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 substraction ................................................ 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
## II Low-level programming

### 11 Assembly
- 11.1 Registers .............................................. 52
- 11.2 Instructions ............................................. 53
- 11.3 ASCII-table ............................................ 56

### 12 C
- 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:

\[(123)_{\text{HEX}}
\]
1 → 0001
2 → 0010
3 → 0011
→ (0001 0010 0011)_{\text{BIN}}

1.2 Dec to bin
Divide on two, keep remainder.
Example:

\[(123)_{\text{DEC}}
\]
123 : 2 → 1 → LSB
61 : 2 → 1
30 : 2 → 0
15 : 2 → 1
7 : 2 → 1
3 : 2 → 1
1 : 2 → 1
→ (0111 1011)_{\text{BIN}}

1.3 Oct to bin
Each octadecimal consists of three binary numbers.
Example:

\[(123)_{\text{OCT}}
\]
1 → 001
2 → 010
3 → 011
→ (001 010 011)_{\text{BIN}}
1.4 Hex to dec

Like every conversion to decimal, we do the following:

Example:

\[(123,123)_{\text{HEX}}\]

\[1 \rightarrow 1 \cdot 16^2 \rightarrow 256\]
\[2 \rightarrow 2 \cdot 16^1 \rightarrow 32\]
\[3 \rightarrow 3 \cdot 16^0 \rightarrow 3\]
\[1 \rightarrow 1 \cdot 16^{-1} \rightarrow \frac{1}{16} \rightarrow 0.0625\]
\[2 \rightarrow 2 \cdot 16^{-2} \rightarrow \frac{1}{32} \rightarrow 0.0078125\]
\[3 \rightarrow 2 \cdot 16^{-3} \rightarrow \frac{3}{16} \rightarrow 0.000732421875\]
\[\rightarrow 256 + 32 + 3 + 0.0625 + 0.0078125 + 0.000732421875\]
\[\rightarrow (291.0710449)_{\text{DEC}}\]

General:
\[(\ldots a_2a_1a_0, a_{-1}a_{-2} \ldots)_r = \ldots + a_2 \cdot r^2 + a_1 \cdot r^1 + a_0 \cdot r^0 + a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} + \ldots\]

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

\[\begin{align*}
1010 \text{ (inverted)} \\
+ 0001 \text{ (add one)} \\
\hline
= 1011 \text{ (-5)}
\end{align*}\]

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

\[\begin{align*}
00100 \text{ (inverted)} \\
+ 00001 \text{ (add one)} \\
\hline
= 00101 \text{ (-1)}
\end{align*}\]

\[\rightarrow -(1 \cdot 2^2 + 1 \cdot 2^0)\]
\[\rightarrow -5\]
# 3 BINARY MATHEMATICAL OPERATIONS

## 3.1 Binary addition

![Binary addition](image1.png)

Figure 1: Binary addition

## 3.2 Binary substraction

![Binary substraction](image2.png)

Figure 2: Binary substraction

## 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:

![Binary multiplication](image3.png)

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:

![Binary division diagram](image)

Figure 4: Binary division

4 BOOLEAN ALGEBRA

4.1 Gates

![AND gate](image)

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

![OR gate](image)

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

<table>
<thead>
<tr>
<th>abc</th>
<th>AND</th>
<th>OR</th>
<th>NAND</th>
<th>NOR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

\[ f(x, y, z) = y' + x'y'z + xz \]

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
# SIMPLIFICATION OF BOOLEAN EXPRESSIONS

