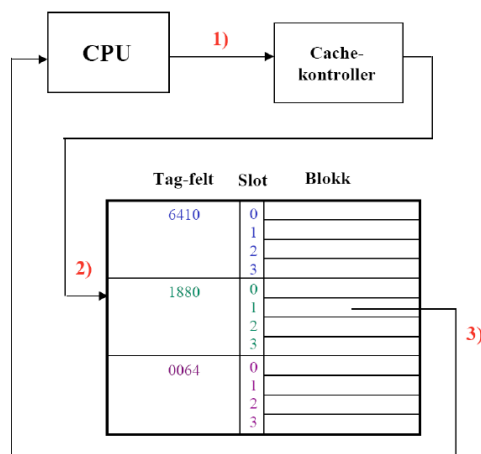


Figure 67: Cache hit

10.1.4.2 Cache miss

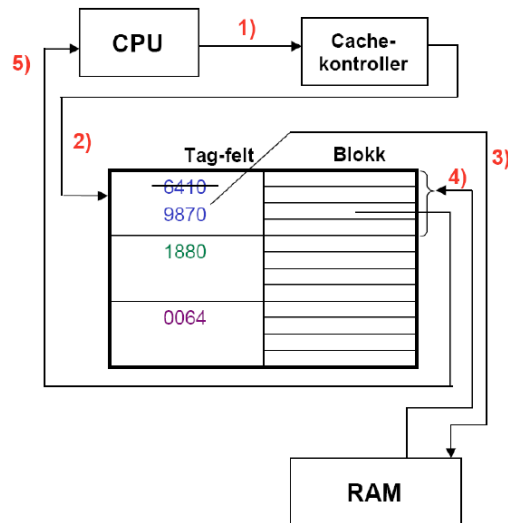


Figure 68: Cache miss

10.1.5 Cache mapping strategies

Direct Mapped Cache

A specific block of RAM can only be placed in a particular block in the cache. More RAM locations must "compete" for the same block in the cache.

- Advantage: Easy to check the appropriate block's cache: Check only one tag field and see if it contains the block you are looking for.
- Disadvantage: High miss-rate (even if space is available elsewhere, a block is placed on one specific place)

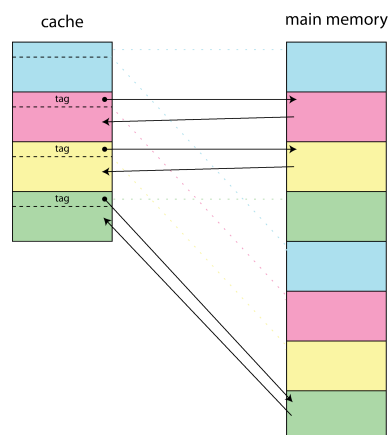


Figure 69: Direct mapped cache

Set-associative Cache A particular block of RAM can be placed in a limited number of block locations in cache.

- Advantage: Easy to check the appropriate block's cache: Search through a limited number of tag fields and see if the block is in cache.
- Disadvantage: Longer search time than direct-mapped (unless you're searching in parallel through the tag fields)

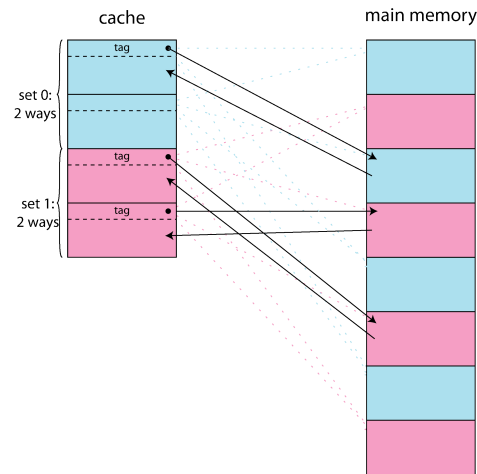


Figure 70: Set-associative cache

Full-associative Cache A specific block of RAM can be placed anywhere in the cache

- Advantage: The cache is fully utilized. Has the least chance of cache miss of all three methods.
- Disadvantage: The search to find a block can be timeconsuming. You can create mechanisms to simplify the search terms, eg. hashing. But this complicates the cache controller and requires additional hardware.

