Computer abstractions and technology

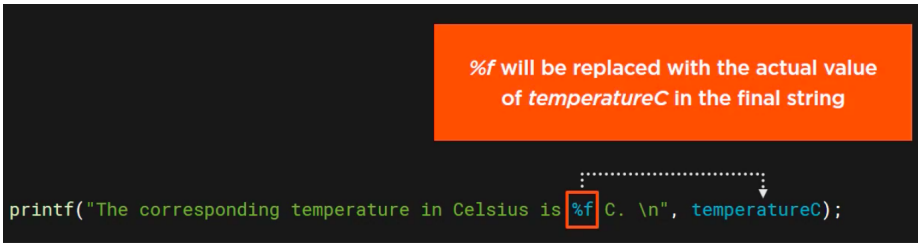
C noter

printf (“”) - output - en tekst der bliver printet og kan ses.

scanf(“Format specifier fx. %f”, variable that receives the input value) - input - kan tage et input fra brugeren i terminalen.

%f - floating-point number. hvis der kommer for mange decimaler, kan man skrive %.2f nu kommer der to decimaler.
%u - unsigned integer - negative tal
%x - hvis man vil outputte et hexi decimal nummer
%s - string format - læser en string. du kan skrive &39s, for at give et specifikt antal char.
& - address of, før ens variabel, fordi scanf skal vide hvor den skal finde variable.

printf kan skrive vores værdi vi har udregnet ind i en print statement på denne måde-



char name[]=”connie”;
char = skal skrives når man vil lave en string
name = er navnet på strengen
[ ] = array contiguos sequence of elements - den er vigtig fordi char ellers kun tag et char og ikke en sekvens af det.
“ “ = når man har en string
´ ´= når man har en char med en karakter som skal udtrykkes

en string (som ovenover) har altid et felt som er null til sidst derfor skal man huske på at hvis man vil lave en string på 6 karaktere skal man lave 7 til null feltet. hvis nul feltet ikke var der ville C når den har en string bare blive ved med at kører og lede efter data at skrive ud.

De mest brugte escape sequences i C.

Some Common Escape Sequences

Escape Sequence	Meaning
\n	Newline
\t	Horizontal tab
\'	Single quotation mark
\"	Double quotation mark
\x43	ASCII hex (‘C’)
\0	Null

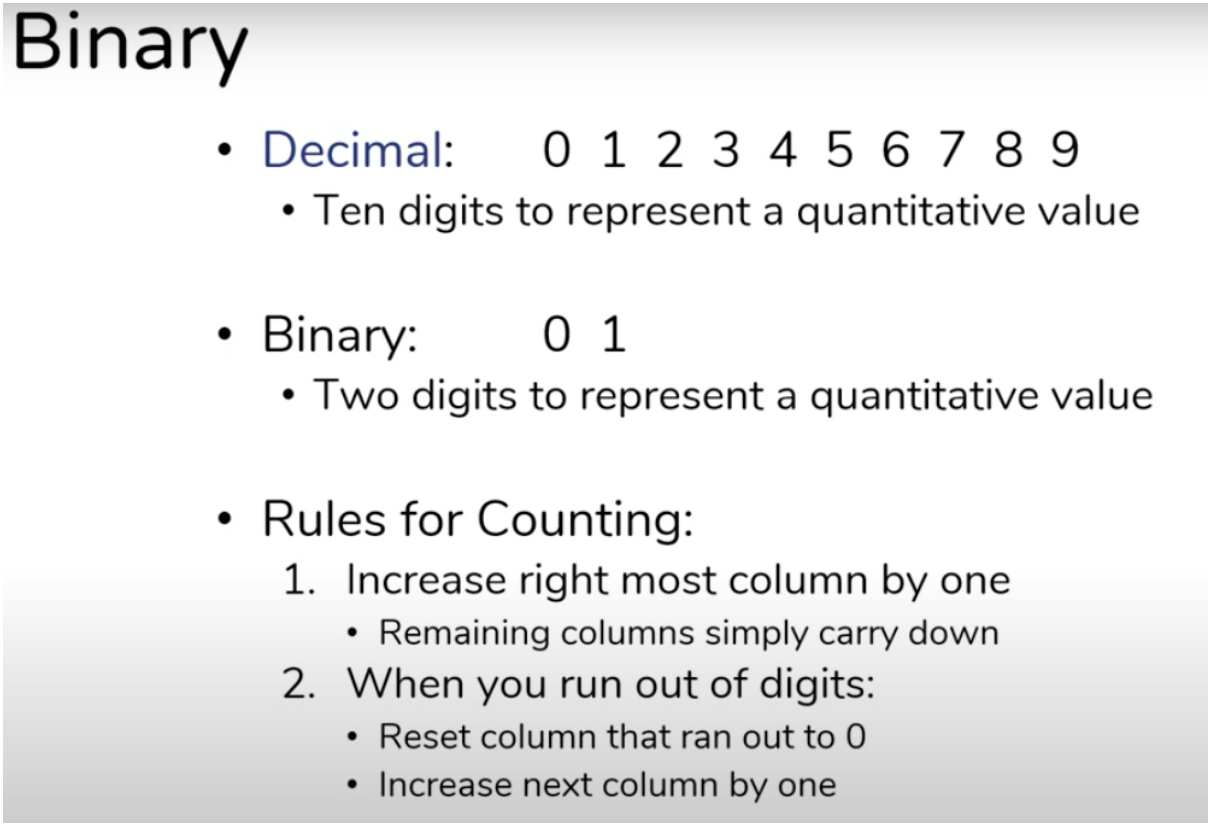
Int representation

Data types size

Data Type	32-bit sizes (in bytes)	64-bit sizes (in bytes)	Remarks
char	1	1	
short	2	2	
int	4	4	
long	4	8	
long long	8	8	
float	4	4	
double	8	8	
long double	16	16	
pointer	4	8	
wchar_t	2	4	Other UNIX platforms usually have wchar_t 4 bytes for both 32-bit and 64-bit mode.
size_t	4	8	This is an unsigned type.
ptrdiff_t	4	8	This is a signed type.

There is 4 bytes on 32 bits, and 8 bytes on 64-bits. So example for “char” the size in 32 bits is $1 \cdot 4 = 4$ bytes and in 64 bits its $1 \cdot 8 = 8$ bytes.

Binary count



When counting binary numbers we follow 2 rules as stated above, this gives us this count:

Binary

0	0	1110	14	11100	28
1	1	1111	15	11101	29
10	2	10000	16	11110	30
11	3	10001	17	11111	31
100	4	10010	18	100000	32
101	5	10011	19	100001	33
110	6	10100	20	100010	34
111	7	10101	21	100011	35
1000	8	10110	22	100100	36
1001	9	10111	23	100101	37
1010	10	11000	24	100110	38
1011	11	11001	25	100111	39
1100	12	11010	26	101000	40
1101	13	11011	27	101001	41

Binary to Decimal numbers

When we have a number as 56062 this actually means:
$50000 \quad 5 \cdot 10000$
$6000 \quad 6 \cdot 1000$
$000 \quad 0 \cdot 100$
$60 \quad 6 \cdot 10$
$2 \quad 2 \cdot 1$

In the same way we can solve a binary number 10101

$10000 \quad 1 \cdot 10000$ = 16
$0000 \quad 0 \cdot 1000$ = 8
$100 \quad 1 \cdot 100$ = 4
$00 \quad 0 \cdot 10$ = 2
$1 \quad 1 \cdot 1$ = 1

Increasing by multiplying with 2 every step so the next decimal would be:
32, 64, 128, 256, 512, 1024, 2048 etc…
We only count the decimals that have a 1 not those with a zero, so the number 10101 would be:

$1 + 4 + 16 = 21$

Examples:

$1110001 = 1 + 16 + 32 + 64 = 113$
$101 = 1 + 4 = 5$
$100111 = 1 + 2 + 4 + 32 = 39$
$11100011 = 1 + 16 + 32 + 64 + 128 = 241$

Decimal to binary

When we want to convert a decimal number to a binary number, we have to find out which of the above “spaces” gives for example 229:

We start by the highest bit in this example 128, and see if that can be subtracted from 229, it can so then we have $229 - 128 = 101$ which gives us a 1.
Now we check if we can subtract 64 from 101, $101 - 64 = 37$ which gives us a 1
$37 - 32 = 5$ which gives us a 1
16 can not be subtracted from 16, nor so 8, these are both 0.
then we have $5 - 4 = 1$ which gives a 1
2 can not be subtracted from 1, which is a 0.
then we have $1 - 1 = 0$, which gives us a 1.
$128 \; 64 \; 32 \; 16 \; 8 \; 4 \; 2 \; 1$
$1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \; 1 = 229$

Binary to Hexadecimal

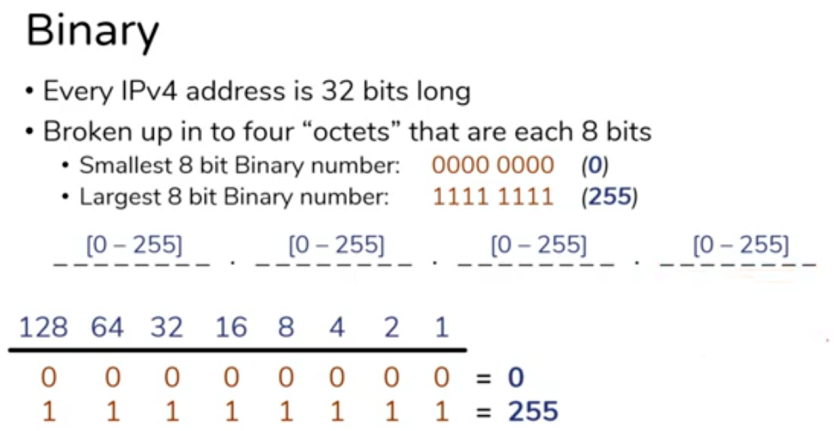
Let's say we have the binary number 10110111 and want it as a hexadecimal number, we first split it into fours, so we have 1011 and 0111, and convert them into decimals.

$1011 = 1 + 2 + 8 = 11$

$0111 = 1 + 2 + 4 = 7$

Then we find the corresponding number in Hex 11 = B and 7 = 7 so the binary number 10110111 in hex is B7.

8 bits and IPV4 adresses



Hexadecimals

DECIMAL	HEX	BINARY
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Hexadecimals goes from $1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$

Hexa is also called base 16 because each place is base 16 to the power of the place.

Decimal to Hexa

If we have the decimal number 348 and we want to find the Hexa number, we remember that it has base 16 and therefore we need to multiply it by 16.

$348 / 16 = 21,75$ this gives us 21 and we divide 75 with 16 to get the remaining
$75 / 16 = 12$ (on a calculator $16 \cdot 0.75$)
so we have 21 R 12
then we find $21 / 16 = 1.3125 = 3125 / 16 = 1\,R\,5$
So we have $1 / 16 = 0\,R\,1$

We read it from the bottom so it gives us 1 5 and 12, since Hexa is not 12 we find the corresponding letter which is c, so the decimal 348 in hexa is 15C.

Hexa to Decimal

When we want to take a hexa number for example 23E to a decimal we find it by saying:
$2 \quad 3 \quad E$
$16^2 \; 16^1 \; 16^0$

THis means that we multiply the number with 16 to the power of n, and fist we convert E = 14, so we have:

$$2 \cdot 16^2 + 3 \cdot 16^1 + 14 \cdot 16^0 = 2 \cdot 256 + 48 + 14 = 574$$

Another example F7D F = 15 D = 13

$F \quad 7 \quad D$
$16^2 \; 16^1 \; 16^0$

THis means that we multiply the number with 16 to the power of n, so we have:

$$15 \cdot 16^2 + 7 \cdot 16^1 + 13 \cdot 16^0 = 15 \cdot 256 + 112 + 13 = 3965$$

If its a fractal number like 13F.C8 then it counts down to 16 to the power of -1 and so on.

Hexa to Binary

Let's say we have the hexadecimal A9 and want it in binary numbers. Firstly we translate the A = 10 and then we look at the binary sequence 8, 4, 2, 1 :

We need to see which numbers that gives 10 and 9

10		9	
8	4 2 1	8	4 2 1
1	0 1 0	1	0 0 1

We put a 1 under the numbers that is needed for getting $10 = 8 + 2$ and $9 = 8 + 1$.
So the Hexa number A9 is in binary 1010100.

Assembly to C - C to Assembly COD 2.1-2.4

Design principles

Design Principle 1: Simplicity favors regularity.

Design Principle 2: Smaller is faster. - A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther. Thus 31 registers may not be better or faster than 32.

Every computer must be able to perform arithmetic. The RISC-V assembly language notation.

$add\ a,\ b,\ c$

instructs a computer to add the two variables b and c and to put their sum in a.

In RISC-V arithmetic the instructions perform one operation and must always have exactly three variables. Suppose we want the sum of 4 variables b, c, d and e into the variable a. We can do it with three instructions like this.

```
add a, b, c       // The sum of b and c is placed in a
add a, a, d       // The sum of b, c, and d is now in a
add a, a, e       // The sum of b, c, d, and e is now in a
```

The natural number of operands for an operation like addition is three: the two numbers being added together and a place to put the sum.

Arithmetic instructions are restricted, this means that they must be from a limited number of special locations built directly in hardware called registers.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like RISC-V.

Although we could simply write instructions using numbers for registers, from 0 to 31, the RISC-V convention is x followed by the number of the register, except for a few register names that we will cover later.

Memory operands

To keep more information than the 32 registers we use something called a memory. As explained above, arithmetic operations occur only on registers in RISC-V instructions; thus, RISC-V must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**.

data transfer instruction A command that moves data between memory and registers.

If we want to access a word in the memory, the instruction must supply the memory address.

Memory: is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0.

For example, in this figure, the address of the third data element is 2, and the value of memory [2] is 10.

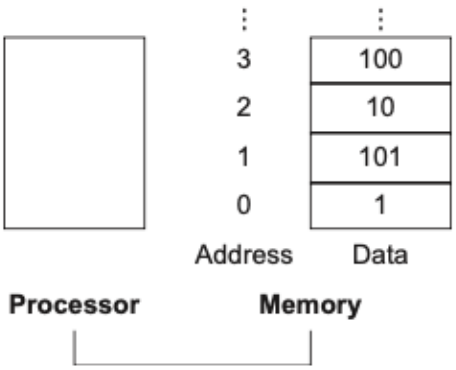


FIGURE 2.2 Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since RISC-V actually uses byte addressing, with each word representing 4 bytes. Figure 2.3 shows the correct memory addressing for sequential word addresses.

The data transfer instruction that copies data from memory to a register is traditionally called load. The format of the load instruction is the name of the operation followed by the register to be loaded, then register and a constant used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address.

Since 8-bit *bytes* are useful in many programs, virtually all architectures today address individual bytes. Therefore, the address of a word matches the address of one of the 4 bytes within the word, and addresses of sequential words differ by 4.

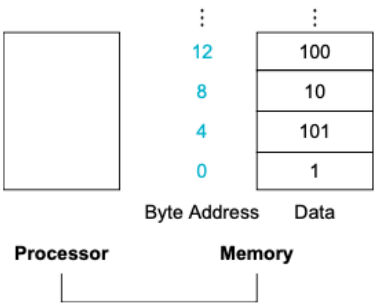


FIGURE 2.3 Actual RISC-V memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since RISC-V addresses each byte, word addresses are multiples of 4: there are 4 bytes in a doubleword.