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>$x + 0 = x$</td>
</tr>
<tr>
<td>P5</td>
<td>$x + x' = 1$</td>
</tr>
<tr>
<td>P3</td>
<td>$x + y = y + x$</td>
</tr>
<tr>
<td></td>
<td>$x + (y + z) = (x + y) + z$</td>
</tr>
<tr>
<td>P4</td>
<td>$x(y + z) = xy + xz$</td>
</tr>
<tr>
<td>T1a</td>
<td>$x + x = x$</td>
</tr>
<tr>
<td>T2a</td>
<td>$x + 1 = 1$</td>
</tr>
<tr>
<td></td>
<td>$x + xy = x$</td>
</tr>
<tr>
<td></td>
<td>$(x + y)' = x'y'$</td>
</tr>
<tr>
<td>P2</td>
<td>$x \cdot 1 = x$</td>
</tr>
<tr>
<td>P5</td>
<td>$xx' = 0$</td>
</tr>
<tr>
<td>P3</td>
<td>$xy = yx$</td>
</tr>
<tr>
<td></td>
<td>$x(yz) = (xy)z$</td>
</tr>
<tr>
<td>P4</td>
<td>$x + (yz) = (x + y)(x + z)$</td>
</tr>
<tr>
<td>T1b</td>
<td>$x \cdot x = x$</td>
</tr>
<tr>
<td>T2b</td>
<td>$x \cdot 0 = 0$</td>
</tr>
<tr>
<td></td>
<td>$x(x + y) = x$</td>
</tr>
<tr>
<td></td>
<td>$(xy)' = x' + y'$</td>
</tr>
</tbody>
</table>

Figure 9: Theorems and postulates

# KARNAUGH DIAGRAM

![Karnaugh Diagram](image.png)

Figure 10: $F = AD + AB' + CD + B'C$
7 COMBINATIONAL LOGIC CIRCUITS

Combination 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:

```
    D_0 = x'y'z'
    D_1 = x'y'z
    D_2 = x'yz'
    D_3 = x'yz
    D_4 = xy'z'
    D_5 = xy'z
    D_6 = xyz'
    D_7 = xyz
```

![Figure 11: 3 bit in / 8 bit out](image)

![Figure 12: Decoder symbol](image)

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$D_0$</td>
</tr>
<tr>
<td>$y$</td>
<td>$D_1$</td>
</tr>
<tr>
<td>$z$</td>
<td>$D_2$</td>
</tr>
<tr>
<td>$x$</td>
<td>$D_3$</td>
</tr>
<tr>
<td>$y$</td>
<td>$D_4$</td>
</tr>
<tr>
<td>$z$</td>
<td>$D_5$</td>
</tr>
<tr>
<td>$x$</td>
<td>$D_6$</td>
</tr>
<tr>
<td>$y$</td>
<td>$D_7$</td>
</tr>
</tbody>
</table>

![Figure 13: Truth table for 3 bit decoder](image)
7.1.1 Other variants

- Enable input
  Enable active → normal operation.
  Enable inactive → all outputs disabled.

![NAND logic: inverse output](image1.png)

Figure 14: NAND logic: inverse output

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

![4x16 decoder](image2.png)

Figure 15: 4x16 decoder
7.2 Encoder

- inverse function of a decoder.
It decodes $2^n$ inputs to $n$ outputs.
Example:
\[ x = D_4 + D_5 + D_6 + D_7 \]
\[ y = D_2 + D_3 + D_6 + D_7 \]
\[ z = D_1 + D_3 + D_5 + D_7 \]

![Diagram](image_url)

Figure 16: 8 bit in / 3 bit out

![Diagram](image_url)

Figure 17: Encoder symbol

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_2$</td>
<td>$D_1$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_0</td>
<td>y</td>
</tr>
<tr>
<td>D_1</td>
<td>x</td>
</tr>
<tr>
<td>D_2</td>
<td></td>
</tr>
<tr>
<td>D_3</td>
<td></td>
</tr>
<tr>
<td>D_4</td>
<td></td>
</tr>
<tr>
<td>D_5</td>
<td></td>
</tr>
<tr>
<td>D_6</td>
<td></td>
</tr>
<tr>
<td>D_7</td>
<td></td>
</tr>
</tbody>
</table>

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.

![Multiplexer symbol](image)

Figure 21: Multiplexer symbol

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 $S$.

