CompSys Exam Notes

Ulrik Stuhr Larsen, Frederik Tollstorff de Voss,
Lotte Maria Bruun, Marcus Astrup Hansen,
Emil Møller Hansen, Noah Maddox Shehadeh Stonall, Esben Hansen

January 2019

Contents

Chapter 1

Machine Architecture

1.1 Arithmetic

1.1.1 Boolean signs (bitwise operations in C)

1. ~: not
2. ^: xor
3. &: and
4. |: or

1.1.2 Boolean rules

Rules:

1. ~(A & B) = ~A | ~B – (De Morgan's Law),
2. (A ^B) & A = A & ~B,
3. (A & B) | (C & (A ^B)) = C ^((A ^C) & (B ^C))

1.2 Assembly

1.2.1 Assembly to C remarks

1. Sometimes the line "rep; ret" occurs when a jump comes just before a ret. The rep is there solely for performance reasons and can be ignored in this case.
2. When adding to a pointer in C, it adds 8 bytes, not 1 bytes as in Assembly. Thus when adding 8 in Assembler, there should only be added 1 in C.

- This is only in 64-bit systems! 32-bit systems uses int pointer and here C will add 4 bytes instead of 8.

3. Subtracting a pointer from a pointer in C gives a long with the distance between the addresses in bytes.
4. The purpose of leaq is to calculate a pointer (effective memory address) and putting it in a register. However, the "pointer" is not always used as a pointer but it does not matter as Assembly does not have types.
5. Shifting right a»k is the same as dividing a by 2^k. Shifting left a«k is the same as multiplying a by 2^k.

1.2.2 Important pages in book

The following is a list of pages with important tables, most of the tables may already be imported below, but here we go:

- Registers: Page 216 (imported)
- Memory entries: Page 217 (imported)
- Arithmetic: Page 228 (imported)
- Special arithmetic (eg. mul): Page 234 (imported)
- Comparisons: Page 238(imported)
- set-instructions: Page 239(imported)
- Jumps: Page 242(imported)
- Conditional moves: Page 253(imported)
- Pointer arithmetic: Page 293(not imported)

1.2.3 Assembly data suffixes and C type sizes

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

1.2.4 Registers

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

All argument registers and return register are caller saved.

When moving from a 32 bit register to a 64 bit one, the padding is automatic. Otherwise adding has to be done manually if wanted.

1.2.4.1 Callee and caller saved registers

This is about the responsibility for saving a register value when a function call is made inside a program. The called function maybe needs to use some of the same registers as the program is using. The program is responsible for saving

all **caller saved** registers in the memory before function call and loading them from memory into the registers after the function call.
Correspondingly, the called function is responsible for saving all **callee saved** registers in the memory before editing them and loading them from memory into the registers before returning.

1.2.5 Operand/instruction forms

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$\mathsf{R}[\mathtt{r}_a]$	Register
Memory	Imm	$\mathsf{M}[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

1.2.6 Mov instructions

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
`movb`			Move byte
`movw`			Move word
`movl`			Move double word
`movq`			Move quad word
`movabsq`	I, R	$R \leftarrow I$	Move absolute quad word

Figure 3.4 Simple data movement instructions.

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow$ ZeroExtend(S)	Move with zero extension
`movzbw`		Move zero-extended byte to word
`movzbl`		Move zero-extended byte to double word
`movzwl`		Move zero-extended word to double word
`movzbq`		Move zero-extended byte to quad word
`movzwq`		Move zero-extended word to quad word

Figure 3.5 Zero-extending data movement instructions. These instructions have a register or memory location as the source and a register as the destination.

Instruction	Effect	Description
MOVS S, R	$R \leftarrow$ SignExtend(S)	Move with sign extension
`movsbw`		Move sign-extended byte to word
`movsbl`		Move sign-extended byte to double word
`movswl`		Move sign-extended word to double word
`movsbq`		Move sign-extended byte to quad word
`movswq`		Move sign-extended word to quad word
`movslq`		Move sign-extended double word to quad word
`cltq`	`%rax` $\leftarrow$ SignExtend(`%eax`)	Sign-extend `%eax` to `%rax`

Figure 3.6 Sign-extending data movement instructions. The MOVS instructions have a register or memory location as the source and a register as the destination. The `cltq` instruction is specific to registers `%eax` and `%rax`.

Instruction		Synonym	Move condition	Description
`cmove`	S, R	`cmovz`	`ZF`	Equal / zero
`cmovne`	S, R	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	S, R		`SF`	Negative
`cmovns`	S, R		`~SF`	Nonnegative
`cmovg`	S, R	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	S, R	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	S, R	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	S, R	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	S, R	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	S, R	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	S, R	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	S, R	`cmovna`	`CF | ZF`	Below or equal (unsigned <=)

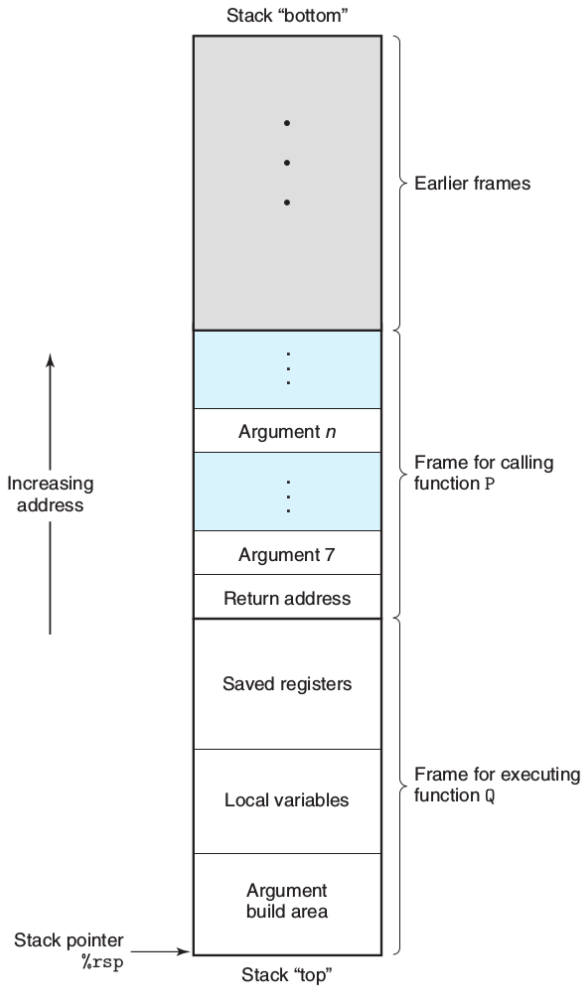
Figure 3.18 The conditional move instructions. These instructions copy the source value S to its destination R when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

1.2.7 Stack and push/pop

Instruction	Effect	Description
`pushq` S	R[%rsp] $\leftarrow$ R[%rsp] $-$ 8; M[R[%rsp]] $\leftarrow$ S	Push quad word
`popq` D	$D \leftarrow$ M[R[%rsp]]; R[%rsp] $\leftarrow$ R[%rsp] $+$ 8	Pop quad word

Figure 3.8 Push and pop instructions.

Figure 3.25
General stack frame structure. The stack can be used for passing arguments, for storing return information, for saving registers, and for local storage. Portions may be omitted when not needed.



1.2.8 Arithmetic instructions

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_{\mathrm{A}} k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_{\mathrm{L}} k$	Logical right shift

Figure 3.10 Integer arithmetic operations. The load effective address (`leaq`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $>>_{\mathrm{A}}$ and $>>_{\mathrm{L}}$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Note: logical right shift always appends zeros while arithmetric right shift always append the sign bit. That is the arithmetic one works for signed types, while logical works for positive numbers or unsigned types.
Note: using xor instruction on a reg/mem spot with itself, sets the value to 0 (because all digits are the same). This instruction uses less bytes to compute result than setting the reg/mem spot to 0 via mov $0, [reg/mem spot].

Instruction		Effect	Description
`imulq`	S	R[%rdx]:R[%rax] $\leftarrow$ $S \times$ R[%rax]	Signed full multiply
`mulq`	S	R[%rdx]:R[%rax] $\leftarrow$ $S \times$ R[%rax]	Unsigned full multiply
`cqto`		R[%rdx]:R[%rax] $\leftarrow$ SignExtend(R[%rax])	Convert to oct word
`idivq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Signed divide
`divq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Unsigned divide

Figure 3.12 Special arithmetic operations. These operations provide full 128-bit multiplication and division, for both signed and unsigned numbers. The pair of registers `%rdx` and `%rax` are viewed as forming a single 128-bit oct word.

1.2.9 Flags

`CF`: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

`ZF`: Zero flag. The most recent operation yielded zero.

`SF`: Sign flag. The most recent operation yielded a negative value.

`OF`: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

For example, suppose we used one of the ADD instructions to perform the equivalent of the C assignment `t = a+b`, where variables `a`, `b`, and `t` are integers. Then the condition codes would be set according to the following C expressions:

`CF`	`(unsigned) t < (unsigned) a`	Unsigned overflow
`ZF`	`(t == 0)`	Zero
`SF`	`(t < 0)`	Negative
`OF`	`(a < 0 == b < 0) && (t < 0 != a < 0)`	Signed overflow

1.2.10 Compare instructions (cmp)

Instruction		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word
TEST	S_1, S_2	S_1 & S_2	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

Figure 3.13 Comparison and test instructions. These instructions set the condition codes without updating any other registers.

1.2.11 Set instructions (basically compute a boolean - set flags)

Instruction		Synonym	Effect	Set condition
`sete`	*D*	`setz`	*D* ← `ZF`	Equal / zero
`setne`	*D*	`setnz`	*D* ← `~ZF`	Not equal / not zero
`sets`	*D*		*D* ← `SF`	Negative
`setns`	*D*		*D* ← `~SF`	Nonnegative
`setg`	*D*	`setnle`	*D* ← `~(SF^OF)&~ZF`	Greater (signed >)
`setge`	*D*	`setnl`	*D* ← `~(SF^OF)`	Greater or equal (signed >=)
`setl`	*D*	`setnge`	*D* ← `SF^OF`	Less (signed <)
`setle`	*D*	`setng`	*D* ← `(SF^OF)|ZF`	Less or equal (signed <=)
`seta`	*D*	`setnbe`	*D* ← `~CF&~ZF`	Above (unsigned >)
`setae`	*D*	`setnb`	*D* ← `~CF`	Above or equal (unsigned >=)
`setb`	*D*	`setnae`	*D* ← `CF`	Below (unsigned <)
`setbe`	*D*	`setna`	*D* ← `CF|ZF`	Below or equal (unsigned <=)

Figure 3.14 The SET instructions. Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate names for the same machine instruction.

1.2.12 Jump instructions (jmp)

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Figure 3.15 **The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Example below:

```
cmp a, b
jle some_location      ;    if a ≤ b, jump to some_location
```

1.2.13 x86prime

- `MOVQ %ra, %rb`: kopiering fra et register til et andet,
- `MOVQ $imm, %rb`: initialisering af register,
- `MOVQ $imm(%rb), %ra`: læsning fra lageret,
- `MOVQ %ra, $Imm(%rb)`: skrivning til lageret,
- `ADDQ/SUBQ/ANDQ/ORQ/XORQ/MULQ %ra, %rb`: aritmetik på registre,
- `ADDQ/SUBQ/ANDQ/ORQ/XORQ/MULQ $imm, %rb`: aritmetik med heltal,
- `JMP target`: ubetinget hop,
- `LEAQ $imm(%rs,%rz,$scale), %ra`.
- `CALL target, %ra`: underprogramkald som gemmer returadressen i `%ra`,
- `RET %ra`: retur fra underprogramkald som læser returadressen fra `%ra`,
- `CBcc %ra, %rb, target`: hop hvis relationen `%ra` *cc* `%rb` er opfyldt,
- `CBcc $imm, %rb, target`: hop hvis relationen `$imm` *cc* `%rb` er opfyldt.