So the real byte addresses in the memory is this and memory [8] is now 10.

Alignment restriction: A requirement that data be aligned in memory on natural boundaries.

store word, sw: is used when we want to copy a word between memory and register. The sw is used when storing words in the memory see example 2.

(load, lw) The RISC-V instructions is called lw, standing for load word.

Example 1

Lets say we have the C assignment statement.

$g = h + A[8]$

This means we have an array A lets say of 100 words, g and h are x20 and x21, then we assume that the starting address or base address is in x22.

We have an element in the memory A[8] and we want to load/transfer that into a register. The address of this array element is the sum of the base of the array A, found in register x22, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction.

The first instruction in RISC-V is loading/tranfering the operand from the memory in to a temporary register.

$lw\ x9,\ 8(x22)\ //\ Temporary\ register\ x9\ gets\ A[8]$

This equals to $x9 = A[8]$

Now we want to add h to x9, and put that sum into the register corresponding to g.

$add\ x20,\ x21,\ x9\ //\ g = h + A[8]$

The register added to form the address (x22) is called the *base register*, and the constant in a data transfer instruction (8) is called the *offset*.

Another example with the use of offset 23 for byte address:

Example 2

Assume variable h is associated with register x21 and the base address of the array A is in x22. What is the RISC-V assembly code for the C assignment statement below?

$A[12] = h + A[8]$

Now we have two operands in the memory and therefor we need more RISC-V instructions. This time we use the proper offset of 32 for byte addressing in the load register instruction to select A[8], and the add instruction places the sum in x9:

$lw\ x9,\ 32(x22)\ //\ Temporary\ register\ x9\ gets\ A[8]$

$add\ x9,\ x21,\ x9\ //\ Temporary\ register\ x9\ now\ hold\ h\ +\ A[8]$

The final instruction stores the sum into A[12], using 48 (4 × 12) because the byte address grows with 4 each time and 12 because that is the location of the memory. as the offset and register x22 as the base register.

$sw\ \ x9,\ 48(x22)\ //\ stores\ h\ +\ A[8]\ back\ into\ A[12]$

The process of putting less frequently used variables (or those needed later) into memory is called *spilling* registers.

The hardware principle relating size and speed suggests that memory must be slower than registers, since there are fewer registers. This suggestion is indeed the case; data accesses are faster if data are in registers instead of memory. Thus, registers take less time to access *and* have higher throughput than memory, making data in registers both considerably faster to access and simpler to use. Accessing registers also uses much less energy than accessing memory. To achieve the highest performance and conserve energy, an instruction set architecture must have enough registers, and compilers must use registers efficiently.

Registers: are primitives used in hardware design that are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction. The size of a register in the RISC-V architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name word in the RISC-V architecture. (Another popular size is a group of 64 bits, called a doubleword in the RISC-V architecture.)

Word: A natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the RISC-V architecture.

Doubleword: Another natural unit of access in a computer, usually a group of 64 bits.

Constant and immediate operands

Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array.

To make this easy we have this quick add instruction with one constant operand is called add immediate or addi. To add 4 to register x22, we can just write.

$addi\ x22,\ x22,\ 4\ //\ x22\ =\ x22\ +\ 4$

By including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory. Every time there is a constant we use addi even in subtraction.

RISC-V assembly language table

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	x5 = x6 + x7	Three register operands; add
	Subtract	`sub x5, x6, x7`	x5 = x6 - x7	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	x5 = x6 + 20	Used to add constants
Data transfer	Load word	`lw x5, 40(x6)`	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	x5 = x6 << x7	Shift left by register
	Shift right logical	`srl x5, x6, x7`	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	x5 = x6 >> 3	Arithmetic shift right by immediate

FIGURE 2.1 RISC-V assembly language revealed in this chapter. This information is also found in Column 1 of the RISC-V Reference Data Card at the front of this book.

Conditional branch	Branch if equal	`beq x5, x6, 100`	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	x1 = PC+4; go to x5+100	Procedure return; indirect call

FIGURE 2.1 (Continued).

Signed and unsigned numbers COD 2.4

A single digit of a binary number is thus the “atom” of computing, since all information is composed of **binary digits** or *bits.*

binary digit: Also called bit. One of the two numbers in base 2, 0 or 1, that are the components of information.

least significant bit: The rightmost bit in an RISC-V word.

most significant bit: The leftmost bit in an RISC-V word.

If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, **overflow** is said to have occurred.

overflow when the results of an operation are larger than be represented in a register

Logical Operations COD 2.6

Instead of computers operating on full words, they now operate in fields of bits within a word, or even on individual bits.

Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation. This simplifies the packing and unpacking of bits into words. These are logical operations.

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

FIGURE 2.8 C and Java logical operators and their corresponding RISC-V instructions. One way to implement NOT is to use XOR with one operand being all ones (FFFF FFFF FFFF $FFFF_{hex}$).

Shifts (slli, srli)

The first class of such operations is called *shifts*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register x19 contained

$00000000\ 00000000\ 00000000\ 00001001_2 = 9_{10}$ and $1001 = 9$

and the instruction to shift left by 4 was executed, the new value would be:

$00000000\ 00000000\ 00000000\ 10010000_2 = 144_{10}\ and\ 10010000 = 144$

The RISC-V names for shift left and right are slli (shift left logical immediate) and srli (shift right logical immediate). This following instruction performs the operation above, if the original value was in register x19 and the result should go in register x11.

$slli\ x11,\ x19,\ 4\ //register\ x11\ =\ register\ x19\ <<\ 4\ bits$

These shift instructions use the I-type format. Since it isn't useful to shift a 32- bit register by more than 31 bits, only the lower bits of the I-type format's 12-bit immediate are actually used. The remaining bits are repurposed as an additional opcode field, funct7.

funct7	immediate	rs1	funct3	rd	opcode
0	4	19	1	11	19

I-Type: or Immediate instructions, can be either Load/Store operations, Branch operations, or Immediate ALU operations. In these instructions, one two register file locations are specified as well as a 16-bit *immediate* value which may be used as an operand or an address. in I-Type instructions the destination register is specified by *rt* (the second read from the register file is ignored).

R-Type: or Register instructions are used for register based ALU operations. The two operands and the destination of the result are specified by locations in the register file. In R-Type instructions the destination (write) register for the register file is specifies by *rd*

J-Type: or Jump instructions, devote all of the non-opcode space to a 26-bit jump destination field. The *rs* and *rt* register addresses, which are present in both R- and I-type instructions, specify the two addresses which the register file is to read.

AND: A logical bit-by- bit operation with two operands that calculates a 1 only if there is a 1 in *both* operands.

For example, if register x11 contains

$00000000\ 00000000\ 00001101\ 11000000_{two}$

and register x10 contains

$00000000\ 00000000\ 00111100\ 00000000_{two}$

then, after executing the RISC-V instruction

```
and x9, x10, x11     // reg x9 = reg x10 & reg x11
```

the value of register x9 would be

$00000000\ 00000000\ 00001100\ 00000000_{two}$

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a *mask*, since the mask “conceals” some bits.

OR: A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in *either* operand.

To place a value into one of these seas of 0s, there is the dual to AND, called **OR**. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers x10 and x11 are unchanged from the preceding example, the result of the RISC-V instruction

```
or x9, x10, x11 // reg x9 = reg x10 | reg x11
```

is this value in register x9:

$00000000\ 00000000\ 00111101\ 11000000_{two}$

NOT: A logical bit-by- bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

XOR: A logical bit-by- bit operation with two operands that calculates the exclusive OR of the two operands. That is, it calculates a 1 only if the values are different in the two operands.