![Demultiplexer symbol](image)

Figure 23: Demultiplexer symbol

7.4 Demultiplexer (DEMUX)
7.5 Adders (half/full)

7.5.1 Half Adder

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

<table>
<thead>
<tr>
<th>C\textsubscript{in}</th>
<th>a</th>
<th>b</th>
<th>S</th>
<th>C\textsubscript{out}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

![Clock signal](image)

Figure 30: Clock signal

8.1.1 D-Latch

The *D-Latch* has the following characteristics:
- Clk=1: D-Latch is transparent
- Clk=0: Locked output

![D-Latch](image)

Figure 31: Positive-edge-triggered D-Latch

The *D-Latch* has the following characteristics:
- Clk=1: D-Latch is transparent
- Clk=0: Locked output

![D-Latch](image)

Figure 32: D-Latch
8.1.2 D-Flip-Flop

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

A more compact version:

The *D-Flip-Flop* has the following characteristics:
- Clk=0: First D-Latch (master) is transparent
- Clk=1: Last D-Latch (slave) is transparent
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

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 most general flip-flop.
8.1.4 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;

\[ Q_{next} = D \cdot E + \overline{E} \cdot Q_{present} \tag{1} \]

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

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.

![Figure 41: SR-latch](image1)

![Figure 42: Possible implementation of SR-Latch](image2)
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.

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 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

Truth table:

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_0$</th>
<th>$in$</th>
<th>out</th>
<th>$s_{1next}$</th>
<th>$s_{0next}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Show next state with Karnaugh:

<table>
<thead>
<tr>
<th>IN</th>
<th>$s_1 s_0$</th>
<th>00 01 11 10</th>
<th>$s_{1next}$</th>
<th>00 01 11 10</th>
<th>$s_{0next}$</th>
<th>00 01 11 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>0</td>
<td>0 0 0 1</td>
<td>0</td>
<td>0 0 0 1</td>
<td>0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1 1 0 1</td>
<td>1</td>
<td>1 1 0 1</td>
<td>1</td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>

out = $s_1 IN + s_1 s_0$

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

$s_{0next} = s_1 s_0 + s_1 IN$

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 chose 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.

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](image)

*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.

![State transition graph of a 3-bit counter](image)

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

<table>
<thead>
<tr>
<th>present</th>
<th>in</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_2$  $s_1$  $s_0$</td>
<td>M</td>
<td>$s_2$  $s_1$  $s_0$</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>1 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

![Truth table and karnaugh maps](image)

**Figure 48:** Truth table and karnaugh maps

![3-bit counter](image)

**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

*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.

![Von Neuman Architecture](image.png)

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)

![Example schematics of a 1-bit arithmetic logic unit (ALU)](image.png)

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).

<table>
<thead>
<tr>
<th>inst</th>
<th>computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>a - b</td>
</tr>
<tr>
<td>001</td>
<td>a - b</td>
</tr>
<tr>
<td>010</td>
<td>a + b</td>
</tr>
<tr>
<td>011</td>
<td>a + b</td>
</tr>
<tr>
<td>100</td>
<td>a * b</td>
</tr>
<tr>
<td>101</td>
<td>a * b</td>
</tr>
<tr>
<td>110</td>
<td>a + b</td>
</tr>
<tr>
<td>111</td>
<td>a - b(^0)</td>
</tr>
</tbody>
</table>

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

Note

- Subtraction will require \( b \) 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.

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 \(OE\) is needed to distinguish between the use of port \(D\) as input or output. \(m\) is the memory word length in bits.
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 RAS/CAS) or with the next clock edge (synchronous write).

RAS/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 RAS is pulsed low, then the column address is asserted on the address bus and CAS is pulsed low. 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.

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.

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 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 (lethand 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.
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.