Figure 1.1: x86-prime

1.2.14 Assembly to C examples

(a) Original C code

```
long absdiff(long x, long y)
{
    long result;
    if (x < y)
        result = y - x;
    else
        result = x - y;
    return result;
}
```

(b) Implementation using conditional assignment

```
1   long cmovdiff(long x, long y)
2   {
3       long rval = y-x;
4       long eval = x-y;
5       long ntest = x >= y;
6       /* Line below requires
7          single instruction: */
8       if (ntest) rval = eval;
9       return rval;
10  }
```

(c) Generated assembly code

```
    long absdiff(long x, long y)
    x in %rdi, y in %rsi
1   absdiff:
2     movq    %rsi, %rax
3     subq    %rdi, %rax        rval = y-x
4     movq    %rdi, %rdx
5     subq    %rsi, %rdx        eval = x-y
6     cmpq    %rsi, %rdi        Compare x:y
7     cmovge  %rdx, %rax        If >=, rval = eval
8     ret                       Return tval
```

Figure 3.17 Compilation of conditional statements using conditional assignment. (a) C function `absdiff` contains a conditional expression. The generated assembly code is shown (c), along with (b) a C function `cmovdiff` that mimics the operation of the assembly code.

Figure 1.2: if statement

(a) C code

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

(b) Equivalent goto version

```
long fact_do_goto(long n)
{
    long result = 1;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
    long fact_do(long n)
    n in %rdi
1   fact_do:
2     movl    $1, %eax       Set result = 1
3   .L2:                   loop:
4     imulq   %rdi, %rax     Compute result *= n
5     subq    $1, %rdi       Decrement n
6     cmpq    $1, %rdi       Compare n:1
7     jg      .L2            If >, goto loop
8     rep; ret               Return
```

Figure 3.19 Code for `do-while` version of factorial program. A conditional jump causes the program to loop.

Figure 1.3: do while loop

(a) C code

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Equivalent goto version

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
 loop:
    result *= n;
    n = n-1;
 test:
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

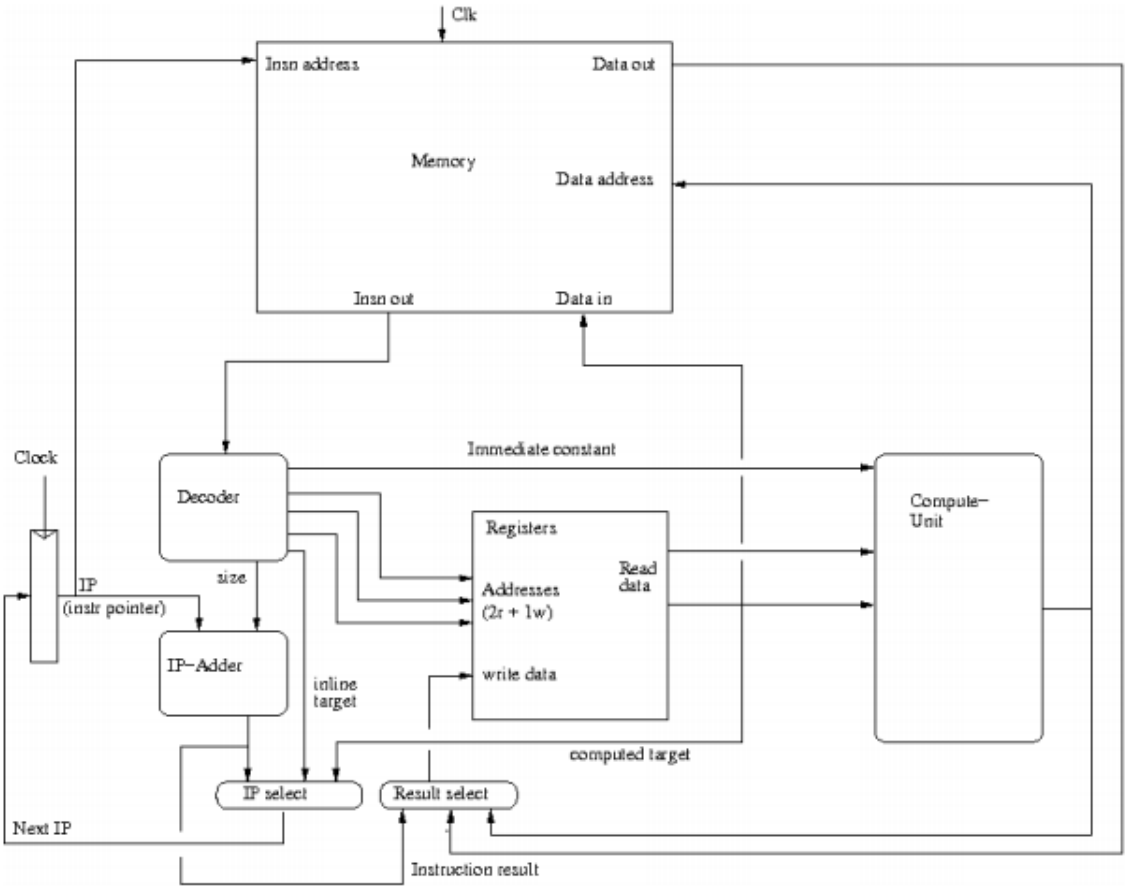
```
  long fact_while(long n)
  n in %rdi
fact_while:
  movl    $1, %eax       Set result = 1
  jmp     .L5            Goto test
.L6:                   loop:
  imulq   %rdi, %rax     Compute result *= n
  subq    $1, %rdi       Decrement n
.L5:                   test:
  cmpq    $1, %rdi       Compare n:1
  jg      .L6            If >, goto loop
  rep; ret               Return
```

Figure 3.20 C and assembly code for `while` version of factorial using jump-to-middle translation. The C function `fact_while_jm_goto` illustrates the operation of the assembly-code version.

Figure 1.4: while loop

1.3 Micro Architecture

1.3.1 Micro architecture diagram



Explanations:

1. IP: Instruction Pointer
   - pointer to an instruction
   - the same as PC (program counter) and instruction address register (IAR)
2. Clock
   -
3. Multiplexer

- In electronics, a multiplexer (or mux) is a device that combines several analog or digital input signals and forwards them into a single output line.

- A multiplexer of 2^n inputs has n select lines, which are used to select which input line to send to the output

1.3.2 Pipelining

1.3.2.1 Hazards

1.3.2.1.1 Structural hazards

Occurs when multiple instructions wants to use the same hardware part at once.

Example: The memory unit is accessed both in the fetch stage, F, where an instruction is retrieved from memory, and the memory stage, M, where data is written and/or read from memory.

1.3.2.1.2 Data hazards

Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (also termed race hazards).
Data hazards can happen if:

1. read after write (RAW), a true dependency
    - An instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a prior instruction, the prior instruction has been processed only partly through the pipeline.
2. write after read (WAR), an anti-dependency
    - Because you have to ensure that an instruction does not change a value K before any previous instructions are done using the old value of K.
3. write after write (WAW), an output dependency
    - Similar to WAR, but here the previous instruction also changes K.

1.3.2.1.3 Control hazards (branching hazards)

Occur with branches as the pipe essentially stall when computing a condition because this is not actually useful work. The amount of control hazards can be lowered using branch prediction. The stall is called branch delay, and if it is not dealt with it is a branch penalty.

1.3.2.2 Steps of pipelining

I x86prime kan der kun udføres aritmetiske beregninger mellem registre og mellem immediates(f.eks. tallet 7) og registre.
Pipelining går ud på at arrangere udførelsen af instruktioner som et samlebånd. Ved pipelining adskiller man hvert trin med registre - så kan trinnene udføres samtidigt, men med forskellige instruktioner i hvert sit trin.
Forklaring fra Assignment 3: (x86prime pipeline)
F: Fetch, instruktionhentning
D: Decode, afkodning, læsning af registre, evt venten på operander
X: eXecute, udførelse af ALU op, af adresseberegning eller af første del af multiplikation
M: Memory, læsning/skrivning af data fra data-cache, midterste del af multiplikation
Y: sidste del af multiplikation
W: Writeback, opdatering af registre

Den klassiske pipeline som man ofte ser i lærebøgerne har 5 trinSee page 420 and 421 (for books interpretation):

'F' for "fetch" - hentning af instruktion
'D' for "decode" - afkodning
'X' for "execute" - udførelse (ALU: aritmetiske beregninger), resolving correctness of a predicted jump
'M' for "memory" - læsning fra eller skriving til lageret
'W' for "writeback" - opdatering af registre

1.3.2.3 Stalling the pipeline

Hvis en instruktion skal bruge resultatet af den forrige ville vi gerne kunne stalle pipelinen. Eksempel:
(obs: insn er en eller anden instruction)

```
insn              FDXMW
movq %r11, %r14    FDXMW
addq %r14, %r17     FDXMW (FORKERT!)
insn                 FDXMW
```

Ovenstående pipelining ville ikke fungere, da instruktionen addq i dette tilfælde skal bruge resultatet fra movq på register %r14. Resultatet bliver nemlig først tilgengælig ved writeback (W). Løsningen ved dette er at stalle pipelinen.

```
insn              FDXMW
movq %r11, %r14    FDXMW
addq %r14, %r17     FDDDDXMW (KORREKT!)
insn                 FFFFDXMW
```

OBS: dette er UDEN forwarding. Med fuld forwarding kan man Decode i addq allerede under movq Write, da movq Memory forwarder %r14 til addq.
En anden løsning er at lave software som finder uafhængige instruktioner(i modsætning til addq i dette tilfælde), og indsætte dem der hvor der ellers ville blive stallet.

1.3.2.4 Shadow / Delay-slot

Det, som er beskrevet i OBS foroven, kaldes for et delay slot eller en "shadow" af den forgående instruktion.)
Ved at fylde skyggen med uafhængige instruktioner, er der ikke noget tidspunkt hvor pipelinen bremses, og den udnyttes derfor mere effektivt. Det er dog svært for compileren at finde uafhængige instruktioner.

1.3.2.5 Bypassing / Forwarding

Vi tilføjer dedikerede forbindelser fra resultatsiden af ALUen til der hvor værdierne skal bruges.

```
insn            FDXMW
movq %r11,%r14   FDXMW
addq %r14,%r17    FDXMW
insn               FDXMW
```

Det kaldes "bypassing" eller "forwarding". Det vil koste lidt på clockfrekvensen i forhold til hvis man ikke har bypassing. Men gevinsten er stor.
Moderne maskiner løser alle data afhængigheder i hardware.