If the register x10 is unchanged from the preceding example and register x12 has the value 0, the result of the RISC-V instruction

```
xor x9, x10, x12 // reg x9 = reg x10 ^ reg x12
```

is this value in register x9:

$00000000\ 00000000\ 00110001\ 11000000_{two}$

Instructions for making decisions COD 2.7

A computer is able to make a decision, this is normally done with an if statement or a go-to statement. In RISC-V we have two decision-making instructions, similar to a if statement, the first instruction is:

$beq\ rs1,\ rs2,\ L1$

This instruction means go to the statement labeled L1 if the value in register rs1 equals the value in register rs2. The mnemonic beq stands for *branch if equal*. The second instruction is:

$Bne\ rs1,\ rs2,\ L1$

It means go to the statement labeled L1 if the value in register rs1 does *not* equal the value in register rs2. The mnemonic bne stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

Conditional branch: An instruction that tests a value and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the test.

Example beq, bne

In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers x19 through x23, what is the compiled RISC-V code for this C *if* statement?

$if\ (i == j)\ f = g + h;\ else\ f = g - h;$

answer in RISC-V:

It would seem that we would want to check if the statement i is equal to j with beq, but it is easier to check if they are not equal to with bne:

$bne\ x22,\ x23,\ else\ //\ go\ to\ else\ if\ i \neq j$

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

$add\ x19,\ x20,\ x21\ //\ f = g + h\ (skipped\ if\ i \neq j)$

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch*. This instruction says that the processor always follows the branch. One way to express an unconditional branch in RISC-V is to use a conditional branch whose condition is always true:

$beq\ x0,\ x0,\ Exit\ //\ if\ 0 == 0,\ go\ to\ exit$

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label Else to this instruction. We also show the label Exit that is after this instruction, showing the end of the *if-then-else* compiled code:

$Else$:

$sub\ x19,\ x20,\ x21\ //\ f = g - h\ (skipped\ if\ i = j)$

$Exit$:

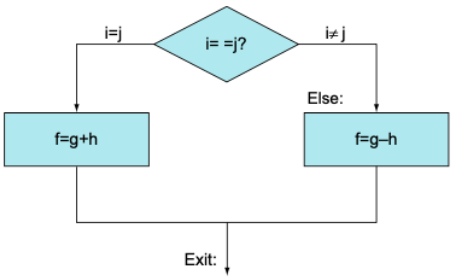


FIGURE 2.9 Illustration of the options in the *if* statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

Example while loop

Here is a traditional loop in C:

$$while\ (save[i]\ ==\ k)$$
$$i\ +=\ 1;$$

Assume that i and k correspond to registers x22 and x24 and the base of the array save is in x25. What is the RISC-V assembly code corresponding to this C code?

The first step is to load save[i] into a temporary register. Before we can load save[i] into a temporary register, we need to have its address. Before we can add i to the base of array save to form the address, we must multiply the index i by 4 due to the byte addressing issue. Fortunately, we can use shift left, since shifting left by 2 bits multiplied by 2_2 or 4

$Loop$:
$slli\ x10,\ x22,\ 2\ //\ temporary\ register\ x10\ =\ i\ \cdot 4$

To get the address of save[i], we need to add x10 and the base of save in x25:

$add\ x10,\ x10,\ x25\ //\ x10\ =\ address\ of\ save[i]$

Now we can use that address to load save[i] into a temporary register:

$lw\ x9,\ 0(x10)\ //\ loading\ save[i]\ into\ temprary\ register\ x9,\ so\ x9\ =\ save[i]$

The next instruction performs the loop test, exiting if save[i] ≠ k:

$bne\ x9,\ x24,\ Exit\ //\ if\ save[i]\ \neq\ k\ then\ go\ to\ exit$

The following instruction adds 1 to i:

$addi\ x22,\ x22,\ 1\ //\ i\ =\ i\ +\ 1$

The end of the loop branches back to the *while* test at the top of the loop. We just add the Exit label after it, and we're done:

$beq\ x0,\ x0,\ Loop\ //\ if\ 0\ ==\ 0\ go\ to\ loop$
$Exit$:

basic block: A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

blt: The branch if less than, blt instruction compares the values in registers rs1 and rs2 and takes the branch if the value in rs1 is smaller, when they are treated as two's complement numbers.

bge: branch if greater than or equal (bge) takes the branch in the opposite case, that is, if the value in rs1 is at least the value in rs2.

bltu: Branch if less than, unsigned (bltu) takes the branch if the value in rs1 is smaller than the value in rs2 when the values are treated as unsigned numbers.

bgeu: Finally, branch if greater than or equal, unsigned (bgeu) takes the branch in the opposite case.

Bounds check shortcut

Treating signed numbers as if they were unsigned gives us a low-cost way of checking if 0 ≤ *x* < *y*, which matches the index out-of-bounds check for arrays.

Use this shortcut to reduce an index-out-of-bounds check: branch to `IndexOutOfBounds` if `x20` ≥ `x11` or if `x20` is negative.

The checking code just uses unsigned greater than or equal to do both checks:

```
bgeu x20, x11, IndexOutOfBounds // if x20 >= x11 or
x20 < 0, goto IndexOutOfBounds
```

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to

implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **branch address table** or **branch table**, and the program needs only to index into the table and then branch to the appropriate sequence.

branch address table Also called **branch table**: A table of addresses of alternative instruction sequences.

The branch table is therefore just an array of double- words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the branch table into a register.

(jalr) To support such situations, computers like RISC-V include an *indirect jump* instruction, which performs an unconditional branch to the address specified in a register. In RISC-V, the jump-and-link register instruction (jalr) serves this purpose.

Exercises

2.1 For the following C statement, write the corresponding RISC-V assembly code. Assume that the C variables f, g, and h, have already been placed in registers x5, x6, and x7 respectively. Use a minimal number of RISC-V assembly instructions.

$f = g + (h - 5)$

Answer in RISC-V

$addi\ x7,\ x7,\ -5$
$add\ x5,\ x6,\ x7$

2.2 Write a single C statement that corresponds to the two RISC-V assembly instructions below.

$add\ f,\ g,\ h\ //\ i\ see\ that\ f\ is\ the\ register\ we\ want\ to\ store\ in\ then\ add\ g + h$

$add\ f,\ i,\ f\ //\ then\ we\ again\ save\ in\ register\ f,\ and\ then\ we\ want\ i + f\ where\ f\ is\ (g + h).$

Answer in C

$f = g + h + i$

2.3 For the following C statement, write the corresponding RISC-V assembly code. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively.

$B[8] = A[i - j]$

Answer in RISC-V

We start by subtracting i - j and store it in f

$sub\ x5,\ x28,\ x29$

$slli\ x5,\ x5,\ 2\ //\ f = f \cdot 4$

$If\ A\ and\ B\ are\ int\ arrays,\ the\ indexes\ have\ to\ be\ multiplied\ by\ sizeof\ int\ (4)\ to\ ob$
$tain\ the\ right\ memory\ offset$

$add\ x5,\ x5,\ x10\ //\ f = f + A$

$lw\ x5,\ 0(x5)\ //\ f = f[f]$

$sw\ x5,\ 32(x11)\ //\ mem[B + 32] = f$

2.4 For the RISC-V assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively.

```
slli  x30, x5,  2   # x30 = f * 4

add   x30, x10, x30 # x30 = &A[f]

slli  x31, x6,  2   # x31 = g * 4

add   x31, x11, x31 # x31 = &B[g]

lw    x5, 0(x30)    # f = A[f]


addi x12, x30, 4

lw   x30, 0(x12)

add  x30, x30, x5

sw   x30, 0(x31)
```

Answer in C:

B[g] = A[f+1] + A[f]

2.5 Show how the value 0xabcdef12 would be arranged in memory of a little-endian and a big-endian machine. Assume the data are stored starting at address 0 and that the word size is 4 bytes.