<table>
<thead>
<tr>
<th>Feature</th>
<th>SRAM</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>access speed</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>memory density</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>no refresh needed</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>simple internal control</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>price per bit</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Figure 57: SRAM vs. DRAM

9.4 Control Unit (CU)

9.4.1 Register Transfer Language (RTL)

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] ← [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

![Execute instruction diagram](Image)

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

![Microarchitecture diagram](Image)

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)

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occurrence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexibility</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Design Cycle</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Speed</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Compactness</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Power</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

![Microarchitecture vs. hardwired](Image)

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 uses 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/Pollled
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: Memory Hierarchy](image1)

![Figure 65: Memory Hierarchy](image2)
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

<table>
<thead>
<tr>
<th>Component</th>
<th>Time</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>&lt; 1 ns</td>
<td>≈ 100 Byte</td>
</tr>
<tr>
<td>L1 (on CPU) cache</td>
<td>≈ 1 ns</td>
<td>≈ 10 KB</td>
</tr>
<tr>
<td>L2, L3 (outside the CPU) cache</td>
<td>2-10 ns</td>
<td>≈ 1MB</td>
</tr>
<tr>
<td>RAM</td>
<td>20-100 ns</td>
<td>≈ 1 GB</td>
</tr>
<tr>
<td>SSD/Flash</td>
<td>100 ns-1µs</td>
<td>≈ 1 TB</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>1 ms</td>
<td>≈ 1 TB</td>
</tr>
</tbody>
</table>

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
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)

![Set-associative cache](image)

**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.

![Full-associative cache](image)
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](image)

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](image)

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.

![Block diagram and flow chart]

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
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)
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:

$$\frac{kn}{k + n - 1}$$

(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:

$$\text{CPI} = \frac{k + n - 1}{n}$$

(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
10.2.2.1 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 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.

![Data hazard illustration](image)

- 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.

![Forwarding](image)
10.2.2.3 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.