1.4 Memory

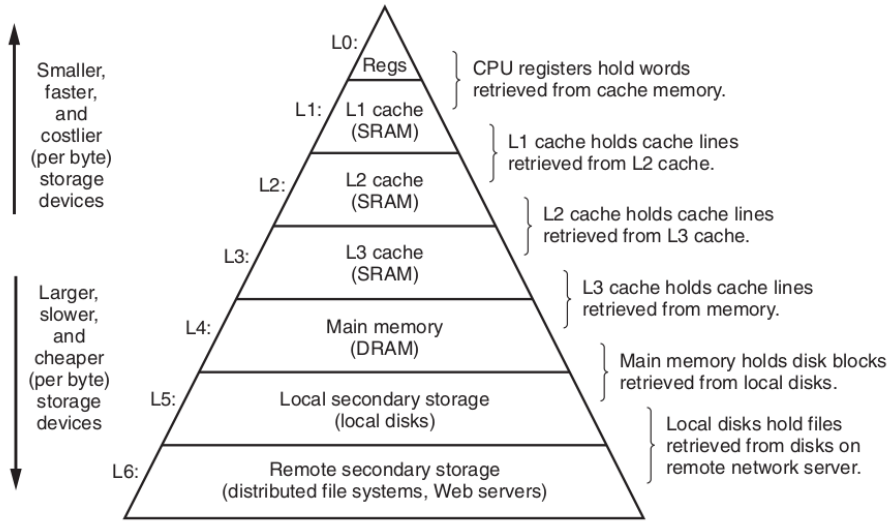


Figure 1.9 **An example of a memory hierarchy.**

1.4.1 SRAM and DRAM: Key differences

SRAM: Static random access memory DRAM: Dynamic random access memory
RAM is volatile memory - the content is lost when the power is gone.

1. SRAM is an on-chip memory whose access time is small while DRAM is an off-chip memory which has a large access time. Therefore SRAM is faster than DRAM.

2. DRAM is available in larger storage capacity while SRAM is of smaller size. SRAM is expensive whereas DRAM is cheap.

3. The cache memory is an application of SRAM. In contrast, DRAM is used in main memory.

4. DRAM is highly dense. As against, SRAM is rarer.

5. The construction of SRAM is complex due to the usage of a large number of transistors. On the contrary, DRAM is simple to design and implement.

6. In SRAM a single block of memory requires six transistors whereas DRAM needs just one transistor for a single block of memory.

7. DRAM is named as dynamic, because it uses capacitor which produces leakage current due to the dielectric used inside the capacitor to separate the conductive plates is not a perfect insulator hence require power refresh

circuitry. On the other hand, there is no issue of charge leakage in the SRAM.

8. Power consumption is higher in DRAM than SRAM. SRAM operates on the principle of changing the direction of current through switches whereas DRAM works on holding the charges.

1.4.2 Disk Storage

1.4.2.1 Abbreviations

Rotational latency: Time it takes for the disk to rotate so that the head can read the first bit of the target sector.
Maximum Rotational latency is the time it takes for the disk to do an entire rotation. Average is half of that.

1.4.2.2 Formulas

BOH Side 630
$Capacity = \frac{\#bytes}{sector} \cdot \frac{average\ \#sectors}{track} \cdot \frac{\#tracks}{surface} \cdot \frac{\#surfaces}{platter} \cdot \frac{\#platters}{disk}$

$T_{max\ rotation\ latency} = \frac{1}{RPM} \cdot \frac{60sec}{1min}$

$T_{avg\ rotation\ latency} = \frac{T_{max\ rotation\ latency}}{2} \cdot transfer\ time$

$T_{avg\ transfer} = \frac{1}{RPM} \cdot \frac{1}{average\#per\ track} \cdot \frac{60secs}{1min}$

1.4.3 Locality

1.4.3.1 Spatial locality

Items with nearby addresses tend to be referenced close together in time.
Data Example: Reference array elements in succession **(stride-1 reference pattern)**
Instruction Example: Reference instructions in sequence

1.4.3.2 Temporal locality

Recently referenced items are likely to be referenced in the near future.
Data Example: Reference the same variable in each iteration of a loop.
Instruction Example: Cycle through loop repeatedly

1.4.3.3 Stride-n reference patterns

Referencing every n'th element of a contiguous array is called a stride-n reference pattern.

In an array[i][j] = A[10][5], a loop that references the first element of each column = [i][0] would be stride-5, since it skips 5 elements every time it references the array.

Stride-1 reference patterns referred to as sequential reference patterns
To create a stride-1 reference pattern, the loops must be permuted so that the rightmost indices change most rapidly, Example:

```
1   int productarray3d(int a[N][N][N])
2   {
3       int i, j, k, product = 1;
4
5       for (j = N-1; j >= 0; j--) {
6           for (k = N-1; k >= 0; k--) {
7               for (i = N-1; i >= 0; i--) {
8                   product *= a[j][k][i];
9               }
10          }
11      }
12      return product;
13  }
```

1.5 Cache

1.5.1 Cache interface

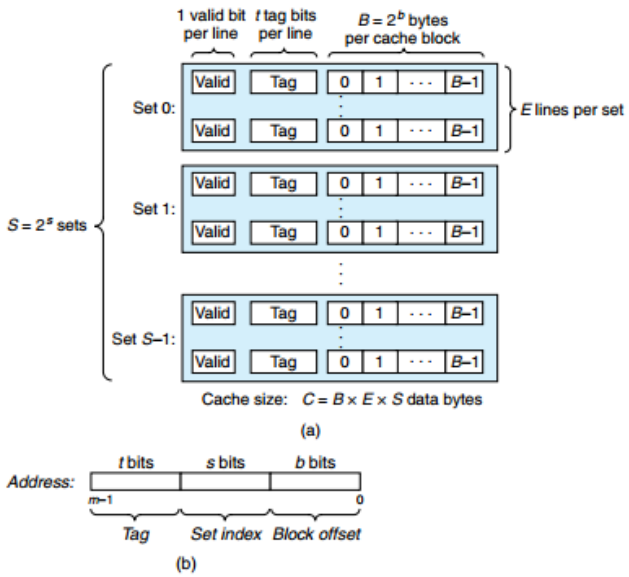


Figure 6.25
General organization of cache (S, E, B, m). (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.

Figure 1.5

1.5.2 Cache mapping

1.5.2.1 Address structure

Checking for hit or miss: First, the valid bit must be set - otherwise it is a miss.
In directly mapped caches:
Structure: In a directly mapped cache, there is only 1 line per set.
In set-associative caches:
Structure: [Tag bits][Set index][Block offset]
First check which set the address belongs to, then check among all the lines in the set whether there is a matching TAG. Ignore the OFFSET bits, as they do not dictate whether there is a hit or miss.

1.5.3 Exponents of 2

Useful if you cannot use scripts for calculating log_2 (e.g. for block offset, cache tag and set index).

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024
2^{11}	2048
2^{12}	4096

Table 1.1: Exponents of 2

1.5.4 Abbreviations

1. VPO: virtual page offset
2. PPO: physical page offset. Is the same number as VPO

1.5.5 Formulars

Fundamental parameters	
Parameter	Description
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits

Derived quantities	
Parameter	Description
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes) not including overhead such as the valid and tag bits

Figure 6.28 **Summary of cache parameters.**

Figure 1.6

The number of sets:

$$S = \frac{C}{B \cdot E}$$

where E is number of lines in a set (ways in associative set cache), C is cache size and B is block size.

1.5.6 Replacement strategies

1.5.6.1 First in first out (FIFO)

First block in is deleted first.

1.5.6.2 Last in first out (LIFO)

Last block in is deleted last.

1.5.6.3 Least recently used (LRU)

The last used block is deleted first.

1.5.6.4 Most recently used (MRU)

The latest used block is deleted first.

1.5.6.5 Random replacement (RR)

A random block is deleted.

1.5.7 Write strategies (storage methods)

1. Write-through: Data is written into the cache and the corresponding main memory location at the same time. The cached data allows for fast retrieval on demand, while the same data in main memory ensures that nothing will get lost if a crash, power failure, or other system disruption occurs.

2. Write-back: Data is written into the cache every time a change occurs, but is written into the corresponding location in main memory only at specified intervals or under certain conditions.

Chapter 2

OS

2.1 Kernel vs process responsibilities

Kernel

1. disk management
2. process and task management (process management for application execution)
3. memory management, allocation and I/O
    - including virtual address translation
4. device management (interaction with devices, both hardware and software)
5. exception handling (not sure???)
6. context switching
7. system call control

Process

1.

2.2 Control Flow

2.2.1 Exceptional Control Flow

- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 4. **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - Implemented by C runtime library

Figure 2.1

2.2.2 Program state

The control flow reacts to changes in the program state as e.g. jumps, branches, calls, and returns.

2.2.3 System state

It is harder for the control flow to react on changes in the system state.
Examples:

1. Data arrives from a disk or a network adapter (interrupt[1], asynchronous exception[2])
2. Instruction divides by zero (fault, synchronous exception)
3. User hits Ctrl-C at the keyboard (interrupt, asynchronous exception)
4. System timer expires (asynchronous exception)

[1] See Figure 2.4
[2] See subsection 2.3.1

2.3 Exceptions

An exception is a transfer of control to the kernel in response to some event (i.e., change in processor state).

Figure 8.1
Anatomy of an exception. A change in the processor's state (an event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

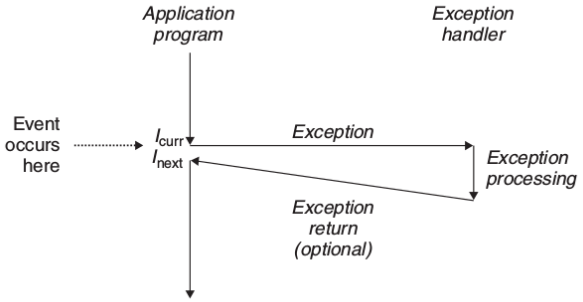


Figure 2.2

Figure 8.2
Exception table. The exception table is a jump table where entry k contains the address of the handler code for exception k.

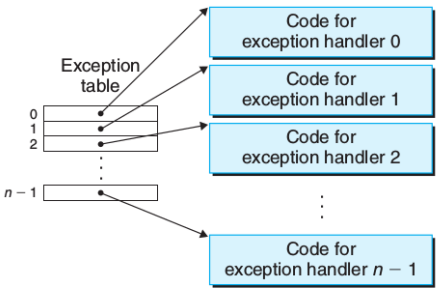


Figure 8.3
Generating the address of an exception handler. The exception number is an index into the exception table.

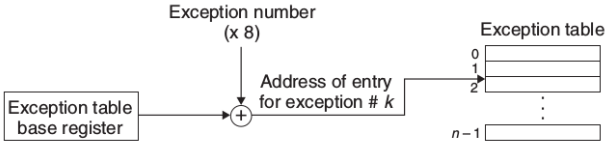


Figure 2.3

2.3.1 Synchronous vs asynchronous exceptions

Asynchronous exception: caused by something external to the process. There might not always be information on what caused it.
Synchronous exception: caused by something internal to the process. There is

information on what caused it. Originated only as a result of execution where the exception is thrown.

See Figure 2.4 for specification of which exceptions are synchronous or asynchronous.

2.3.2 Exception classes

Class	Cause	Async/sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4 **Classes of exceptions.** Asynchronous exceptions occur as a result of events in I/O devices that are external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

Figure 2.4

Figure 8.5
Interrupt handling. The interrupt handler returns control to the next instruction in the application program's control flow.

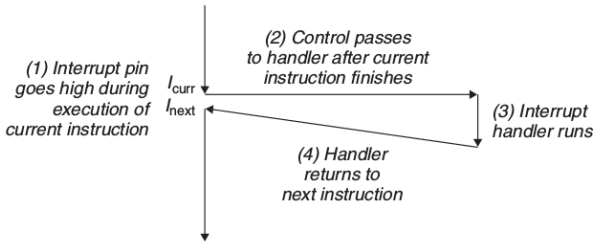


Figure 8.6
Trap handling. The trap handler returns control to the next instruction in the application program's control flow.