Little endian system stores the most least significant byte of the word at the smallest address

Address	byte
0	12
1	EF
2	DC
3	AB

A big-endian system stores the most significant byte of a word at the smallest address

Address	byte

0	AB
1	CD
2	EF
3	12

2.6 Translate 0xabcdef12 into decimal.

We want to translate the hexadecimal ABCDEF12 into decimal, firstly we want to translate the letters to number, A = 10, B = 11, C = 12, D = 13, E = 14, F = 15, 1, 2.

$$10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 1 \quad 2$$

$$16^7 \; 16^6 \; 16^5 \; 16^4 \; 16^3 \; 16^2 \; 16^1 \; 16^0$$

$$10 \cdot 268435456 = 2684354560$$

$$11 \cdot 16777216 = 184549376$$

$$12 \cdot 1048576 = 12582912$$

$$13 \cdot 65536 = 851968$$

$$14 \cdot 4096 = 57344$$

$$15 \cdot 256 = 3840$$

$$1 \cdot 16 = 16$$

$$2 \cdot 1 = 2$$

So the answer is that ABCDEF12 in decimals is

2 + 16 + 3840 + 57344 + ... = 2882400018

2.24 Consider the following RISC-V loop:

LOOP: beq x6, x0,

DONE addi x6, x6, -1

addi x5, x5, 2

jal x0, LOOP DONE:

Answer: 20

You can experimentally test this by adding the following instructions before the loop so you can run it:

```
addi    x6, x0, 10

addi    x5, x0, 0
```

2.25 Translate the following C code to RISC-V assembly code. Use a minimum number of instructions. Assume that the values of a, b, i, and j are in registers x5, x6, x7, and x29, respectively. Also, assume that register x10 holds the base address of the array D.

for(i=0; i<a; i++)

for(j=0; j<b; j++) D[4*j] = i + j;

Answer:

LOOP0: beq x7, x5, DONE0

    addi x29, x0, 0

LOOP1: beq x29, x6, DONE1

    slli x30, x29, 2

    slli x30, x30, 2

    add x30, x10, x30

    add x31, x7, x29,

    sw x31 0(x30)

    addi x29, x29, 1

    jal x0, LOOP1

DONE1:

    addi x7, x7, 1

    jal x0, LOOP0

DONE0:

Pseudoinstruktioner:

Move (mv) betyder at kopier i C, det er det samme som addi xi, xj, 0
Load immediate (li) kan skrives som addi xi, zero, k
Negate (neg) kan skrives som sub ki, zero, xi
No operation (nop) kan skrives som add zero, zero, zero
lb - load byte
sb - store byte

godbolt.org RISC-V interpreter
https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/ anden intepreter

The Von Neumann bottleneck, er vejen fra computeren til hukommelsen. Jo længere der er imellem de to kredsløb, jo længere tid er "bussen" som kører med informationen i mellem de to er langsom. Stort problem når man laver programmer i dag.

Byte ordering:

Om man bruger big endian (at den vigtigste byte, kommer først i hukommelsen) eller little endian (den vigtigste byte, kommer sidst i hukommelsen) er sådan set ligegyldigt.

Forskel på registers og memory

Memory er dynamisk adresse, hvor registeren vi opererer i er statisk kodet i instruktionerne.

Memory transfer instructions:

Note: se hvordan den dynamiske indhold af en register har indflydelse på hvilken memory adresse som vi får adgang til.

Instructions	Meaning
lb Xi, V(Xj)	Xi = Memory [Xj + V]
sb Xi, V(Xj)	Memory [Xj, V] = Xi
lw Xi, V(Xj)	Xi = Memory [Xj + V]
sw Xi, V(Xj)	Memory [Xj, V] = Xi

Program counter

Control flow:

The program counter (PC) is a special register that contains the address of the current instruction in memory.

Den kan ikke kører kontrol strukturer (if, while, for, etc)
Derfor har vi conditional jumps:
INDSÆT BILLEDE

Jal - laver et jump and link - den springer fra en linje til en anden. - lidt ligesom if else.

Loops er implementeret ved at hoppe baglæns:
INDSÆT BILLEDE

Noter til COD 2.8-2.9

2.8 Procedures/functions

Procedure or function is used to structure programs, for making them easier to read and for making it possible to reuse code.

Procedures are one way to implement abstraction in software.

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

Similarly, in the execution of a procedure, the program must follow these six steps:
1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

RISC-V

Return address - A link to the calling site that allows a procedure to return to the proper address; in RISC-V it is stored in register x1.

Caller - The program that instigates a procedure and provides the necessary parameter values.

Callee - A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

Program counter - (PC) The register containing the address of the instruction in the program being executed.

Stack - A data structure for spilling registers organized as a last-in-first-out queue.

Stack pointer - A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In RISC-V, it is register sp, or x2.

Push - Add element to stack.
Pop - Remove element from stack.

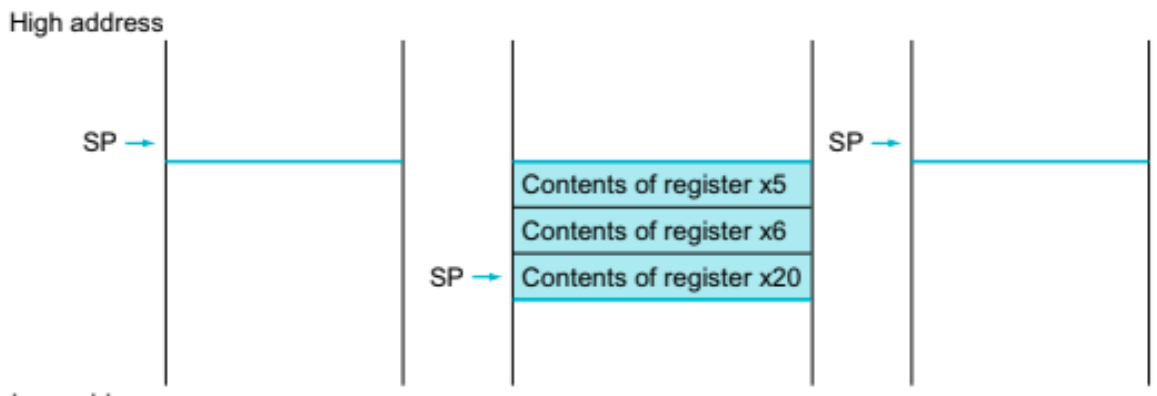
By historical precedent, stacks “grow” from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, RISC-V

Software separates 19 of the registers into two groups:

- x5−x7 and x28−x31: temporary registers that are not preserved by the callee (called procedure) on a procedure call
- x8−x9 and x18−x27: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller does not expect registers x5 and x6 to be preserved across a procedure call,



Nested procedures

Procedures that do not call others are called leaf procedures.

Nested procedures can make recursive calls, but you have to be careful with that.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register x10 and then using jal x1, A. Then suppose that procedure A calls procedure B via jal x1, B with an argument of 7, also placed in x10. Since A hasn't finished its task yet, there is a conflict over the use of register x10. Similarly, there is a conflict over the return address in register x1, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

Hardware/software interface

A C variable is generally a location in storage, and its interpretation depends both on its type and storage class. Example types include integers and characters

C has two storage classes: automatic and static.

Automatic variables - are local to a procedure and are discarded when the procedure exits.

Static variables - exist across exits from and entries to procedures.

To simplify access to static data, some RISC-V
compilers reserve a register x3 for use as the global pointer, or gp.

Global pointer - The register that is reserved to point to the static area.

(Stack) Allocating Space for New Data on the Stack

The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures.

Procedure frame - Also called activation record. The segment of the stack containing a procedure's saved registers and local variables.

Frame pointer - A value denoting the location of the saved registers and local variables for a given procedure.