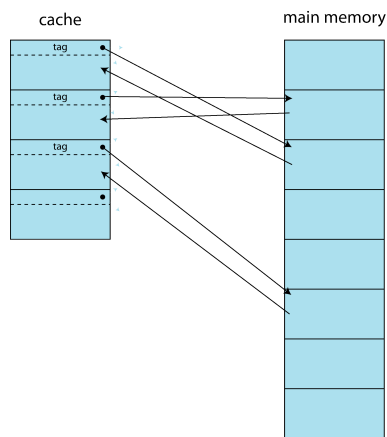


Figure 71: Full-associative cache

10.1.6 Cache coherency

A write operation will lead to a temporary inconsistency between the content of the cache and the main memory, called *dirty cache*. Several strategies are used in different designs to correct this inconsistency with varying delay. Major strategies are:

Write through

a simple policy where each write to the cache is followed by a write to the main memory. Thus, the write operations do not really profit from the cache.

Write back

delayed write back where a block that has been written to in the cache is marked as dirty. Only when dirty blocks are reused for another memory block will they be written back into the main memory.

10.1.7 Cache block replacement strategies

When we get a "cache miss", we sometimes need an existing block thrown out to make room for a new block. Which block is removed can have a major impact on the speed of the program being executed.

First-in-first-out (FIFO)

- The blocks are organized as a ring buffer.
- The block that has been in the memory for the longest time gets replaced (regardless of whether it is recently used)

Least recently used (LRU)

- The block that has been in the cache the longest time without being written/read to, gets overwritten.
- Pure LRU are rarely used because it involves a lot of administration which itself is consuming (including a timestamp that must be updated each time a block is accessed)

Random

- Throwing out a random block when you need space for a block.

Hybrid

- Divides blocks of time groups, and then throw out one randomly selected from the group that has remained longest in the cache without being used.

10.1.8 Cache Architecture

There are two distinct cache architectures with respect to where to place the cache in relation to the bus between the CPU and the main memory referred to as *look-through* and *look-aside*.

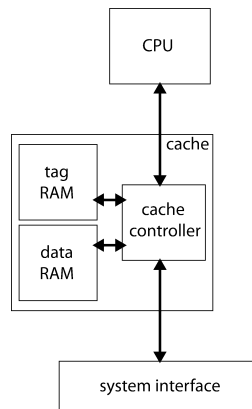


Figure 72: Look-through architecture

Look-through architecture

The cache is physically placed between CPU and memory (system interface).

- Memory access is initiated after a cache miss is determined (i.e., with a delay).
- Only if a cache miss is determined, is a memory access initiated.
- CPU can use cache while memory is in use by other units.

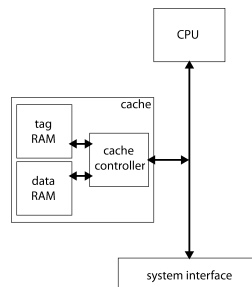


Figure 73: Look-aside architecture

Look-aside architecture

The cache shares the bus between CPU and memory (system interface).

10.1.9 Virtual Memory

Virtual memory extends the amount of main memory as seen by programs/processes beyond the capacity of the physical memory. Additional space on the hard drive (swap space) is used to store a part of the virtual memory that is, at present, not in use. The task of the virtual memory controller is quite similar to a cache controller: it distributes data between a slow and fast storage medium.

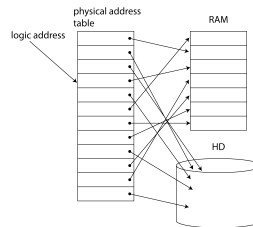


Figure 74: The principle of virtual memory

A virtual memory controller may simply be part of the operating system rather than a hardware component, but most in most designs today there is a HW memory management unit (MMU) using a translation look-aside buffer (TLB) that supports virtual memory on the hardware level.

The principle of virtual memory is that each logic address is translated into a physical address, either in the main memory or on the hard drive. Processes running on the CPU only see the logic addresses and a coherent virtual memory.

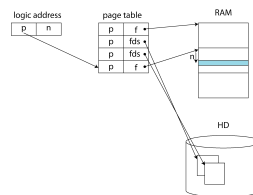


Figure 75: Virtual memory paging

A pointer for each individual logic address would require as much space as the entire virtual memory. Thus, a translation table is mapping memory blocks (called pages (fixed size) or segments (variable size)). A logic address can, thus, be divided into a page number and a page offset. A location in memory that holds a page is called page frame.

A translation look-aside buffer (TLB) is a cache for the page table, accelerating the translation of logic to physical address by the MMU. The interaction of these elements is shown in the block diagram and flow chart of figure 76.