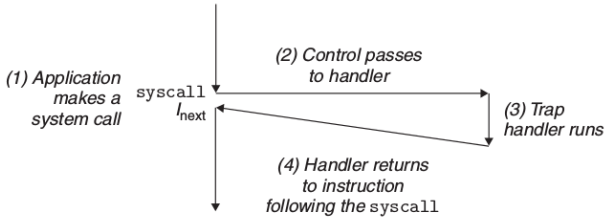


Figure 8.7
Fault handling. Depending on whether the fault can be repaired or not, the fault handler either re-executes the faulting instruction or aborts.

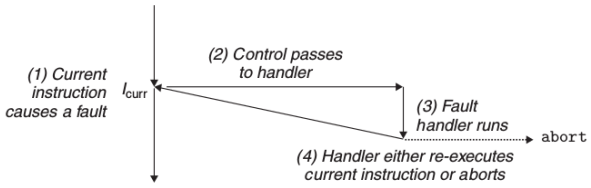


Figure 8.8
Abort handling. The abort handler passes control to a kernel `abort` routine that terminates the application program.

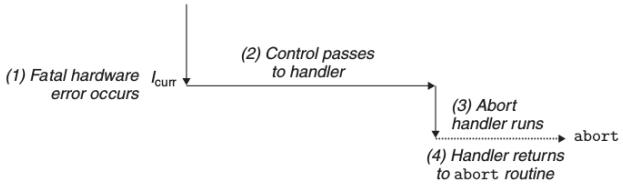


Figure 2.5

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–255	OS-defined exceptions	Interrupt or trap

Figure 8.9 Examples of exceptions in x86-64 systems.

Figure 2.6

2.3.3 Interrupts

Interrupts occur *asynchronously* as a result of signals from I/O devices that are external to the processor. Hardware interrupts are asynchronous in the sense that they are not caused by the execution of any particular instruction. Exception handlers for hardware interrupts are often called *interrupt handlers*.

2.3.4 Traps and syscalls

Traps are *intentional* exceptions that occur as a result of executing an instruction. Like interrupt handlers, trap handlers return control to the next instruction. The most important use of traps is to provide a procedure-like interface between user programs and the kernel, known as a *system call*.

System calls are in kernel mode. Common sys calls are read, write, fork and execve. From the programmers perspective a sys calls are identical to a function call.

Number	Name	Description	Number	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

Figure 8.10 **Examples of popular system calls in Linux x86-64 systems.**

Figure 2.7

2.3.5 Faults

Faults result from error conditions that a handler might be able to correct. When a fault occurs, the processor transfers control to the fault handler. If the handler is able to correct the error condition, it returns control to the faulting instruction, thereby re-executing it. Otherwise, the handler returns to an `abort` routine in the kernel that terminates the application program that caused the fault. Figure 8.7 summarizes the processing for a fault.

Figure 2.8

A classic example of a fault is the page fault exception.

2.3.6 Aborts

Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted. Abort handlers never return control to the application program. As shown in Figure 8.8, the handler returns control to an `abort` routine that terminates the application program.

Figure 2.9

2.4 Process

2.4.1 Short overview

A process is an instance of a program in execution (abstraction). Key abstractions:

1. An independent logical control flow that provides the illusion that our program has exclusive use of the processor.
2. A private address space that provides the illusion that our program has exclusive use of the memory system.

Process can be terminanted by using `void exit(status)`, so the process terminates with an exit status of `status`. exit is called once but never returns.

2.4.2 Process states

1. Running: Process is either executing, or waiting to be executed and will eventually be scheduled (i.e., chosen to execute) by the kernel
2. Stopped/Suspended: Process execution is suspended and will not be scheduled until further notice. The process may or may not be waiting on an event.
3. Terminated: Process is stopped permanently

2.4.3 Process and kernel context

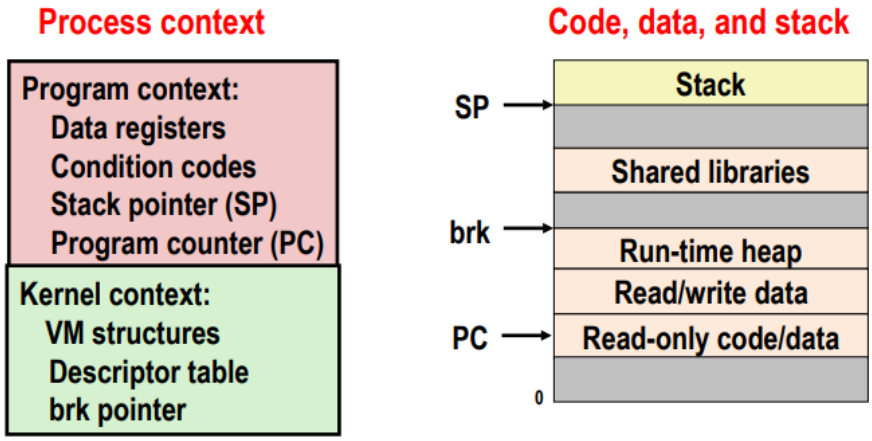


Figure 2.10

2.4.4 Context switch

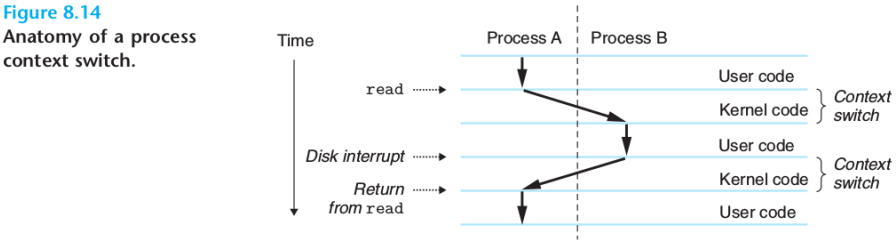


Figure 2.11

2.5 Fork

See man-page fork(2) for more info.

2.5.1 Short overview

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process. The child process and the parent process run in separate memory

spaces. At the time of fork() both memory spaces have the same content.
fork() is called once but returns twice.

2.5.2 Remarks

1. Fork returns -1 on failure, 0 if child, 0> for parent (the parent gets the ID of the child).
2. The new process gets a copy of the parent's virtual address space.
   - Mutexs are just variables so they are a part of this virtual address space.
3. Threads are not copied into the new process.
   - OBS: the states of mutexes, condition variables, and other pthreads objects are part of the parent's virtual address space, and so they are inherited to the child! This may cause deadlocks.
     - print uses a buffer (a variable) before actually printing, so this buffer may be inherited and the child will print too.
     - if a mutex is locked and not unlocked before child is created, the mutex may never be unlocked.
4. File descriptors are also inherited, including open file status flags, current file offset, and signal-driven I/O attributes
5. Remember to free malloc()ed memory in the child too (it should be freed in both the parent and child).

2.5.2.1 fork() penalty (only Linux)

From man-page:
Under Linux, fork() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.
This basically says that on fork()ing, no actual copying is done, but the pages are marked to be copied if the child tries to modify them. Effectively, if the child only reads from that memory, or completely ignore it, there is no copy overhead. This is very important for the common fork()/exec() pattern.

2.5.3 Reaping children

- **Idea**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a "zombie"
    - Living corpse, half alive and half dead
- **Reaping**
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- **What if parent doesn't reap?**
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Figure 2.12

2.6 Signals

Figure 8.27
Signal handling. Receipt of a signal triggers a control transfer to a signal handler. After it finishes processing, the handler returns control to the interrupted program.

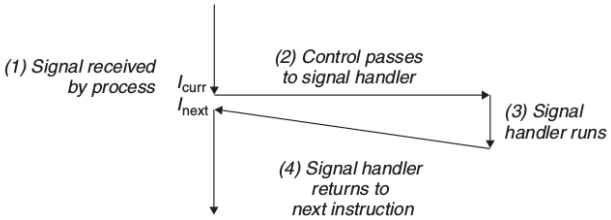


Figure 2.13

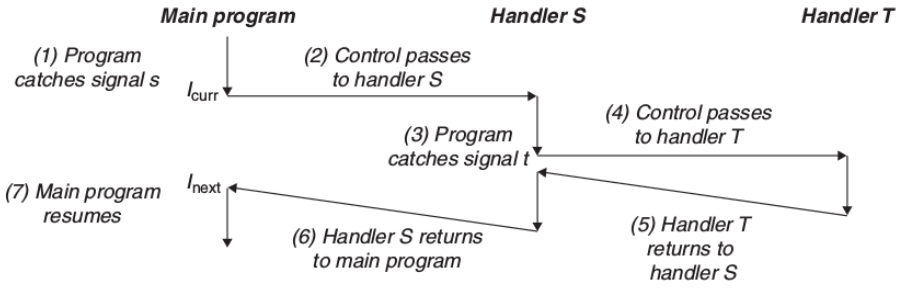


Figure 8.31 **Handlers can be interrupted by other handlers.**

Figure 2.14

2.6.1 Different signals

Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core [a]	Trace trap
6	SIGABRT	Terminate and dump core [a]	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core [a]	Floating-point exception
9	SIGKILL	Terminate [b]	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core [a]	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT [b]	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	Stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	Stop until next SIGCONT	Background process wrote to terminal
23	SIGURG	Ignore	Urgent condition on socket
24	SIGXCPU	Terminate	CPU time limit exceeded
25	SIGXFSZ	Terminate	File size limit exceeded
26	SIGVTALRM	Terminate	Virtual timer expired
27	SIGPROF	Terminate	Profiling timer expired
28	SIGWINCH	Ignore	Window size changed
29	SIGIO	Terminate	I/O now possible on a descriptor
30	SIGPWR	Terminate	Power failure

Figure 8.26 **Linux signals.** *Notes:* (a) Years ago, main memory was implemented with a technology known as *core memory*. "Dumping core" is a historical term that means writing an image of the code and data memory segments to disk. (b) This signal can be neither caught nor ignored. (Source: `man 7 signal`. Data from the Linux Foundation.)

Figure 2.15

2.6.2 Block signals

Linux provides implicit and explicit mechanisms for blocking signals:

Implicit blocking mechanism. By default, the kernel blocks any pending signals of the type currently being processed by a handler. For example, in Figure 8.31, suppose the program has caught signal s and is currently running handler S. If another signal s is sent to the process, then s will become pending but will not be received until after handler S returns.

Explicit blocking mechanism. Applications can explicitly block and unblock selected signals using the `sigprocmask` function and its helpers.

Figure 2.16

2.7 Threads

2.7.1 Remarks

1. context switched, just like processes
2. thread context: Thread ID, stack, stack pointer, PC, condition codes, and GP registers
    - the remaining process context is shared between the thread: Code, data, heap, and shared library segments of the process virtual address space. Open files and installed handlers.

2.7.2 Races, Deadlock and Livelocks

1. Races: outcome depends on arbitrary scheduling decisions elsewhere in the system
2. Deadlock: improper resource allocation prevents forward progress
3. Livelock / Starvation / Fairness: external events and/or system scheduling decisions can prevent sub-task progress

2.7.3 Speed up when using threads

A related measure, known as *efficiency*, is defined as

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

Figure 2.17

Threads (t)	1	2	4	8	16
Cores (p)	1	2	4	4	4
Running time (T_p)	1.06	0.54	0.28	0.29	0.30
Speedup (S_p)	1	1.9	3.8	3.7	3.5
Efficiency (E_p)	100%	98%	95%	91%	88%

Figure 12.36 Speedup and parallel efficiency for the execution times in Figure 12.35.