- 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data movement</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lea[bwl] [cmr],[mr]</td>
<td>Copy address: ( mr \rightarrow \text{Addr}(cmr) )</td>
<td></td>
</tr>
<tr>
<td>mov[bwl] [cmr],[mr]</td>
<td>Copy data: ( mr \rightarrow cmr )</td>
<td></td>
</tr>
<tr>
<td>pop[wl] [r]</td>
<td>Pop value: ( r \rightarrow \text{pop} )</td>
<td></td>
</tr>
<tr>
<td>push[wl] [r]</td>
<td>Push value: ( r \rightarrow \text{push}(r) )</td>
<td></td>
</tr>
<tr>
<td><strong>Block operations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clw</td>
<td>Clear D-flag: ( D \rightarrow 0 )</td>
<td></td>
</tr>
<tr>
<td>cmpsb</td>
<td>Compare byte: ( \text{SDF} \rightarrow \text{SDF} + 1 )</td>
<td></td>
</tr>
<tr>
<td>movsb</td>
<td>Move byte: ( \text{SDF} \rightarrow \text{SDF} )</td>
<td></td>
</tr>
<tr>
<td>rep (instr)</td>
<td>Repeat: ( \text{instr} \times \text{CX} )</td>
<td></td>
</tr>
<tr>
<td>repnz (instr)</td>
<td>Repeat until zero: ( \text{instr} \times \text{CX} )</td>
<td></td>
</tr>
<tr>
<td>repz (instr)</td>
<td>Repeat while zero: ( \text{instr} \times \text{CX} )</td>
<td></td>
</tr>
<tr>
<td>scasb</td>
<td>Scan byte: ( \text{AL} \rightarrow \text{SDF} \times \text{SDF} + 1 )</td>
<td></td>
</tr>
<tr>
<td>std</td>
<td>Set D-flag: ( D \rightarrow 1 )</td>
<td></td>
</tr>
<tr>
<td>stosb</td>
<td>Store byte: ( \text{SDF} \rightarrow \text{SDF} \times \text{SDF} + 1 )</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add[bwl] [cmr],[mr]</td>
<td>Add with carry: ( mr \rightarrow mr + cmr + C )</td>
<td></td>
</tr>
<tr>
<td>addl[bwl] [cmr],[mr]</td>
<td>Add: ( mr \rightarrow mr + cmr )</td>
<td></td>
</tr>
<tr>
<td>dec[bwl] [mr]</td>
<td>Decrement: ( mr \rightarrow mr - 1 )</td>
<td></td>
</tr>
<tr>
<td>divw [mr]</td>
<td>Unsigned divide: ( \text{AL} \rightarrow \text{AX} \times \text{AX} )</td>
<td></td>
</tr>
<tr>
<td>divl [mr]</td>
<td>Unsigned divide: ( \text{AL} \rightarrow \text{AX} \times \text{AX} )</td>
<td></td>
</tr>
<tr>
<td>idivb [mr]</td>
<td>Signed divide: ( \text{AL} \rightarrow \text{AH} \times \text{AX} )</td>
<td></td>
</tr>
<tr>
<td>idivw [mr]</td>
<td>Signed divide: ( \text{AL} \rightarrow \text{AX} \times \text{AX} )</td>
<td></td>
</tr>
<tr>
<td>intmul [mr]</td>
<td>Signed multiply: ( \text{AX} \rightarrow \text{AX} \times \text{AX} )</td>
<td></td>
</tr>
<tr>
<td>intmul [cmr],[mr]</td>
<td>Signed multiply: ( mr \rightarrow mr \times cmr )</td>
<td></td>
</tr>
<tr>
<td>inc[bwl] [mr]</td>
<td>Increment: ( mr \rightarrow mr + 1 )</td>
<td></td>
</tr>
<tr>
<td>mulw [mr]</td>
<td>Unsigned multiply: ( \text{AX} \rightarrow \text{AX} \times \text{AX} )</td>
<td></td>
</tr>
<tr>
<td>mul [mr]</td>
<td>Unsigned multiply: ( \text{AX} \rightarrow \text{AX} \times \text{AX} )</td>
<td></td>
</tr>
<tr>
<td>neg[bwl] [mr]</td>
<td>Negate: ( mr \rightarrow -mr )</td>
<td></td>
</tr>
<tr>
<td>sub[bwl] [cmr],[mr]</td>
<td>Subtract: ( mr \rightarrow mr - cmr )</td>
<td></td>
</tr>
<tr>
<td><strong>Masking</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and[bwl] [cmr],[mr]</td>
<td>Bit-wise AND: ( mr \rightarrow mr \times cmr )</td>
<td></td>
</tr>
<tr>
<td>not[bwl] [mr]</td>
<td>Bit-wise invert: ( mr \rightarrow \neg mr )</td>
<td></td>
</tr>
<tr>
<td>or[bwl] [cmr],[mr]</td>
<td>Bit-wise OR: ( mr \rightarrow mr \lor cmr )</td>
<td></td>
</tr>
<tr>
<td>xor[bwl] [cmr],[mr]</td>
<td>Bit-wise XOR: ( mr \rightarrow mr \oplus cmr )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 84: A subset of the x86 instructions
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
<th>Effect</th>
<th>C</th>
<th>S</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extensions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chw</td>
<td>Extend byte→word</td>
<td>8-bit %AL is extended to 16-bit %AX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cwd</td>
<td>Extend word→double</td>
<td>16-bit %AX is extended to 32-bit %DX:%AX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cwo</td>
<td>Extend double→ext</td>
<td>Extends 16-bit %AX to 32-bit %EAX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cdq</td>
<td>Extend ext→quad</td>
<td>Extends 32-bit %EAX to 64-bit %EDX:%EAX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Shifting</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shr(bwl) [c],[mr]</td>