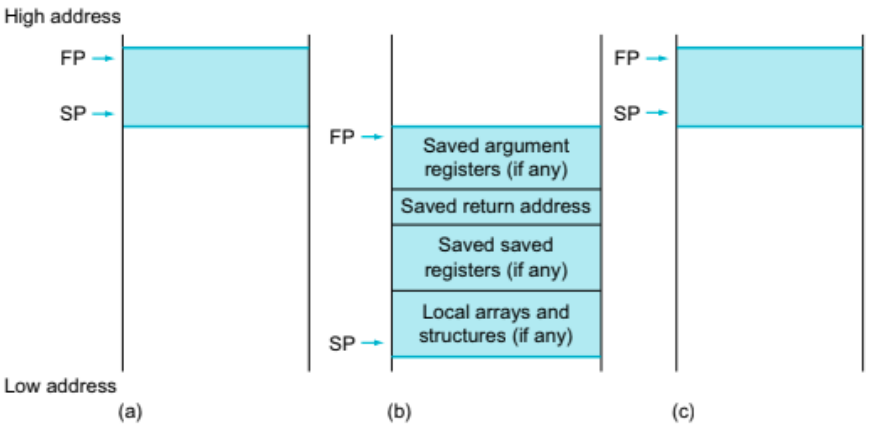


FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`fp` or `x8`) points to the first word of the frame, often a saved argument register, and the stack pointer (`sp`) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in `sp` on a call, and `sp` is restored using `fp`. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book.

(Heap) Allocating Space for New Data on the Heap

In addition to automatic variables that are local to procedures, C programmers need space in memory for static variables and for dynamic data structures.

Although arrays tend to be a fixed length and thus are a good match to the static data segment, data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the heap, and it is placed next in memory.

Text segment - The segment of a UNIX object file that contains the machine language code for routines in the source file.

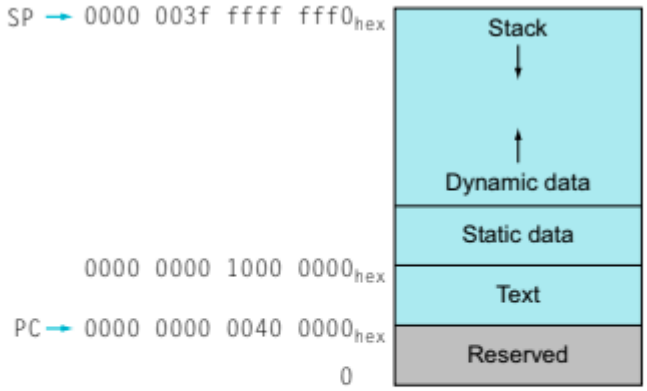


FIGURE 2.13 The RISC-V memory allocation for program and data. These addresses are only a software convention, and not part of the RISC-V architecture. The stack pointer is initialized to 0000 003f ffff fff0$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0000 0000 0040 0000$_{hex}$. The static data starts immediately after the end of the text segment; in this example, we assume that address is 0000 0000 1000 0000$_{hex}$. Dynamic data, allocated by `malloc` in C and by `new` in Java, is next. It grows up toward the stack in an area called the *heap*. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book.

C allocates and frees space on the heap with explicit functions. malloc() allocates space on the heap and returns a pointer to it, and free() releases space on the heap to which the pointer points.

RISC-V register conventions

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

FIGURE 2.14 RISC-V register conventions. This information is also found in Column 2 of the RISC-V Reference Data Card at the front of this book.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

2.9 communicating with people

Load byte unsigned (lbu) - loads a byte from memory, placing it in the rightmost 8 bits of a register.

Store byte (sb) - takes a byte from the rightmost 8 bits of a register and writes it to memory.

The RISC-V instruction set has explicit instructions to load and store such 16-bit quantities, called halfwords. Load half unsigned loads a halfword from memory, placing it in the rightmost 16 bits of a register, filling the leftmost 16 bits with zeros.

Like load byte, load half (lh) - treats the halfword as a signed number and thus sign-extends to fill the 16 leftmost bits of the register.

Store half (sh) - takes a halfword from the rightmost 16 bits of a register and writes it to memory.

Computer Arithmetic COD 3.1-3.3 3.5

Two´s complement

Two's complement is a way to represent negative numbers in binary. The main benefit of using two's complement is that it allows for the efficient representation and manipulation of both positive and negative numbers using the same set of bit patterns. Additionally, it simplifies the logic for basic arithmetic operations such as addition and subtraction, as the same circuits can be used for both signed and unsigned numbers. This makes it a popular choice for use in digital systems and computer architectures.

Then we first find the 1´s complement which is the binary number inverted.
1111 0011 1111 1111 1111 1111 1111 1111
When we want to find 2´s complement we add a 1 to the least significant bit.

```
  1111 0011 1111 1111 1111 1111 1111 1111
+ 0000 0000 0000 0000 0000 0000 0000 0001
------------------------------------------------------
= 1111 0011 1111 1111 1111 1111 1111 1110
```

3.2 Addition and subtraction

Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: the appropriate operand is simply negated before being added.

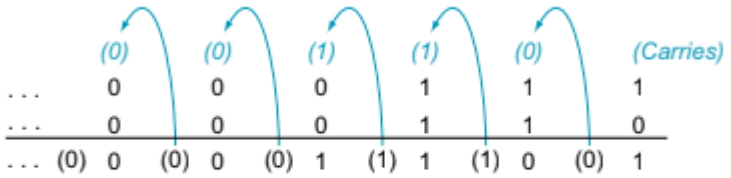


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Addition:

```
  0      1    1 carries over so it gets a 1 on the left, because 10 = 2 in binary
+ 0    + 1
= 0      1 0

  1      1
+ 0      1
```

$= 1 \quad + 1$
$1\ 1$
For example:

1010
$+\ 1001$
10011

100110
$+\ 110101$
1011011

Subtraction:

$1 \quad 1$
$-\ 0 \quad -\ 1$
$= 1 \quad = 0$
If we want to do 0-1 thats not possible then we borrow a 1, like in this example.
$\downarrow 2$
10
$-\ 1$
$= 0\ 1$

Recall that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word. When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, −10 + 4 = −6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.2 Overflow conditions for addition and subtraction.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result.

Addition has overflowed if the sum is less than either of the addends, whereas subtraction has overflowed if the difference is greater than the minuend.

Arithmetic Logic Unit (ALU): Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.

A major point of this section is that, independent of the representation, the finite word size of computers means that arithmetic operations can create results that are too large to fit in this fixed word size.

3.3 Multiplication

$$
\begin{array}{lll}
\text{Multiplicand} & & 1000_{ten} \\
\text{Multiplier} & \times & 1001_{ten} \\
\hline
 & & 1000 \\
 & & 0000 \\
 & & 0000 \\
 & & 1000 \\
\hline
\text{Product} & & 1001000_{ten}
\end{array}
$$

The first operand is called the multiplicand and the second the multiplier. The final result is called the product.