Figure 2.18

2.7.4 Atomicity

Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a succeed-or-fail definition — they either successfully change the state of the system, or have no apparent effect. That is atomic functions doesn't need mutexes while non-atomic functions need mutexes if multiple thread

2.8 Virtual memory

Page is what you want to store and page frame is where you want to store.

Page - logical addresses (addresses referred in a code / program)

Page frame - physical addresses (addresses present in RAM / primary memory)

2.8.1 Abbreviations

1. PP: Physical Page
2. PTE: Page Table Entry

- Liste som indeholder mappings fra virtual memory addresses til physical memory addresses. (tror jeg)

3. VP: Virtual Page
   - Missing explanation
4. TLB: Translation lookaside buffer
   - The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache.
5. MMU: memory management unit
   - translating between VM and PM
   - when a process seeks info on a PP, MMU checks if allowed to do the operation on the PP using permission bits (see PTE permissions)
6. PTBR: Page table base register
   - holds the base address for the page table of the current process.

2.8.2 PTE permissions

Permission bits:

1. SUP: set if only kernel can access, unset if user processes can access
2. READ
3. WRITE
4. EXEC: if permitted to execute from entry

2.8.3 Address translation symbols

1. Basic Parameters
   - N = 2^n : Number of addresses in virtual address space
   - M = 2^m : Number of addresses in physical address space
   - P = 2^p : Page size (bytes)
2. Components of the virtual address (VA)
   - TLBI: TLB index
   - TLBT: TLB tag
   - VPO: Virtual page offset
   - VPN: Virtual page number
3. Components of the physical address (PA)
   - PPO: Physical page offset (same as VPO)
   - PPN: Physical page number

2.8.4 Address Translation

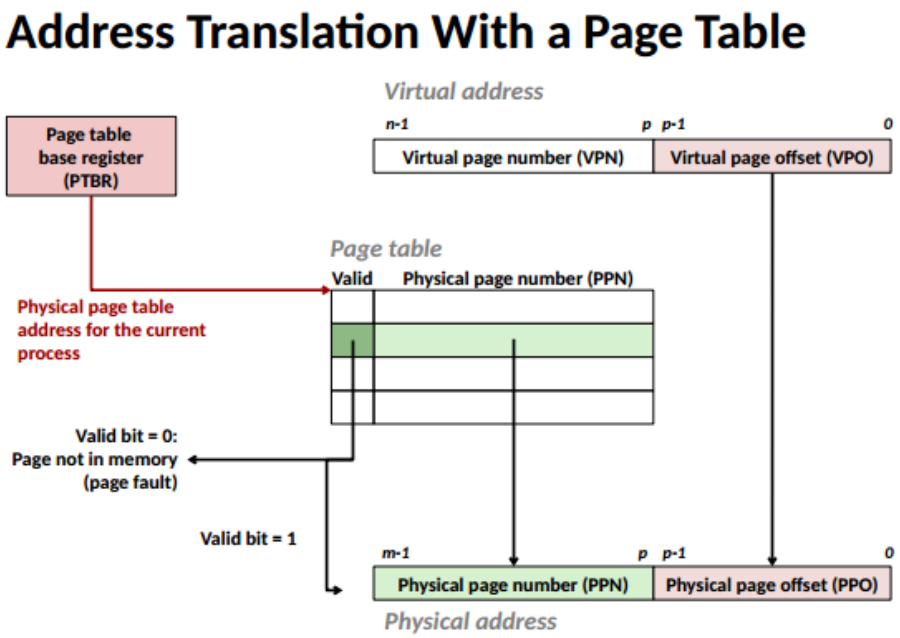


Figure 2.19

Start by seeing if the correct page is in the TLB if not look in the page table. There is only page fault if there if there is miss in both TLB and page table. Remember to check the valid-bit as Troels will probably try to trick us with an invalid entry. Once the PPN is found in either TLB or page table, the physical address is found by concatenating PPN and VPO (PPN first).

2.8.5 Fragmentation in memory

BOH 9.9.4 (Side 882)

2.8.5.1 Internal fragmentation

Internal fragmentation occurs when a process is allocated more memory than required, leaving some space unused. This occurs when memory is divided into fixed-sized partitions.
Internal fragmentation can be removed by allocating memory dynamically or having partitions of different sizes.

2.8.5.2 External fragmentation

External fragmentation is the various free spaced holes that are generated in either your memory or disk space.
These occur when a dynamic memory allocation algorithm allocates some memory, and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced.
Internal fragmentation(menes der ikke external?) can be solved by using virtual memory addressing with its paging and segmentation.
Another solution is Compaction: shuffling the fragmented memory into one contiguous location.

2.9 Memory allocaters

A block has a header and a footer where header=footer.

Implicit free list: the header/footer includes the size of the block which is used to locate where the next or previous block begins.
OBS: Blocksize includes both the data blocks AND the header and footer!

Explicit free list: uses pointers
Remember:

1. the heap starts from the buttom so the previous block is the block under it!

2. if header/footer includes whether the previous is allocated, then remember to change this bit of the next block when you free something.

immediate coalescing: adjacent free blocks are merged immediately each time a block is freed.

2.9.1 Advantages and Disadvantages

Implicit:

1. the implicit free list is not appropriate for a generalpurpose allocator (although it might be fine for a special-purpose allocator where the number of heap blocks is known beforehand to be small).

2.9.2 malloc()

The malloc function returns a pointer to a block of memory of at least size bytes that is suitably aligned for any kind of data object that might be contained in the block.
If malloc encounters a problem (e.g., the program requests a block of memory

that is larger than the available virtual memory), then it returns NULL and sets errno. Malloc does not initialize the memory it returns.

2.9.3 Remarks

Remarks:

1. In 32-bit mode, malloc returns a block whose address is always a multiple of 8. In 64-bit mode, the address is always a multiple of 16.

2.9.4 Other allocation functions

Applications that want initialized dynamic memory can use calloc, a thin wrapper around the malloc function that initializes the allocated memory to zero. Applications that want to change the size of a previously allocated block can use the realloc function.

Chapter 3

Network

3.1 UDP: User Datagram Protocol

The UDP protocol provides a connectionless service to its applications. This is a no-frills service that provides no reliability, no flow control, and no congestion control. You basically just send the data and hope that the receiver is able to receive it.

3.2 TCP

TCP includes guaranteed delivery of data and flow control (that is, sender/receiver speed matching). TCP also breaks long messages into shorter segments and provides a congestion-control mechanism, so that a source throttles its transmission rate when the network is congested.

3.2.1 TCP segment structure/format (fields)

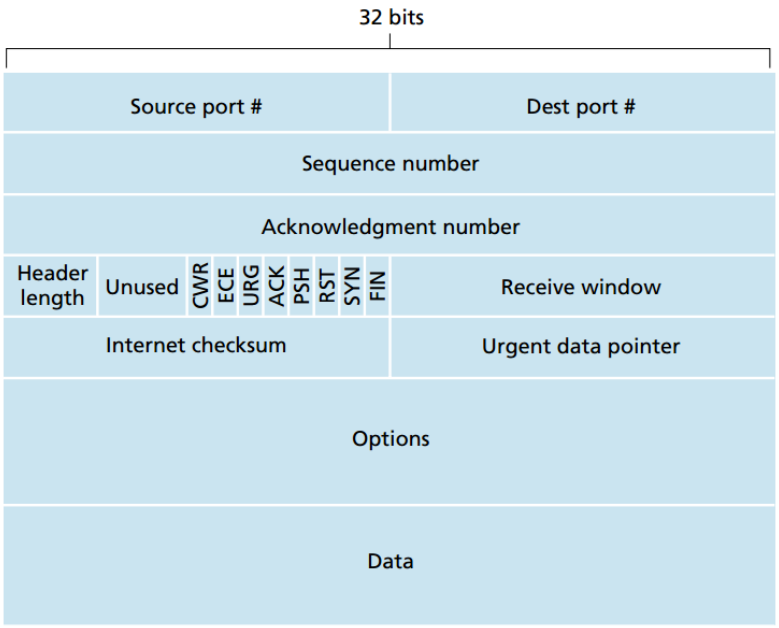


Figure 3.29 ♦ TCP segment structure

Figure 3.1

3.2.2 GBN, SR and TCP

TLDR; TCP is a mix between SR and GBN
GBN: Go-Back-N ARQ is a specific instance of the automatic repeat request (ARQ) protocol, in which the sending process continues to send a number of frames specified by a window size even without receiving an acknowledgement (ACK) packet from the receiver. It is a special case of the general sliding window protocol with the transmit window size of N and receive window size of 1. It can transmit N frames to the peer before requiring an ACK.
SR: Selective Repeat is part of the automatic repeat-request (ARQ). With selective repeat, the sender sends a number of frames specified by a window size even without the need to wait for individual ACK from the receiver as in Go-Back-N ARQ. The receiver may selectively reject a single frame, which may be retransmitted alone; this contrasts with other forms of ARQ, which must send every frame from that point again. The receiver accepts out-of-order frames and buffers them. The sender individually retransmits frames that have timed out.
TCP acknowledgments are cumulative and correctly received but out-of-order

segments are not individually ACKed by the receiver. Consequently, the TCP sender need only maintain the smallest sequence number of a transmitted but unacknowledged byte (SendBase) and the sequence number of the next byte to be sent (NextSeqNum). In this sense, TCP looks a lot like a GBN-style protocol. But there are some striking differences between TCP and Go-Back-N. Many TCP implementations will buffer correctly received but out-of-order segments consider also what happens when the sender sends a sequence of segments 1, 2, . . . , N, and all of the segments arrive in order without error at the receiver. Further suppose that the acknowledgment for packet n $<$ N gets lost, but the remaining N − 1 acknowledgments arrive at the sender before their respective timeouts. In this example, GBN would retransmit not only packet n, but also all of the subsequent packets n + 1, n + 2, . . . , N. TCP, on the other hand, would retransmit at most one segment, namely, segment n. Moreover, TCP would not even retransmit segment n if the acknowledgment for segment n + 1 arrived before the timeout for segment n.

3.2.3 TCP: Three-way handshake

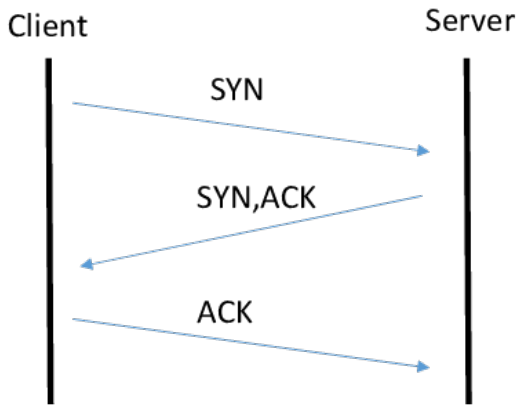


Figure 3.2

When establishing a TCP-connection, the host (user) sends a SYNchronize packet to the receiver (e.g. a server). The server then replies with a SYN+ACK (in one packet). Finally, the user sends back an ACK to the server, indicating it has received the SYN. A TCP-connection is now made between the two. This is unidirectional, since packets can be sent both ways.
For a monodirectional (one-way) connection, only a two-way handshake is required (SYN from host, ACK from receiver)

3.2.4 TCP Congestion control (and phases)

MSS: Maximum Segment Size
CWND: Congestion window. Is set to CWND/2 + 3 if received triple-duplicate ACK
RTT: Round-trip time
sstresh: Slow start threshold
Algorithms/stages:
Slow-start:
Increase the congestion window after a connection is initialized or after a time-out. It starts with a window a small multiple of the MSS in size. Although the initial rate is low, the rate of increase is very rapid; for every packet acknowledged, the congestion window increases by 1 MSS so that the congestion window effectively doubles for every RTT.