<td>Left rotate</td>
<td>[mr] ← ([mr] C) ×[c]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcr(bwl) [c],[mr]</td>
<td>Right C-rotate</td>
<td>[mr] ← ([mr] C) ×[c]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rol(bwl) [c],[mr]</td>
<td>Left rotate</td>
<td>[mr] ← [mr] ×[c]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ror(bwl) [c],[mr]</td>
<td>Right rotate</td>
<td>[mr] ← [mr] ×[c]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sar(bwl) [c],[mr]</td>
<td>Left shift</td>
<td>[mr] ← [mr] ×[c]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>shr(bwl) [c],[mr]</td>
<td>Right arithmetic shift</td>
<td>[mr] ← [mr] ×[c]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>shr(bwl) [c],[mr]</td>
<td>Right logical shift</td>
<td>[mr] ← [mr] ×[c]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>btswl [c],[mr]</td>
<td>Bit-test</td>
<td>bit[c] of [mr]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bswl [c],[mr]</td>
<td>Bit-clear</td>
<td>[mr] ← 0</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bsxwl [c],[mr]</td>
<td>Bit-set</td>
<td>[mr] ← 1</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp(bwl) [cmr]2,[cmr]1</td>
<td>Compare values</td>
<td>[cmr]2 - [cmr]1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>test(bwl) [cmr]2,[cmr]1</td>
<td>Test bits</td>
<td>[cmr]2 ∧ [cmr]1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Jumps</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call [a]</td>
<td>Call</td>
<td>push %EIP; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ja [a]</td>
<td>Jump on unsigned &gt;</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jae [a]</td>
<td>Jump on unsigned ≥</td>
<td>IF C = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jb [a]</td>
<td>Jump on unsigned ≤</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jbe [a]</td>
<td>Jump on unsigned ≤</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jc [a]</td>
<td>Jump on carry</td>
<td>IF C = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>je [a]</td>
<td>Jump on =</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp [a]</td>
<td>Jump</td>
<td>%EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jq [a]</td>
<td>Jump on &gt;</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jgo [a]</td>
<td>Jump on ≥</td>
<td>IF S = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jl [a]</td>
<td>Jump on &lt;</td>
<td>IF S = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jle [a]</td>
<td>Jump on ≤</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jnc [a]</td>
<td>Jump on non-carry</td>
<td>IF C = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jna [a]</td>
<td>Jump on ≠</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jns [a]</td>
<td>Jump on non-negative</td>
<td>IF S = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jnz [a]</td>
<td>Jump on non-zero</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>js [a]</td>
<td>Jump on negative</td>
<td>IF S = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jz [a]</td>
<td>Jump on zero</td>
<td>IF Z = 0; %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop [a]</td>
<td>Loop</td>
<td>%ECX ← %ECX - 1; IF %ECX = 0 %EIP ← [a]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>Return</td>
<td>%EIP ← pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td>Fetch cycles</td>
<td>%EDX:%EAX ← (number of cycles)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 85: A subset of the x86 instructions
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
<th>Effect</th>
<th>C</th>
<th>S</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld</td>
<td>Float load 1</td>
<td>Push 1.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fldl [m]</td>
<td>Float int load long</td>
<td>Push long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fldq [m]</td>
<td>Float int load quad</td>
<td>Push long long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flds [m]</td>
<td>Float int load short</td>
<td>Push short [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fldl [m]</td>
<td>Float load long</td>
<td>Push double [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flds [m]</td>
<td>Float load short</td>
<td>Push float [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fldz</td>