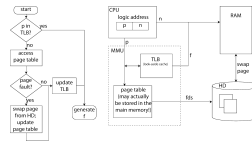


Figure 76: Block diagram and flow chart

Note that the penalty for page-failures is much more severe than that for cache misses, since a hard drive has an access time that is up to 50000 times longer than that of the main memory and about 1 million times longer than a register or L1 cache access, whereas the penalty of a cache miss is roughly less than a 100 times increased access time.

10.2 Speed up the single-cycle design

10.2.1 Multi-cycle

Multi-cycle processors break up the instruction into its fundamental parts, and executes each part of the instruction in a different clock cycle. Since signals have less distance to travel in a single cycle, the cycle times can be sped up considerably.

Typically, an instruction is executed over at least 5 cycles, which are named as such:

IF (instruction fetch)

Fetch the instruction from memory

ID (instruction decode)

Decode the instruction, and generate the necessary control signals

EX (execute)

Feed the necessary control signals into the ALU and produce a result

MEM

Read from memory, if specified

WB (write back)

Write the result back to the register file or to memory.

This is just a textbook example, and modern processes tend to use many more steps than this to execute an instruction.

- If necessary multi-cycle will use more than one clock cycle per instruction.
- The different steps in a instruction can share same component
- Don't need separate data- and instruction memory
- Needs extra registers to cache data

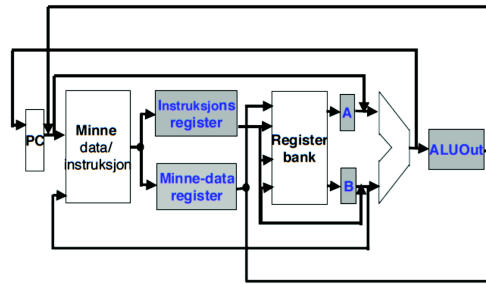


Figure 77: Multicycle

10.2.2 Pipelining

Introduces assembly line principle for executing instructions. Each instruction must be split into independent parts (subinstructions) performed consecutively. Each subinstruction can be performed independently of the other subinstructions. Next instruction is started before the previous instruction is completely finished. Every instruction takes the same amount of time to execute, but the processor performs multiple instruction at the same time.

For example, we have the following subinstructions in a pipeline

- IF: Instruction fetch (See the instruction)
- DE: decode and load (from a register)
- EX: Execute
- WB: Write back (write the result to a register)

PS! The read and write from the register/MEM can be made on each half of a cycle ($1/2$ clock period)

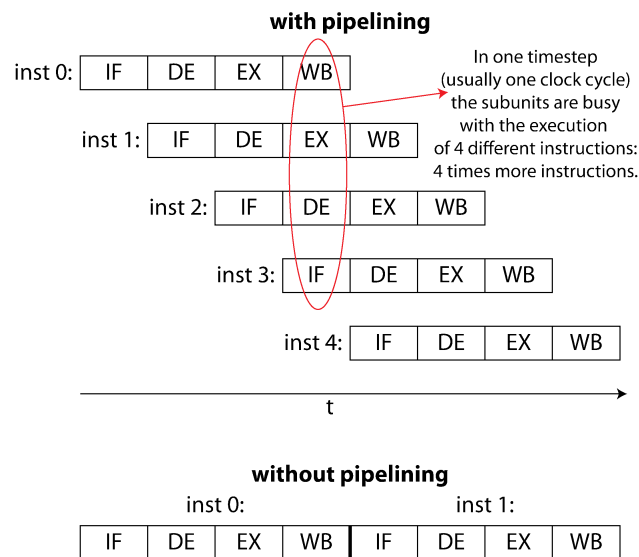


Figure 78: 4-stage pipeline execution

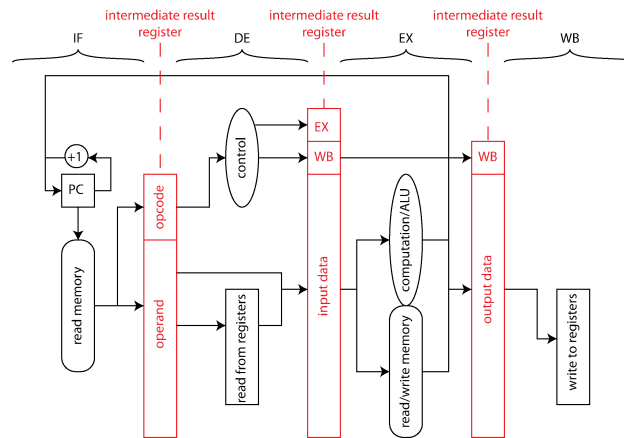


Figure 79: 4-stage pipeline simplified block diagram

Pipelining with multiple steps

- The 4-stage pipeline is the shortest that exists for a CPU.
- Modern CPU use significantly more steps
- Pentium III has 16 steps
- Pentium 4 has 31 steps

10.2.2.1 Speed-up

Having a four stage pipeline does not mean you get four times faster processing. It's always some time spent on administration of instructions.