Fast retransmit & Fast recovery:
TCP Tahoe:
If three duplicate ACKs are received (i.e. four ACKs acknowledging the same packet, which are not piggybacked on data and do not change the receiver's advertised window), Tahoe performs a **fast retransmit**, sets the slow start threshold to half of the current congestion window, reduces the congestion window to 1 MSS, and resets to slow start state.

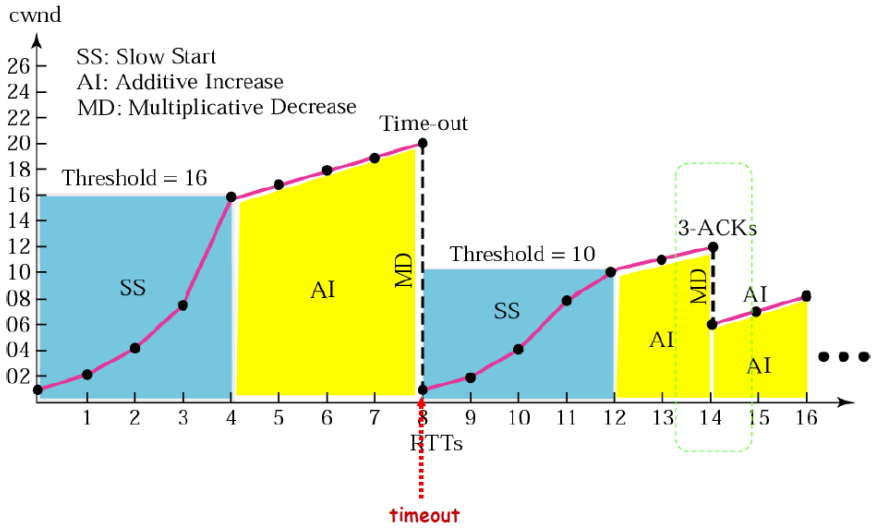
TCP Reno:
If three duplicate ACKs are received, Reno will perform a **fast retransmit** and skip the slow start phase by instead halving the congestion window (instead of setting it to 1 MSS like Tahoe), setting the slow start threshold equal to the new congestion window, and enter a phase called *fast recovery.*

Congestion avoidance(Additive increase):
Also called "Congestion avoidance": Once ssthresh is reached, TCP changes from slow-start algorithm to the linear growth (congestion avoidance) algorithm:

Additive increase: As long as non-duplicate ACKs are received, the congestion window is additively increased by one MSS every round-trip time.

Multiplicative decrease: on "loss event" set threshold to new value: threshold = CWND/2, and go back to Slow Start.

Example **[TCP congestion control]**



0-4 is slow start. 4-8 is additive increase. At 8 TCP Tahoe happens and 8-12 is slow start. Then 12-14 is additive increase. At 14 TCP Reno happens and then 14-16 is additive increase.

3.3 Other protocols

1. ARP: Address resolution protocol
   - Address Resolution Protocol (ARP) is a protocol for mapping an IP address to a physical machine address (MAC address) that is recognized in the local network
   - hierarchical address to non-hierarchical (MAC addresses)
     - MAC: Media Access Control. Unique ID of a network interface
2. IP: Internet Protocol
   - Responsible for routing and
   - IPv4 and IPv6
3. FTP: File Transfer Protocol
4. DHCP: Dynamic Host Configuration Protocol
   - DHCP allows a host to obtain (be allocated) an IP address automatically. A network administrator can configure DHCP so that a given host receives the same IP address each time it connects to the

network, or a host may be assigned a temporary IP address that will be different each time the host connects to the network. In addition to host IP address assignment, DHCP also allows a host to learn additional information, such as its subnet mask, the address of its first-hop router (often called the default gateway), and the address of its local DNS server.

- Uses UDP

5. DNS: Domain Name System

   -

3.4 Pull- and push-based protocols

In **push protocols**, the client opens a connection to the server and keeps it constantly active. The server PUSHes (sends) the new events to the client.
In **pull protocols**, the client periodically connects to the server, checks for and gets (pulls) recent events and then closes the connection and disconnects from the server. The client repeats this whole procedure to get updated about new events. In this mode, the clients periodically PULLs the new events from the server.

3.5 Circuit switched and packet switched network

3.5.1 Circuit switched

A client reserves the resources needed along a path (buffers, link transmission rate) to provide for communication between the end systems for the duration of the communication session between the end systems.

3.5.2 Packet switched

The resources (buffers, link transmission rate) are not reserved; a session's messages use the resources on demand and, as a consequence, may have to wait (that is, queue) for access to a communication link.

3.5.2.1 Packet definition

A network packet is a formatted unit of data carried by a packet-switched network. A packet consists of control information and user data, which is also known as the **payload**.

3.5.3 Sequence number

This sequence number is included on each transmitted packet, and acknowledged by the opposite host as an acknowledgement number to inform the sending host that the transmitted data was received successfully.
The sequence number is included on each transmitted packet, and acknowledged by the opposite host as an acknowledgement number to inform the sending host that the transmitted data was received successfully.

3.5.4 HTTP

3.5.4.1 HTTP requests

(These are in the method field, e.g. you can send a HTTP request with GET in the method field.)

1. GET: get a website (ingen data andet end URL'en nødvendig)
2. HEAD: get a header
3. POST – Sender information til en webside (f.eks. fra formular hvor brugeren har udfyldt nogle oplysninger).
4. PUT – (Over)skrive en webside (dvs lagre en ny version).
5. DELETE: deletes the specified resource.
6. TRACE – Sende forespørgslen uændret tilbage (for at kontrollere forbindelsen)
7. OPTIONS – Spørger hvilke metoder serveren understøtter.
8. CONNECT – Anvendes med proxy-servere til SSL-tunneller.

Methods PUT and DELETE are defined to be idempotent, meaning that multiple identical requests should have the same effect as a single request.

3.5.5 HTTP format

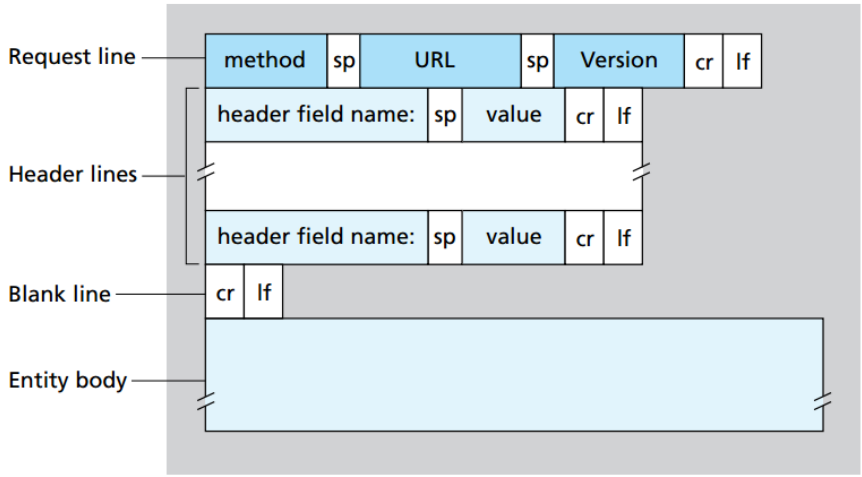
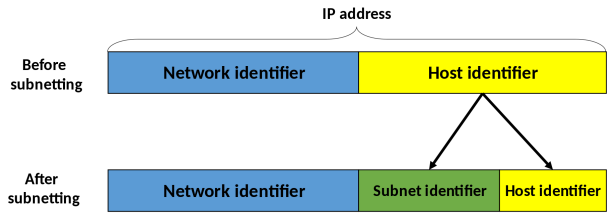


Figure 2.8 ♦ General format of an HTTP request message

Figure 3.3

3.5.6 Bitmasking & Subnetting

Bitmasking IP-addresses can be used for subnetting as well as converting IPv4 to IPv6.
The practice of dividing a network into two or more networks is called subnetting.



An IPv4 subnet mask is 32 bits, a sequence of ones (1) followed by a block of zeros (0). The ones indicate bits in the address used for the network prefix and the trailing block of zeros designates that part as being the host identifier.

The following example shows the separation of the network prefix and the host identifier from an address (192.0.2.130) and its associated /24 subnet mask (255.255.255.0). The operation is visualized in a table using binary address formats.

	Binary form	Dot-decimal notation
IP address	11000000.00000000.00000010.10000010	192.0.2.130
Subnet mask	11111111.11111111.11111111.00000000	255.255.255.0
Network prefix	11000000.00000000.00000010.00000000	192.0.2.0
Host identifier	00000000.00000000.00000000.10000010	0.0.0.130

The result of the bitwise AND operation of IP address and the subnet mask is the network prefix 192.0.2.0. The host part, which is 130, is derived by the bitwise AND operation of the address and the one's complement of the subnet mask.
Below table can be helpful for calculating how big subnets and their masks have to be according to specifications:

Subnet	1	2	4	8	16	32	64	128	256
Host	256	128	64	32	16	8	4	2	1
Subnet Mask	/24	/25	/26	/27	/28	/29	/30	/31	/32

Note: The amount of users a single subnet can have for, example a /28 subnet mask is not 16, but 14. The reason for this, is the fact that the lowest address (all 0s in subnet bits) is the Host address, and the highest (all 1s in subnet bits) is the Broadcasting address.

3.5.7 Formulas

1. $d = N\frac{L}{R}$
   - d: end to end delay
   - N: number of links on the route (N = 1 + number of routers between source and destination)
   - R: transmission rate
2. Traffic intensity: La/R
   - L: bits in packet
   - a: average rate of packet arrival (unit is packet/time)
   - R: transmission rate (bits/sec) at which packets are pushed out of the queue
   - La/R $>$ 1 means that packets arrive faster than the queue can forward them and so the queuing delay goes to infinity.

3.5.8 Socket interface

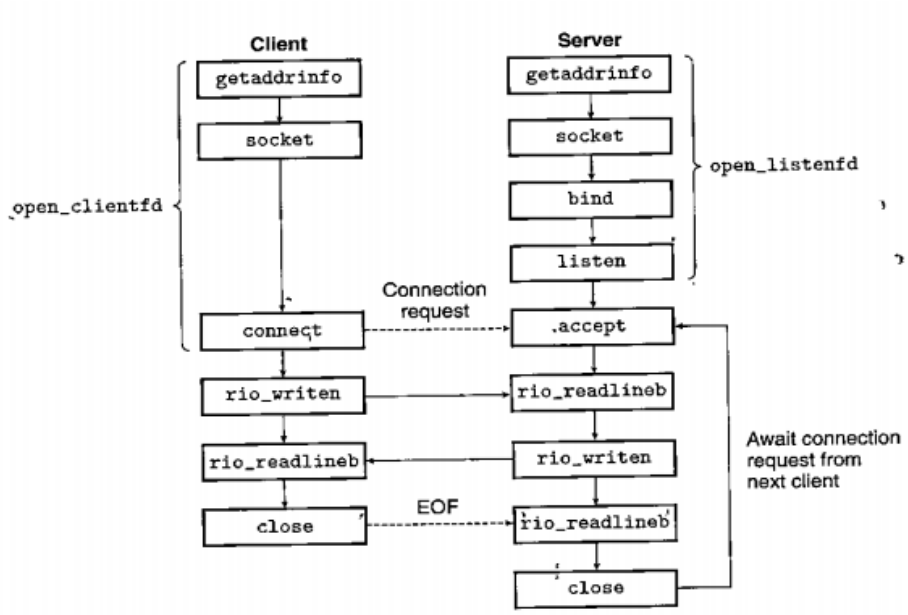


Figure 11.12 Overview of network applications based on the sockets interface.