<td>Float load zero</td>
<td>Push 0.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fistl [m]</td>
<td>Float int store long</td>
<td>Store %ST(0) in long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fistpl [m]</td>
<td>Float int store and pop long</td>
<td>Pop %ST(0) into long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fistpq [m]</td>
<td>Float int store and pop quad</td>
<td>Pop %ST(0) into long long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fistq [m]</td>
<td>Float int store quad</td>
<td>Store %ST(0) in long long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fstpl [m]</td>
<td>Float int store and pop short</td>
<td>Pop %ST(0) into short [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fist [m]</td>
<td>Float int store short</td>
<td>Store %ST(0) in short [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fstl [m]</td>
<td>Float store long</td>
<td>Store %ST(0) in double [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fstpl [m]</td>
<td>Float store and pop long</td>
<td>Pop %ST(0) into double [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsts [m]</td>
<td>Float store and pop short</td>
<td>Pop %ST(0) into float [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ftsl [m]</td>
<td>Float store short</td>
<td>Store %ST(0) in float [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fabs</td>
<td>Float absolute</td>
<td>%ST(0) -</td>
<td>%ST(0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd %ST(X)</td>
<td>Float add</td>
<td>%ST(0) - %ST(0) + %ST(X)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd [s] [m]</td>
<td>Float add</td>
<td>%ST(0) - %ST(0) + float/double [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>faddp [m]</td>
<td>Float add and pop</td>
<td>%ST(1) - %ST(0) + %ST(1); pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fchs</td>
<td>Float change sign</td>
<td>%ST(0) - -%ST(0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdiv %ST(X)</td>
<td>Float div</td>
<td>%ST(0) - %ST(0) / %ST(X)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdiv [s] [m]</td>
<td>Float div</td>
<td>%ST(0) - %ST(0) / float/double [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdwp [m]</td>
<td>Float reverse div and pop</td>
<td>%ST(1) - %ST(0) + %ST(1); pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdwrp [m]</td>
<td>Float div and pop</td>
<td>%ST(1) - %ST(1) + %ST(0); pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fiadd [s] [m]</td>
<td>Float int add</td>
<td>%ST(0) - %ST(0) + short/long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fidiv [s] [m]</td>
<td>Float int div</td>
<td>%ST(0) - %ST(0) / short/long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fimul [s] [m]</td>
<td>Float int mul</td>
<td>%ST(0) - %ST(0) * short/long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fismul [s] [m]</td>
<td>Float int sub</td>
<td>%ST(0) - %ST(0) / short/long [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fimul %ST(X)</td>
<td>Float mul</td>
<td>%ST(0) - %ST(0) * %ST(X)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmul [s] [m]</td>
<td>Float mul</td>
<td>%ST(0) - %ST(0) * float/double [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fimulp [m]</td>
<td>Float mul and pop</td>
<td>%ST(1) - %ST(0) * %ST(1); pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsqrt</td>
<td>Float square root</td>
<td>%ST(0) - √ST(0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsqrt %ST(X)</td>
<td>Float sqrt</td>
<td>%ST(0) - %ST(0) - %ST(X)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsub [s] [m]</td>
<td>Float sub</td>
<td>%ST(0) - %ST(0) - float/double [m]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsup %ST(X)</td>
<td>Float sub</td>
<td>%ST(0) - %ST(0) - %ST(X)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsub [s] [m]</td>
<td>Float reverse sub and pop</td>
<td>%ST(1) - %ST(0) - %ST(1); pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsubpr [m]</td>
<td>Float sub and pop</td>
<td>%ST(1) - %ST(1) - %ST(0); pop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fyl2xpl</td>
<td>Float ???</td>
<td>%ST(1) - %ST(1) * log2(%ST(0) + 1); pop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 86: A subset of the x87 floating-point instructions
### 11.3 ASCII-table