Effective speed-up

A k -stage pipeline that requires one clock cycle per execution of n instructions, will gain a speed-up of;

$$\frac{kn}{k + n - 1} \quad (2)$$

Another popular measure of the performance of the pipeline is the *average clock cycles per instructions* (CPI). Ideally, it would be 1, but due to the initial delay of 'filling' the pipeline, it is:

$$CPI = \frac{k + n - 1}{n} \quad (3)$$

10.2.2.2 Pipelining Hazards

At any given time there may be subinstructions from four instructions in a pipeline. Sometimes, not all subinstructions are valid. Next instruction can't be executed right after if this happens. Such a situation is called a pipeline hazard.

The three major classes of these hazards:

- 1 Resource hazard
- 2 Data hazard
- 3 Control Hazard

10.2.2.2.1 Resource Hazards

Can occur if two subinstructions in a pipeline want to access the same resource (eg. memory)

- Solution 1: Designing so this will not occur (!)
- Solution 2: Stop (STALL) the pipeline so the resource can be accessed sequentially. (read in a NOP?)
- Solution 3: Use the local registry which is organized in a REGISTER FILE
- Solution 4: Use the Harvard architecture that has two separate memories for data and instructions.

Other Resources can also provide hazards, slightly depending on how the CPU architecture is built;

- Memory, cache, registry files, buses, ALU, ...

10.2.2.2.2 Data Hazards

Data hazard occur because two different instructions need to access the same data simultaneously, or if it need results from previous instruction before it has produced a valid response.

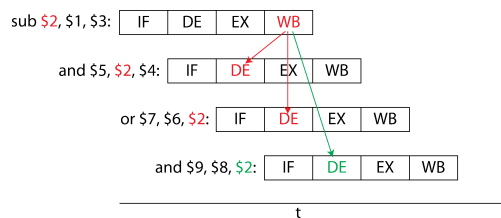


Figure 80: Data hazard illustration

- Solution 1: Detecting dependence in the IF stage of the pipeline for the first instruction and stop (STALL) next instructions EX-stage to the WB-stage until the first instruction is completed.
- Solution 2: Turn on the order of instructions so that one does not depend on instruction earlier in the pipeline.
- Solution 3: Have a shortcut (FORWARDING) in the pipeline, in this case a direct datapath which is activated by a hazard.

The best solution is number three. The reason for this is that all happens in the hardware and not in the compiler. Forwarding is also used between other devices in the datapath, eg. between the output of MEM and the entrance of the ALU.

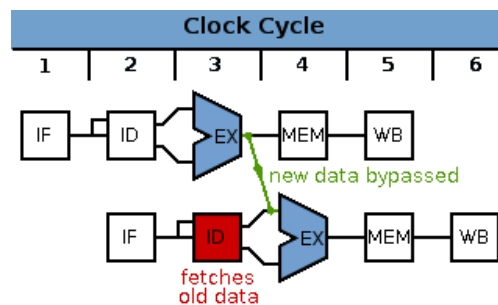


Figure 81: Forwarding

10.2.2.2.3 Control Hazards

Initially we have a pipeline that obtains the next instruction continuously. But the problem arises when we get a JUMP instruction. Then we need to clear the pipeline for other instructions and read in a new one.

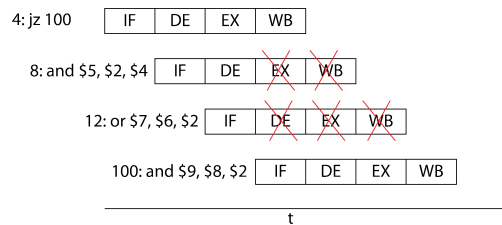


Figure 82: Control hazard illustration

- Solution 1: Stop (STALL) and not retrieve other instructions before it is clear that one should not jump.
- Solution 2: Try to predict whether or not to jump, execute as usual when not jumping, stop pipeline at jumping (as solution 1).
- Solution 3: Doubling of hardware for portions of the pipeline. This is to have two parallel pipelines that include both instruction addresses. If the instruction is a jump then "flush" the other.