The first observation is that the number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier. In fact, if we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n + m bits long. That is, n + m bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

In this example, we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

1. Just place a copy of the multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1, or

2. Place 0 (0 × multiplicand) in the proper place if the digit is 0.

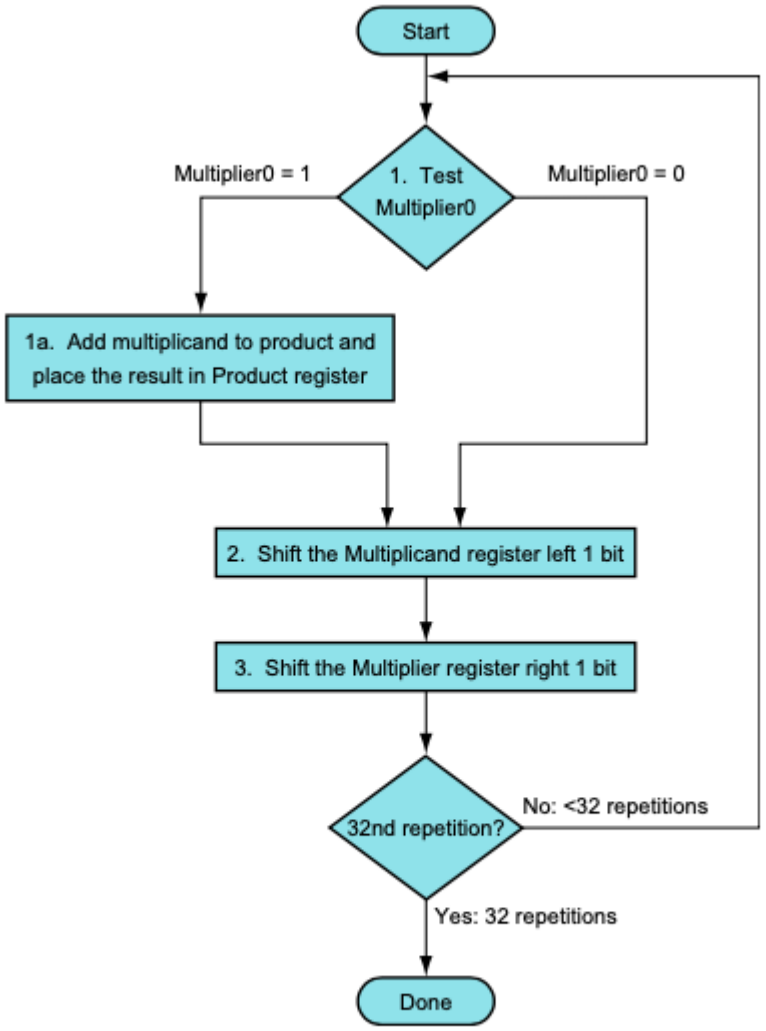


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

RISC-V multiplikation

To produce a properly signed or unsigned 64-bit product, RISC-V has four instructions: *multiply* (mul), *multiply high* (mulh), *multiply high unsigned* (mulhu), and *multiply high*

signed-unsigned (mulhsu). To get the integer 32-bit product, the programmer uses mul. To get the upper 32 bits of the 64-bit product, the programmer uses (mulh) if both operands are signed, (mulhu) if both operands are unsigned, or (mulhsu) if one operand is signed and the other is unsigned.

3.5 Floating points

The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point.

$1.0_{ten} \; x \; 10^{-9}$ betyder det samme.

Scientific notation - A notation that renders numbers with a single digit to the left of the decimal point.

Normalized - A number in floating-point notation that has no leading 0s.

A standard scientific notation for reals in the normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since real digits to the right of the binary point replace the unnecessary leading 0s.

Floating point - Computer arithmetic that represents numbers in which the binary point is not fixed. as it is for integers.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$1.0_{two} \cdot 2^{-1}$

To keep a binary number in the normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; *binary point* will do fine.

Floating-Point Representation

A designer of a floating-point representation must find a compromise between the size of the fraction and the size of the exponent, because a fixed word size

means you must take a bit from one to give a bit to the other. This tradeoff is between *precision* and *range*: increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented.

Fraction - The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the mantissa.

Exponent - In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

Floating-point numbers are usually a multiple of the size of a word.

The representation of a RISC-V floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), *exponent* is the value of the 8-bit exponent field (including the sign of the exponent), and *fraction* is the 23-bit number. As we recall from Chapter 2, this representation is *sign and magnitude*, since the sign is a separate bit from the rest of the number.

In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

F involves the value in the fraction field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon.

Overflow (floating-point) - A situation in which a positive exponent becomes too large to fit in the exponent field.

Underflow (floating-point) - A situation in which a negative exponent becomes too large to fit in the exponent field.

One way to reduce the chances of underflow or overflow is to offer another format that has a larger exponent. In C, this number is called *double*, and operations on doubles are called **double precision** floating-point arithmetic; **single precision** floating point is the name of the earlier format.

The representation of a double precision floating-point number takes one RISC-V doubleword, as shown below, where *s* is still the sign of the number, *exponent* is the value of the 11-bit exponent field, and *fraction* is the 52-bit number in the fraction field.

Double precision - A floating-point value represented in a 64-bit doubleword.

Single precision - A floating-point value represented in a 32-bit word.

Exception - Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow, for example.

Interrupt - An exception that comes from outside of the processor. (Some architectures use the term interrupt for all exceptions.)

Expectations and interrupts

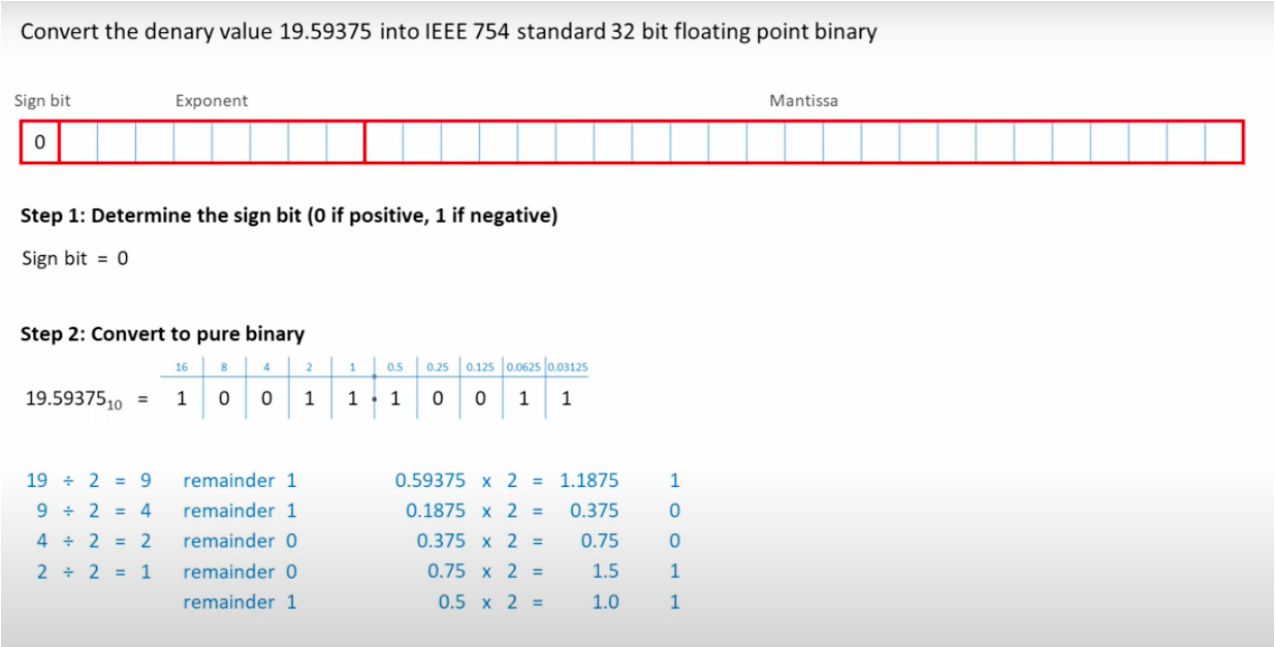
What should happen on an overflow or underflow to let the user know that a problem occurred? Some computers signal these events by raising an **exception**, sometimes called an **interrupt**. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

IEEE 754 Floating-Point Standard

The IEEE 754 is a floating point standard that all the computers in the world uses. Its practical because it can hold more/greater values than decimal numbers. It consists of a signed bit (1 bit), the exponent (8 bits) and the Mantissa (23 bits) in single precision (32 bits), and in double precision (64 bits) it has signed bit (1 bit), Exponent (11 bits) and the mantissa (52 bits).