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hr</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hr</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hr</th>
<th>Oct</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>NUL (null)</td>
<td>32</td>
<td>20</td>
<td>040</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>100</td>
<td>#64:95</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td>SOH (start of heading)</td>
<td>65</td>
<td>41</td>
<td>101</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>141</td>
<td>#97:99</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>002</td>
<td>STX (start of text)</td>
<td>66</td>
<td>42</td>
<td>102</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>142</td>
<td>#99:99</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>003</td>
<td>ETX (end of text)</td>
<td>67</td>
<td>43</td>
<td>103</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>143</td>
<td>#99:99</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>004</td>
<td>EOT (end of transmission)</td>
<td>68</td>
<td>44</td>
<td>104</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>194</td>
<td>#100:99</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>005</td>
<td>ENQ (enquiry)</td>
<td>69</td>
<td>45</td>
<td>105</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>195</td>
<td>#101:99</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>006</td>
<td>ACK (acknowledge)</td>
<td>70</td>
<td>46</td>
<td>106</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>196</td>
<td>#102:99</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>007</td>
<td>BEL (bell)</td>
<td>71</td>
<td>47</td>
<td>107</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>197</td>
<td>#103:99</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>010</td>
<td>BS (backspace)</td>
<td>72</td>
<td>48</td>
<td>108</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>198</td>
<td>#104:99</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>011</td>
<td>TAD (horizontal tab)</td>
<td>73</td>
<td>49</td>
<td>109</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>199</td>
<td>#105:99</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>012</td>
<td>LF (NL line feed, new line)</td>
<td>74</td>
<td>4A</td>
<td>112</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>192</td>
<td>#106:99</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>013</td>
<td>VT (vertical tab)</td>
<td>75</td>
<td>4B</td>
<td>113</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>193</td>
<td>#107:99</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>014</td>
<td>FF (form feed, new page)</td>
<td>76</td>
<td>4C</td>
<td>114</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>194</td>
<td>#108:99</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>015</td>
<td>CR (carriage return)</td>
<td>77</td>
<td>4D</td>
<td>115</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>195</td>
<td>#109:99</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>016</td>
<td>SO (shift out)</td>
<td>78</td>
<td>4E</td>
<td>116</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>196</td>
<td>#110:99</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>017</td>
<td>SI (shift in)</td>
<td>79</td>
<td>4F</td>
<td>117</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>197</td>
<td>#111:99</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>020</td>
<td>DLE (data link escape)</td>
<td>80</td>
<td>50</td>
<td>120</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>192</td>
<td>#112:99</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>021</td>
<td>DC1 (device control 1)</td>
<td>81</td>
<td>51</td>
<td>121</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>193</td>
<td>#113:99</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>022</td>
<td>DC2 (device control 2)</td>
<td>82</td>
<td>52</td>
<td>122</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>194</td>
<td>#114:99</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>023</td>
<td>DC3 (device control 3)</td>
<td>83</td>
<td>53</td>
<td>123</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>195</td>
<td>#115:99</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>024</td>
<td>DC4 (device control 4)</td>
<td>84</td>
<td>54</td>
<td>124</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>196</td>
<td>#116:99</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>025</td>
<td>NAK (negative acknowledge)</td>
<td>85</td>
<td>55</td>
<td>125</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>197</td>
<td>#117:99</td>
</tr>
<tr>
<td>22</td>
<td>6</td>
<td>026</td>
<td>SYN (synchronous idle)</td>
<td>86</td>
<td>56</td>
<td>126</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>198</td>
<td>#118:99</td>
</tr>
<tr>
<td>23</td>
<td>7</td>
<td>027</td>
<td>ETB (end of trans. block)</td>
<td>87</td>
<td>57</td>
<td>127</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>199</td>
<td>#119:99</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>030</td>
<td>CAN (cancel)</td>
<td>88</td>
<td>58</td>
<td>130</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>19A</td>
<td>#120:99</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>031</td>
<td>EM (end of medium)</td>
<td>89</td>
<td>59</td>
<td>131</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>19B</td>
<td>#121:99</td>
</tr>
<tr>
<td>26</td>
<td>A</td>
<td>032</td>
<td>SUB (substitute)</td>
<td>90</td>
<td>5A</td>
<td>132</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>19D</td>
<td>#122:99</td>
</tr>
<tr>
<td>27</td>
<td>B</td>
<td>033</td>
<td>ESC (escape)</td>
<td>91</td>
<td>5B</td>
<td>133</td>
<td>[</td>
<td>123</td>
<td>7B</td>
<td>19E</td>
<td>#123:99</td>
</tr>
<tr>
<td>28</td>
<td>C</td>
<td>034</td>
<td>FS (file separator)</td>
<td>92</td>
<td>5C</td>
<td>134</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td>19F</td>
<td>#124:99</td>
</tr>
<tr>
<td>29</td>
<td>D</td>
<td>035</td>
<td>GS (group separator)</td>
<td>93</td>
<td>5D</td>
<td>135</td>
<td>]</td>
<td>125</td>
<td>7D</td>
<td>1A5</td>
<td>#125:99</td>
</tr>
<tr>
<td>30</td>
<td>E</td>
<td>036</td>
<td>RS (record separator)</td>
<td>94</td>
<td>5E</td>
<td>136</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>19B</td>
<td>#126:99</td>
</tr>
<tr>
<td>31</td>
<td>F</td>
<td>037</td>
<td>US (unit separator)</td>
<td>95</td>
<td>5F</td>
<td>137</td>
<td>_</td>
<td>127</td>
<td>7F</td>
<td>19C</td>
<td>#127:99</td>
</tr>
</tbody>
</table>

Figure 87: ASCII
12 C

12.1 Inline-code

```c
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