10.3 Superscalar CPU

A superscalar CPU architecture implements a form of parallelism called instruction level parallelism within a single processor. It therefore allows faster CPU throughput than would otherwise be possible at a given clock rate. A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier.

In the Flynn taxonomy, a single-core superscalar processor is classified as an SISD processor (Single Instructions, Single Data), while a multi-core superscalar processor is classified as an MIMD processor (Multiple Instructions, Multiple Data).

While a superscalar CPU is typically also pipelined, pipelining and superscalar architecture are considered different performance enhancement techniques.

The superscalar technique is traditionally associated with several identifying characteristics (within a given CPU core):

- Instructions are issued from a sequential instruction stream
- CPU hardware dynamically checks for data dependencies between instructions at run time (versus software checking at compile time)
- The CPU accepts multiple instructions per clock cycle

Part II

Low-level programming

11 Assembly

11.1 Registers

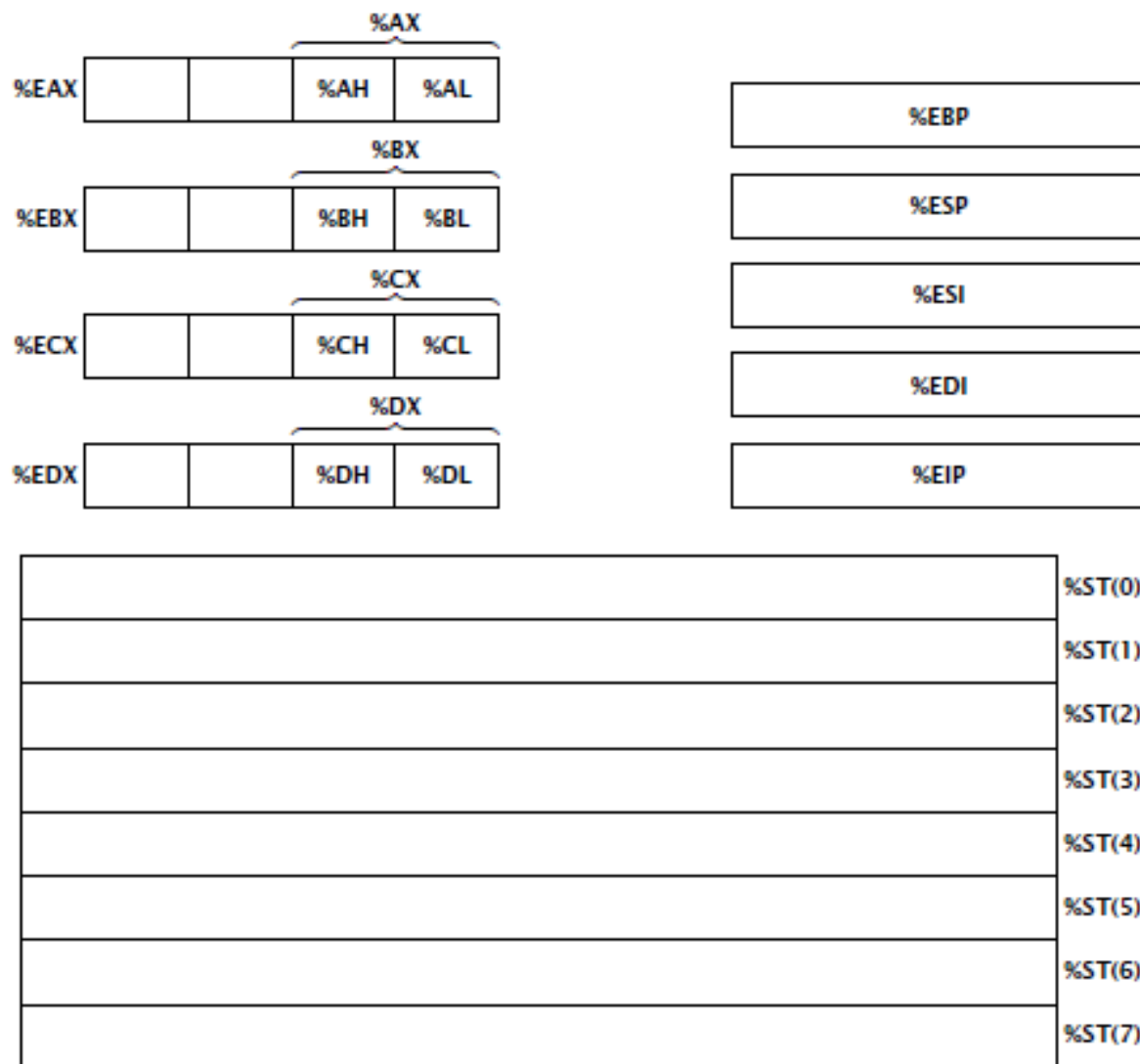


Figure 83: The most important x86/x87 registers

11.2 Instructions

Instruction	Explanation	Effect	C	S	Z
Data movement					
lea {bwl} {cmr},{mr}	Copy address	{mr} ← Adr({cmr})			
mov {bwl} {cmr},{mr}	Copy data	{mr} ← {cmr}			
pop {wl} {r}	Pop value	{r} ← pop			
push {wl} {cr}	Push value	push {cr}			
Block operations					
cld	Clear D-flag	D ← 0			
cmps b	Compare byte	(%EDI) − (%ESI); %ESI ← %ESI ± 1; %EDI ← %EDI ± 1	✓	✓	✓
movs b	Move byte	(%EDI) ← (%ESI); %ESI ← %ESI ± 1; %EDI ← %EDI ± 1			
rep (instr)	Repeat	Repeat (instr) %ECX times			