Figure 3.4

3.5.9 Delay

1. Processing delay: time it takes router to process the packet header and other factors such as checking for bit-level errors
2. Queuing delay: time the packet spends in routing queues
3. Transmission delay: time it takes to push the packet's bits onto the link
4. Propagation delay: time from it is transmitted from the queue to it arrives at receiver

3.6 TCP/IP model (aka Internet protocol suite)

It is a conceptual model and set of communications protocols. It provides end-to-end data communication specifying how data should be packetized, addressed, transmitted, routed, and received. This functionality is organized into four abstraction layers, which classify all related protocols according to the scope of networking involved.
Another model is the OSI model, but we do not work with that.

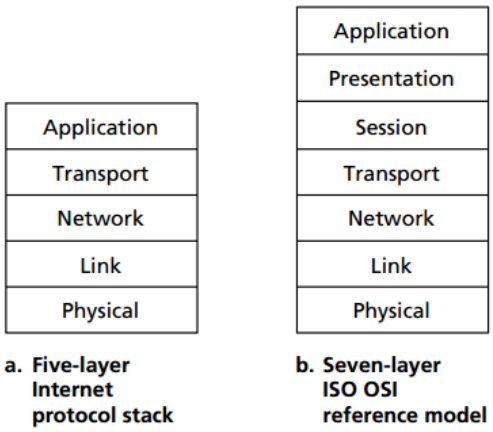
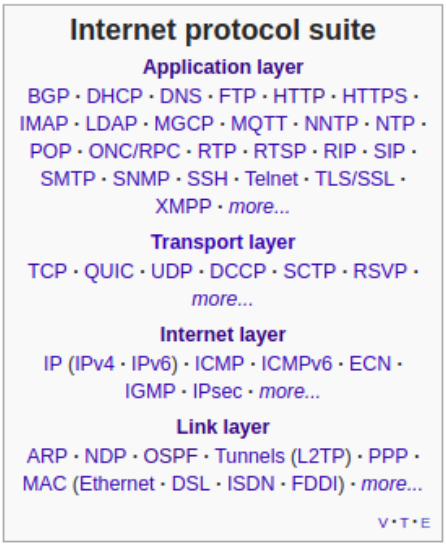


Figure 1.23 ♦ The Internet protocol stack (a) and OSI reference model (b)

The layers are in a hierarchy where a layer depends on the layers below it.

3.6.1 Examples of protocols in the different layers



3.6.2 Application Layer

The application layer is the scope within which applications create user data and communicate this data to other applications on another or the same host.

It includes protocols such as the HTTP protocol, SMTP (which provides for the transfer of e-mail messages), and FTP (which provides for the transfer of files between two end systems).

3.6.3 Transport Layer

The Internet's transport layer transports application-layer messages between application endpoints. It uses either TCP or UDP to transport the messages.

The maximum TCP segment size is denoted by MSS (Maximum Segment Size), which denotes bytes a segment can consist of. If TCP wants to send a file, but the file is too large to fit into a single segment, then TCP sends the file across multiple segments. The first segment will have sequence number 0 and the size will be MSS. Because of zero indexing the next segment number will be the MSS value, the third will be MSS $\times$ 2 and so on. The sequence number is therefore the sequence number of the first byte in the data field. The acknowledgement number, that the sending host put in its segment, is the sequence number of the segment that it wants from the receiving host.
TCP is full-duplex because because data can be sent together with an ACK, meaning that one can reply to, and transmit data at the same time.

3.6.4 Segment format (packet fields)

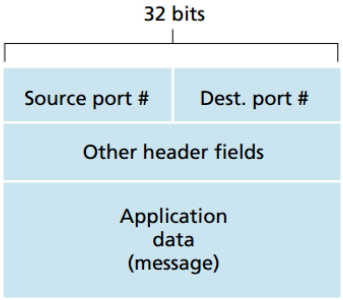


Figure 3.3 ♦ Source and destination port-number fields in a transport-layer segment

If TCP is used, then it is the same as a TCP segment, whose structure can be seen at subsection 3.2.1 (I think it is the same??).

3.6.5 Network Layer

The Internet's network layer moves network-layer packets (datagrams) from one host to another. The transport layer sends segment and a destination

address and the network layer delivers the segment to the transport layer in the destination host.
The Internet's network layer includes the IP protocol, which defines the fields in the datagram as well as how the end systems and routers act on these fields.
The Internet's network layer also contains routing protocols.
The network layer is often a mixed implementation of hardware and software.

3.6.5.1 Internet checksum

Each line is 16 bits (2 bytes) long.
To find the checksum, XOR all the lines together, using carry's and wrap-around.
The ! (not) of the resulting 16-bit sequence is the checksum.

3.6.5.2 Packet fragmentation

Fragmentation occurs when an IP datagram traverses a network which has a maximum transmission unit (MTU) that is smaller than the size of the datagram. Fragmentation is done by the router, where the fragments continue until they are reassembled into the original packet at the receiving host.
Information needed for the destination host in order to re-assemble the fragments are:
1. **Fragment ID**: An identifier from the IP header which associates all fragments to the same datagram.
2. **Place & Offset**: To identify the fragments position within the original datagram.
3. **Length**: Each fragment must tell the length of data carried.
4. **MF flag**: Used to determine if more fragments are to follow.
Note: Maximum IP datagram header size is 60 bytes, minimum is 20 bytes (which is the normal header size).

3.6.6 IPv4 datagram format (packet fields)

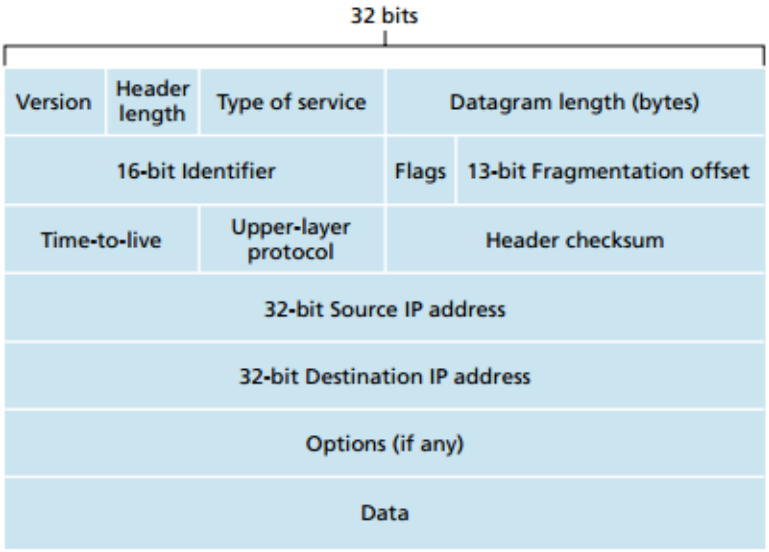


Figure 4.16 ♦ IPv4 datagram format

Explanations:

1. Version number. These 4 bits specify the IP protocol version of the datagram. By looking at the version number, the router can determine how to interpret the remainder of the IP datagram. Different versions of IP use different datagram formats. The datagram format for IPv4 is shown in Figure 4.16. The datagram format for the new version of IP (IPv6) is discussed in Section 4.3.5 (book).

2. Header length. Because an IPv4 datagram can contain a variable number of options (which are included in the IPv4 datagram header), these 4 bits are needed to determine where in the IP datagram the payload (e.g., the transport-layer segment being encapsulated in this datagram) actually begins. Most IP datagrams do not contain options, so the typical IP datagram has a 20-byte header.

3. Type of service. The type of service (TOS) bits were included in the IPv4 header to allow different types of IP datagrams to be distinguished from each other. For example, it might be useful to distinguish real-time datagrams (such as those used by an IP telephony application) from non-real-time traffic (for example, FTP). The specific level of service to be provided is a policy issue determined and configured by the network

administrator for that router. Two of the TOS bits are used for Explicit Congestion Notification.

4. Datagram length. This is the total length of the IP datagram (header plus data), measured in bytes. Since this field is 16 bits long, the theoretical maximum size of the IP datagram is 65,535 bytes. However, datagrams are rarely larger than 1,500 bytes, which allows an IP datagram to fit in the payload field of a maximally sized Ethernet frame.

5. Identifier, flags, fragmentation offset. These three fields have to do with so-called IP fragmentation. Interestingly, the new version of IP, IPv6, does not allow for fragmentation.

6. Time-to-live. The time-to-live (TTL) field is included to ensure that datagrams do not circulate forever (due to, for example, a long-lived routing loop) in the network. This field is decremented by one each time the datagram is processed by a router. If the TTL field reaches 0, a router must drop that datagram.

7. Protocol. This field is typically used only when an IP datagram reaches its final destination. The value of this field indicates the specific transport-layer protocol to which the data portion of this IP datagram should be passed. For example, a value of 6 indicates that the data portion is passed to TCP, while a value of 17 indicates that the data is passed to UDP. For a list of all possible values, see [IANA Protocol Numbers 2016]. Note that the protocol number in the IP datagram has a role that is analogous to the role of the port number field in the transport-layer segment. The protocol number is the glue that binds the network and transport layers together, whereas the port number is the glue that binds the transport and application layers together. We'll see in Chapter 6 that the link-layer frame also has a special field that binds the link layer to the network layer.

8. Header checksum. The header checksum aids a router in detecting bit errors in a received IP datagram. The header checksum is computed by treating each 2 bytes in the header as a number and summing these numbers using 1s complement arithmetic. As discussed in Section 3.3, the 1s complement of this sum, known as the Internet checksum, is stored in the checksum field. A router computes the header checksum for each received IP datagram and detects an error condition if the checksum carried in the datagram header does not equal the computed checksum. Routers typically discard datagrams for which an error has been detected. Note that the checksum must be recomputed and stored again at each router, since the TTL field, and possibly the options field as well, will change. An interesting discussion of fast algorithms for computing the Internet checksum is [RFC 1071]. A question often asked at this point is, why does TCP/IP perform error checking at both the transport and network layers? There are several reasons for this repetition. First, note that only the IP header

is checksummed at the IP layer, while the TCP/ UDP checksum is computed over the entire TCP/UDP segment. Second, TCP/ UDP and IP do not necessarily both have to belong to the same protocol stack. TCP can, in principle, run over a different network-layer protocol (for example, ATM) [Black 1995]) and IP can carry data that will not be passed to TCP/UDP.

9. Source and destination IP addresses. When a source creates a datagram, it inserts its IP address into the source IP address field and inserts the address of the ultimate destination into the destination IP address field. Often the source host determines the destination address via a DNS lookup, as discussed in Chapter 2. We'll discuss IP addressing in detail in Section 4.3.3.