repnz (instr)	Repeat until zero	Repeat (instr) %ECX times while Z			
repz (instr)	Repeat while zero	Repeat (instr) %ECX times while Z			
scas b	Scan byte	%AL − (%EDI); %EDI ← %EDI ± 1	✓	✓	✓
std	Set D-flag	D ← 1			
stos b	Store byte	(%EDI) ← %AL; %EDI ← %EDI ± 1			
Arithmetic					
adc {bwl} {cmr},{mr}	Add with carry	{mr} ← {mr} + {cmr} + C	✓	✓	✓
add {bwl} {cmr},{mr}	Add	{mr} ← {mr} + {cmr}	✓	✓	✓
dec {bwl} {mr}	Decrement	{mr} ← {mr} − 1		✓	✓
div b {mr}	Unsigned divide	%AL ← %AX/{mr}; %AH ← %AX mod {mr}	?	?	?
div w {mr}	Unsigned divide	%AX ← %DX:%AX/{mr}; %DH ← %DX:%AX mod {mr}	?	?	?
div l {mr}	Unsigned divide	%EAX ← %EDX:%EAX/{mr}; %EDX ← %EDX:%EAX mod {mr}	?	?	?
idiv b {mr}	Signed divide	%AL ← %AX/{mr}; %AH ← %AX mod {mr}	?	?	?
idiv w {mr}	Signed divide	%AX ← %DX:%AX/{mr}; %DH ← %DX:%AX mod {mr}	?	?	?
idiv l {mr}	Signed divide	%EAX ← %EDX:%EAX/{mr}; %EDX ← %EDX:%EAX mod {mr}	?	?	?
imul b {mr}	Signed multiply	%AX ← %AL × {mr}	✓	?	?
imul w {mr}	Signed multiply	%DX:%AX ← %AX × {mr}	✓	?	?
imul l {mr}	Signed multiply	%EDX:%EAX ← %EAX × {mr}	✓	?	?
imul {wl} {cmr},{mr}	Signed multiply	{mr} ← {mr} × {cmr}	✓	?	?
inc {bwl} {mr}	Increment	{mr} ← {mr} + 1		✓	✓
mul b {mr}	Unsigned multiply	%AX ← %AL × {mr}	✓	?	?
mul w {mr}	Unsigned multiply	%DX:%AX ← %AX × {mr}	✓	?	?
mull {mr}	Unsigned multiply	%EDX:%EAX ← %EAX × {mr}	✓	?	?
neg {bwl} {mr}	Negate	{mr} ← −{mr}	✓	✓	✓
sub {bwl} {cmr},{mr}	Subtract	{mr} ← {mr} − {cmr}	✓	✓	✓
Masking					
and {bwl} {cmr},{mr}	Bit-wise AND	{mr} ← {mr} ∧ {cmr}	0	✓	✓
not {bwl} {mr}	Bit-wise invert	{mr} ← $\overline{\text{mr}}$			
or {bwl} {cmr},{mr}	Bit-wise OR	{mr} ← {mr} ∨ {cmr}	0	✓	✓
xor {bwl} {cmr},{mr}	Bit-wise XOR	{mr} ← {mr} ⊕ {cmr}	0	✓	✓

Figure 84: A subset of the x86 instructions

Instruction	Explanation	Effect	C	S	Z
Extensions					
cbw	Extend byte→word	8-bit %AL is extended to 16-bit %AX			
cwd	Extend word→double	16-bit %AX is extended to 32-bit %DX:%AX			
cwde	Extend double→ext	Extends 16-bit %AX to 32-bit %EAX			
cdq	Extend ext→quad	Extends 32-bit %EAX to 64-bit %EDX:%EAX			
Shifting					
rcl{bwl} {c},{mr}	Left C-rotate	$\{mr\} \leftarrow (\{mr\}, C) \cup^{[c]}$	✓		
rcr{bwl} {c},{mr}	Right C-rotate	$\{mr\} \leftarrow (\{mr\}, C) \cup^{[c]}$	✓		
rol{bwl} {c},{mr}	Left rotate	$\{mr\} \leftarrow \{mr\} \cup^{[c]}$	✓		
ror{bwl} {c},{mr}	Right rotate	$\{mr\} \leftarrow \{mr\} \cup^{[c]}$	✓		
sal{bwl} {c},{mr}	Left shift	$\{mr\} \leftarrow \{mr\} \stackrel{[c]}{\ll} 0$	✓	✓	✓
sar{bwl} {c},{mr}	Right arithmetic shift	$\{mr\} \leftarrow S \stackrel{[c]}{\gg} \{mr\}$	✓	✓	✓
shr{bwl} {c},{mr}	Right logical shift	$\{mr\} \leftarrow 0 \stackrel{[c]}{\gg} \{mr\}$	✓	✓	✓
Testing					
bt{wl} {c},{mr}	Bit-test	bit {c} of {mr}	✓		
btc{wl} {c},{mr}	Bit-change	bit {c} of {mr} ← (bit {c} of {mr})	✓		
btr{wl} {c},{mr}	Bit-clear	bit {c} of {mr} ← 0	✓		
bts{wl} {c},{mr}	Bit-set	bit {c} of {mr} ← 1	✓		
cmp{bwl} {cmr} ₁ , {cmr} ₂	Compare values	{cmr} ₂ − {cmr} ₁	✓	✓	✓
test{bwl} {cmr} ₁ , {cmr} ₂	Test bits	{cmr} ₂ ∧ {cmr} ₁	✓	✓	✓
Jumps					
call {a}	Call	push %EIP; %EIP ← {a}			
ja {a}	Jump on unsigned >	if Z ∧ C: %EIP ← {a}			
jae {a}	Jump on unsigned ≥	if C: %EIP ← {a}			
jb {a}	Jump on unsigned <	if C: %EIP ← {a}			
jbe {a}	Jump on unsigned ≤	if Z ∨ C: %EIP ← {a}			
jc {a}	Jump on carry	if C: %EIP ← {a}			
je {a}	Jump on =	if Z: %EIP ← {a}			
jmp {a}	Jump	%EIP ← {a}			
jg {a}	Jump on >	if Z ∧ S = 0: %EIP ← {a}			
jge {a}	Jump on ≥	if S = 0: %EIP ← {a}			
jl {a}	Jump on <	if S ≠ 0: %EIP ← {a}			
jle {a}	Jump on ≤	if Z ∨ S ≠ 0: %EIP ← {a}			
jnc {a}	Jump on non-carry	if C: %EIP ← {a}			
jne {a}	Jump on ≠	if Z: %EIP ← {a}			
jns {a}	Jump on non-negative	if S: %EIP ← {a}			
jnz {a}	Jump on non-zero	if Z: %EIP ← {a}			
js {a}	Jump on negative	if S: %EIP ← {a}			
jz {a}	Jump on zero	if Z: %EIP ← {a}			
loop {a}	Loop	%ECX ← %ECX − 1; if %ECX ≠ 0: %EIP ← {a}			
ret	Return	%EIP ← pop			
Miscellaneous					
rdtsc	Fetch cycles	%EDX:%EAX ← (number of cycles)			