10. Options. The options fields allow an IP header to be extended. Header options were meant to be used rarely—hence the decision to save overhead by not including the information in options fields in every datagram header. However, the mere existence of options does complicate matters—since datagram headers can be of variable length, one cannot determine a priori where the data field will start. Also, since some datagrams may require options processing and others may not, the amount of time needed to process an IP datagram at a router can vary greatly. These considerations become particularly important for IP processing in highperformance routers and hosts. For these reasons and others, IP options were not included in the IPv6 header, as discussed in Section 4.3.5.

11. Data (payload). Finally, we come to the last and most important field—the raison d'etre for the datagram in the first place! In most circumstances, the data field of the IP datagram contains the transport-layer segment (TCP or UDP) to be delivered to the destination. However, the data field can carry other types of data, such as ICMP messages (discussed in Section 5.6).

3.6.7 Link Layer

The link layer delivers the datagram to the next node along the route. At this next node, the link layer passes the datagram up to the network layer. The layer depend on the specific link-layer protocol that is employed over the link. The protocols provide reliable delivery, from transmitting node, over one link, to receiving node. This reliable is not the same as for TCP, as it is just guaranteed delivery over one link, where TCP guarantees from end-to-end.

3.6.8 Physical Layer

The physical layer moves the individual bits within the frame from one node to the next. The protocols in this layer are again link dependent and further

depend on the actual transmission medium of the link (for example, twisted-pair copper wire, single-mode fiber optics).

3.6.9 Name of a packet in the layers

I do not know if a packet is the right word in the title - really it is just the name of the thing that the layer sends. But the book uses says e.g. "a transport layer packet is called a segment", so I think that I am right.

1. Application: message
2. Transport: Segment
3. Network: Datagram
4. Link: Frame
5. Physical: Bits

3.7 Routing Algorithms

3.7.1 Abbreviations and routing protocols

1. LS: link-state algorithm
2. LSR: Link-State Routing
3. DV: distance-vector algorithm (s 412)
4. OSPF: open shortest path first (s. 420)
    - routing protocol for IP networks, that implements Dijkstra's algorithm. It uses a LSR protocol.
    - link-state protocol that uses flooding of link-state information and a Dijkstra's least-cost path algorithm. Each router constructs a complete topological map (that is, a graph) of the entire autonomous system. Each router then locally runs Dijkstra's shortest-path algorithm to determine a shortest-path tree to all subnets, with itself as the root node
5. RIP
6. BGP

3.7.2 Broad routing algorithm classifications

3.7.3 Centralized vs decentralized

Centralized routing algorithms computes least-cost path based on complete, global information about connectivity and link costs. Algorithms with global state information are often referred to as link-state (LS) algorithms.

Decentralized routing algorithm calculates the least-cost path in an iterative, distributed manner by the routers. Each node begins with only the knowledge of the costs of its own directly attached links. An example is the distance-vector (DV) algorithms.

3.7.4 Load sensitivity (congestion sensitivity)

In a load-sensitive routing algorithm the cost of each link takes into account the amount of congestion. Today's internet routing algorithms (OSPF, RIP, BGP..) are load-**in**sensitive

3.7.5 Static vs dynamic

Static routing algorithms only rarely update link costs, often needs a human to update it.

Dynamic routing algorithms When traffic loads of topology of the network changes the link costs in the algorithm are updated. Higher risk of routing loops or route oscillation.

3.7.6 Least-cost path vs shortest path

The shortest path is the path with the least number of links between the source and the destination. The least-cost path is the path with the least sum of link weights in the path.

The links are only rarely weighted the same and thus least-cost path is not always the same path as the shortest path.

3.7.7 LS: link-state algorithm

Centralized routing algorithm.

3.7.7.1 Dijkstra's algorithm code

Link-State (LS) Algorithm for Source Node *u*

```
1  Initialization:
2   N' = {u}
3   for all nodes v
4     if v is a neighbor of u
5       then D(v) = c(u,v)
6     else D(v) = ∞
7
8  Loop
9   find w not in N' such that D(w) is a minimum
10  add w to N'
11  update D(v) for each neighbor v of w and not in N':
12        D(v) = min(D(v), D(w)+ c(w,v) )
13   /* new cost to v is either old cost to v or known
14    least path cost to w plus cost from w to v */
15 until N'= N
```

A network with nodes in set N where the distances from a node u to all other nodes are calculated. The algorithm goes through all possible paths from u to all other nodes. If the algorithm looks at a path to a node k, and this path is less costly than the value in D(k), then D(k) is set to the cost of the new path.

3.7.8 DV: distance-vector algorithm

(p. 412 in book)
Each node maintains a vector of estimates of the costs (distances) to all other nodes in the network.

poisoned reverse

3.7.9 Advantages and Disadvantages

1. Advantages:
2. Does not need perfect and complete knowledge of the network before the algorithm can run.
3. Has a low messaging complexity since distance vectors are exchanged between connected neighbors only
4. Not a centralized protocol and each node can compute the distance vectors simultaneously and in an iterative fashion.

Disadvantages:

1. Has low speed of convergence (potentially never), can have routing loops and count-to-infinity problems.

2. Not very robust since a faulty router can potentially affect the distance vector computations on all routers.

3.7.10 Bellman-Ford algorithm code

Distance-Vector (DV) Algorithm

At each node, x:

```
1  Initialization:
2    for all destinations y in N:
3       Dx(y) = c(x,y) /* if y is not a neighbor then c(x,y) = ∞ */
4    for each neighbor w
5       Dw(y) = ? for all destinations y in N
6    for each neighbor w
7       send distance vector  Dx = [Dx(y): y in N] to w
8
9  loop
10     wait  (until I see a link cost change to some neighbor w or
11              until I receive a distance vector from some neighbor w)
12
13     for each y in N:
14         Dx(y) = minv{c(x,v) + Dv(y)}
15
16 if Dx(y) changed for any destination y
17       send distance vector Dx  = [Dx(y): y in N] to all neighbors
18
19 forever
```

The Bellman-Ford equation:

$$d_x(y) = min_v\{c(x,v) + d_v(y)\}$$

The basic idea is as follows. Each node x begins with $D_x(y)$, an estimate of the cost of the least-cost path from itself to node y, for all nodes, y, in N. Let $D_x = [D_x(y) : yinN]$ be node x's distance vector, which is the vector of cost estimates from x to all other nodes, y, in N. With the DV algorithm, each node x maintains the following routing information:

• For each neighbor v, the cost c(x,v) from x to directly attached neighbor, v • Node x's distance vector, that is, $D_x = [D_x(y) : yinN]$, containing x's estimate of its cost to all destinations, y, in N • The distance vectors of each of its neighbors, that is, $D_v = [D_v(y) : yinN]$ for each neighbor v of x

3.7.11 Link-state broadcast

3.8 Error detection

3.8.1 Parity Checks

Even parity: bit is 0 if even number of 1's
Odd parity: bit is 0 if odd number of 1's

OBS: if an even number of bit errors occur, the error will be undetected.

3.8.2 Checksum

Når der ved udregnes checksum, sker det ved, at to 16-bit tal lægges sammen (summen af disse lægges så til et tredje 16-bit tal). Den endelige sum omregnes til ens-kompliment (flipper 1 til 0 og 0 til 1), der så bliver checksummen. Hvis der opstår en error således, at den samme plads i de to 16-bit tal, der lægges sammen, begge flippes (og den ene er 1 og den anden 0), så resulterer det i den samme sum - og dermed den samme checksum.

3.9 Misc terminology

1. Bandwidth: maximum rate of data transfer across a given path (bits/sec, Mbps)
2. Transmission rate: bits that are transmitted per time (bits/sec, Mbps)
3. FDM: frequency-division multiplexing
    - Each circuit continuously gets a fraction of the bandwidth
4. TDM: time-division multiplexing
    - Each circuit gets all of the bandwidth periodically during brief intervals of time (during a slot)
5. ISP: Internet Service Provider
6. CDN: Content Delivery Networks
7. IXP: Internet Exchange Point
    - ISP and CDN can exchange Internet traffic between their networks through IXPs
    - IXPs reduce the portion of an ISP's traffic that must be delivered via their upstream transit providers (ISPs higher in the hierarchy), thereby reducing the average per-bit delivery cost of their service.
    - Furthermore, the increased number of paths available through the IXP improves routing efficiency and fault-tolerance.
8. Flow control
    - sender/receiver speed matching such that the sender does not send too much data for the receiver to take in and not less either such that it is the most effective.
9. Parity bit

- A parity bit, or check bit, is a bit added to a sequence of binary code to ensure that the total number of 1-bits in the string is even or odd, as a form of simple error detection.
  Even parity means that an odd number of 1's in the sequence results in a 1.
  Odd parity means that an even number of 1's result in a 1.

10. Port
    - Endpoint of communication
    - Used to channel packets into the right device on a LAN and the right application (process) on a device.
    - Are identified for each protocol and address combination by 16-bit unsigned numbers, commonly known as the port number.
    - Each socket has a port with a port number to identify it
11. Socket address: IP address and port number
12. LAN: Local Area Network
13. WAN: Wide Area Network

3.10 Time to transmit a file

Remember:

1. multiply F by 8 if it is given in bytes
2. remember that F is in exponents of 2 so 10GB=8*10***1024**. 8 because GB is bytes.

3.10.1 Client-server

- Sending an F-bit file to N receivers
  - Transmitting NF bits at rate u_s
  - … takes at least NF/u_s time
- Receiving the data at the slowest receiver
  - Slowest receiver has download rate $d_{min} = min_i\{d_i\}$
  - … takes at least F/d_{min} time
- Download time: $max\{NF/u_s, F/d_{min}\}$

Figure 3.5

3.10.2 Peer-to-peer

- Components of distribution latency
  - Server must send each bit: min time F/u_s
  - Slowest peer must receive each bit: min time F/d_{min}
- Upload time using all upload resources
  - Total number of bits: NF
  - Total upload bandwidth $u_s + sum_i(u_i)$
- Total: $max\{F/u_s, F/d_{min}, NF/(u_s+sum_i(u_i))\}$
- *Peer to peer is self-scaling*
  - *Download time grows slowly with N*
    - Client-server: $max\{NF/u_s, F/d_{min}\}$
    - Peer-to-peer: $max\{F/u_s, F/d_{min}, NF/(u_s+sum_i(u_i))\}$

Figure 3.6

F bits to N receivers at server upload rate u_s. u is udpload rate and d is download rate.

Chapter 4

Misc

4.1 Javascript cheat sheet

4.1.1 Convert between number systems

Example:

$$(264).toString(”16”)$$

Returns "108", which is 264 is decimal written in hex (that is 16).

4.1.2 8-Bit ASCII table

Dec	Symbol	Binary	Dec	Symbol	Binary
65	A	0100 0001	83	S	0101 0011
66	B	0100 0010	84	T	0101 0100
67	C	0100 0011	85	U	0101 0101
68	D	0100 0100	86	V	0101 0110
69	E	0100 0101	87	W	0101 0111
70	F	0100 0110	88	X	0101 1000
71	G	0100 0111	89	Y	0101 1001
72	H	0100 1000	90	Z	0101 1010
73	I	0100 1001	91	[	0101 1011
74	J	0100 1010	92	\	0101 1100
75	K	0100 1011	93	]	0101 1101
76	L	0100 1100	94	^	0101 1110
77	M	0100 1101	95	_	0101 1111
78	N	0100 1110	96	`	0110 0000
79	O	0100 1111	97	a	0110 0001
80	P	0101 0000	98	b	0110 0010
81	Q	0101 0001	99	c	0110 0011
82	R	0101 0010	100	d	0110 0100