Figure 85: A subset of the x86 instructions

Instruction	Explanation	Effect	C	S	Z
Load					
fldl	Float load 1	Push 1.0			
fildl {m}	Float int load long	Push long {m}			
fildq {m}	Float int load quad	Push long long {m}			
filds {m}	Float int load short	Push short {m}			
fldl {m}	Float load long	Push double {m}			
flds {m}	Float load short	Push float {m}			
fldz	Float load zero	Push 0.0			
Store					
fistl {m}	Float int store long	Store %ST(0) in long {m}			
fistpl {m}	Float int store and pop long	Pop %ST(0) into long {m}			
fistpq {m}	Float int store and pop quad	Pop %ST(0) into long long {m}			
fistq {m}	Float int store quad	Store %ST(0) in long long {m}			
fistps {m}	Float int store and pop short	Pop %ST(0) into short {m}			
fists {m}	Float int store short	Store %ST(0) in short {m}			
fstl {m}	Float store long	Store %ST(0) in double {m}			
fstpl {m}	Float store and pop long	Pop %ST(0) into double {m}			
fstps {m}	Float store and pop short	Pop %ST(0) into float {m}			
fstz {m}	Float store short	Store %ST(0) in float {m}			
Arithmetic					
fabs	Float absolute	$\%ST(0) \leftarrow \%ST(0) $			
fadd %ST(X)	Float add	$\%ST(0) \leftarrow \%ST(0) + \%ST(X)$			
fadd{sl} {m}	Float add	$\%ST(0) \leftarrow \%ST(0) + \text{float/double } \{m\}$			
faddp {m}	Float add and pop	$\%ST(1) \leftarrow \%ST(0) + \%ST(1)$; pop			
fchs	Float change sign	$\%ST(0) \leftarrow -\%ST(0)$			
fdiv %ST(X)	Float div	$\%ST(0) \leftarrow \%ST(0) \div \%ST(X)$			
fdiv{sl} {m}	Float div	$\%ST(0) \leftarrow \%ST(0) \div \text{float/double } \{m\}$			
fdivp {m}	Float reverse div and pop	$\%ST(1) \leftarrow \%ST(0) \div \%ST(1)$; pop			
fdivrp {m}	Float div and pop	$\%ST(1) \leftarrow \%ST(1) \div \%ST(0)$; pop			
fiadd{sl} {m}	Float int add	$\%ST(0) \leftarrow \%ST(0) + \text{short/long } \{m\}$			
fidiv{sl} {m}	Float int div	$\%ST(0) \leftarrow \%ST(0) \div \text{short/long } \{m\}$			
fimul{sl} {m}	Float int mul	$\%ST(0) \leftarrow \%ST(0) \times \text{short/long } \{m\}$			
fisub{sl} {m}	Float int sub	$\%ST(0) \leftarrow \%ST(0) - \text{short/long } \{m\}$			
fmul %ST(X)	Float mul	$\%ST(0) \leftarrow \%ST(0) \times \%ST(X)$			
fmul{sl} {m}	Float mul	$\%ST(0) \leftarrow \%ST(0) \times \text{float/double } \{m\}$			
fmulp {m}	Float mul and pop	$\%ST(1) \leftarrow \%ST(0) \times \%ST(1)$; pop			
fsqrt	Float square root	$\%ST(0) \leftarrow \sqrt{\%ST(0)}$			
fsub %ST(X)	Float sub	$\%ST(0) \leftarrow \%ST(0) - \%ST(X)$			
fsub{sl} {m}	Float sub	$\%ST(0) \leftarrow \%ST(0) - \text{float/double } \{m\}$			
fsubp {m}	Float reverse sub and pop	$\%ST(1) \leftarrow \%ST(0) - \%ST(1)$; pop			
fsubrp {m}	Float sub and pop	$\%ST(1) \leftarrow \%ST(1) - \%ST(0)$; pop			
fyl2xpl	Float ???	$\%ST(1) \leftarrow \%ST(1) \times \log_2(\%ST(0) + 1)$; pop			
Stack operations					
fld %STX	Float load	Push copy of %ST(X)			
fst %STX	Float store	Store copy of %ST(0) in %ST(X)			
fstp %STX	Float store and pop	Pop %ST(0) into %ST(X)			

Figure 86: A subset of the x87 floating-point instructions

11.3 ASCII-table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Figure 87: ASCII

12 C

12.1 Inline-code

```
uint mult (uint a, uint b)
{
    uint res, top;
    asm("mull %%edx" :
        "=a" (res), "=d" (top) :
        "a" (a), "d" (b));
    if (top) {
        fprintf(stderr,
            "\n**Overflow** \n");
        exit(1);
    }
    return res;
}
```

12.1.1 Assembly

The code is written as usual, except;
registers is indicated by %%eax
%0,%1, ... specify parameters
multiple instructions is separated by \n

12.1.2 Parameters

Input and output uses a special notation;

"xxx" (var)

which is interpreted as follows:

- Variable 'var' set a C variable.
- The specification xxx place restrictions on how variable comes to assembly code:

a register %eax

g no restrictions

b registers %ebx

n same as param n

r an arbitrary registry

= variable is changed

m memory