IEEE 754 Standard for Single Precision 32 bit floating point binary

Now we want to look into how we convert a floating point into the standard IEEE 754, with single precision. If we have the number 19.59375, we fist determine the signed bit, which is 0 because the number we are working with is positive. that zero is placed in the signed bit.



Then we have to convert the number into binary numbers, like in step 2. so decimal to binary. We do this by dividing the exponent 19 with 2 and then for the floating part .59375 we have to multiply it with 2, until the fraction is 0.

Then for step 3, we want to move the binary point, so only one none-zero bit is remained in front of it. Therefor we move 4 times to the left and then we get the number

$$1.001110011 \cdot 2^4$$

Then we determine the biased exponent by taking the 4 and add it to 127, which is 131 and then make it into a binary number.

IEEE 754 Format	Sign	Exponent	Mantissa	Exponent Bias
32 bit single precision	1 bit	8 bits	23 bits (+ 1 not stored)	$2^{(8-1)} - 1 = 127$
64 bit double precision	1 bit	11 bits	52 bits (+ 1 not stored)	$2^{(11-1)} - 1 = 1023$
128 bit quadruple precision	1 bit	15 bits	112 bits (+ 1 not stored)	$2^{(15-1)} - 1 = 16383$

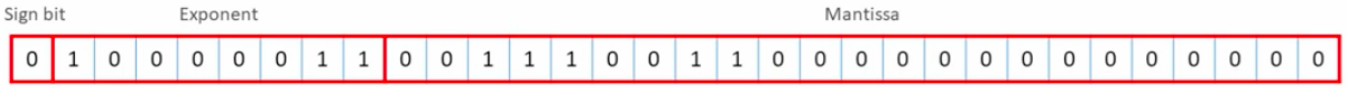
$4 + 127 = 131 = 10000011$

atlast remove the leading 1 from the mantissa so that $1.001110011 = 001110011$

We know that the 1 belongs there if we want to use the number for calculating something.

Then the last thing to do is to path out the rest of the mantissa with 0.

Convert the denary value 19.25 into IEEE 754 standard 32 bit floating point binary



Step 1: Determine the sign bit (0 if positive, 1 if negative)

Sign bit = 0

Step 2: Convert to pure binary

	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125
19.59375_{10} =	1	0	0	1	1	1	0	0	1	1

Step 3: Normalise to determine the mantissa and the unbiased exponent *(place the binary point after leftmost 1)*

1 0 0 1 1 . 1 0 0 1 1 = 1.001110011×2^4

Step 4: Determine the biased exponent *(add 127 then convert to an unsigned binary integer)*

$4 + 127 = 131_{10} = 10000011_2$

Step 5: Remove the leading 1 from the mantissa *(remove the leftmost 1)*

1.001110011 = 001110011

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{ten} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{ten} \times 10^{1}$. Thus, the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really

$$0.016 \times 10^{1}$$

Step 2. Next comes the addition of the significands:

$$\begin{array}{rr} & 9.999_{ten} \\ + & 0.016_{ten} \\ \hline & 10.015_{ten} \end{array}$$

The sum is $10.015_{ten} \times 10^{1}$.

Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{ten} \times 10^{1} = 1.0015_{ten} \times 10^{2}$$

After the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4. Since we assumed that the significand could be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$1.0015_{ten} \times 10^{2}$$

is rounded to four digits in the significand to

$$1.002_{ten} \times 10^{2}$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

Floating point multiplikation side 218 i COD

Floating-Point Instructions in RISC-V

RISC-V supports the IEEE 754 single-precision and double-precision formats with these instructions:

- Floating-point addition, single (fadd.s) and addition, double (fadd.d)
- Floating-point subtraction, single (fsub.s) and subtraction, double (fsub.d)
- Floating-point multiplication, single (fmul.s) and multiplication, double (fmul.d)
- Floating-point division, single (fdiv.s) and division, double (fdiv.d)
- Floating-point square root, single (fsqrt.s) and square root, double (fsqrt.d)
- Floating-point equals, single (feq.s) and equals, double (feq.d)
- Floating-point less-than, single (flt.s) and less-than, double (flt.d)
- Floating-point less-than-or-equals, single (fle.s) and less-than-or-equals, double (fle.d)

RISC-V floating-point operands

32 floating-point registers	f0-f31	An f-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

Arithmetic	FP add single	fadd.s f0, f1, f2	f0 = f1 + f2	FP add (single precision)
	FP subtract single	fsub.s f0, f1, f2	f0 = f1 - f2	FP subtract (single precision)
	FP multiply single	fmul.s f0, f1, f2	f0 = f1 * f2	FP multiply (single precision)
	FP divide single	fdiv.s f0, f1, f2	f0 = f1 / f2	FP divide (single precision)
	FP square root single	fsqrt.s f0, f1	f0 = √f1	FP square root (single precision)
	FP add double	fadd.d f0, f1, f2	f0 = f1 + f2	FP add (double precision)
	FP subtract double	fsub.d f0, f1, f2	f0 = f1 - f2	FP subtract (double precision)
	FP multiply double	fmul.d f0, f1, f2	f0 = f1 * f2	FP multiply (double precision)
	FP divide double	fdiv.d f0, f1, f2	f0 = f1 / f2	FP divide (double precision)
	FP square root double	fsqrt.d f0, f1	f0 = √f1	FP square root (double precision)
Comparison	FP equality single	feq.s x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (single precision)
	FP less than single	flt.s x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (single precision)
	FP less than or equals single	fle.s x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (single precision)
	FP equality double	feq.d x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (double precision)
	FP less than double	flt.d x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (double precision)
	FP less than or equals double	fle.d x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (double precision)
Data transfer	FP load word	flw f0, 4(x5)	f0 = Memory[x5 + 4]	Load single-precision from memory
	FP load doubleword	fld f0, 8(x5)	f0 = Memory[x5 + 8]	Load double-precision from memory
	FP store word	fsw f0, 4(x5)	Memory[x5 + 4] = f0	Store single-precision from memory
	FP store doubleword	fsd f0, 8(x5)	Memory[x5 + 8] = f0	Store double-precision from memory

FIGURE 3.17 RISC-V floating-point architecture revealed thus far. This information is also found in column 2 of the RISC-V Reference Data Card at the front of this book.

Accurate Arithmetic

Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can't really represent.

Guard - The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

Round - Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format. It is also the name of the second of two extra bits kept on the right during intermediate floating-point calculations, which improves rounding accuracy.

Units in the last place (ulp) - The number of bits in error in the least significant bits of the significand between the actual number and the number that can be represented.

Sticky bit - A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

Fused multiply add - A floating-point instruction that performs both a multiply and an add, but rounds only once after the add.

Exercises

3.1 What is 5ED4 − 07A4 when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

Answer:

$5ED4 \;=\; 0101\ 1110\ 1101\ 0100$

$07A4 \;=\; 0000\ 0111\ 1010\ 0100$

$(-)$ ━----------

$=\qquad 0101\ 0111\ 0011\ 0000$

0 7 10 4

$8,\ 4,\ 2,\ 1 \quad 8,\ 4,\ 2,\ 1 \quad 8,\ 4,\ 2,\ 1 \quad 8,\ 4,\ 2,\ 1 \;=\; 0000\ 0111\ 1010\ 0100$

$0\ \ 0\ \ 0\ \ 0 \quad 0\ \ 1\ \ 1\ \ 1 \quad 1\ \ 0\ \ 1\ \ 0 \quad 0\ \ 1\ \ 0\ \ 0$

3.3 Convert 5ED4 into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

Answer:

First I convert to the value of E and D, E = 14, and D = 13. then we find out which bits we need to make the number and put 1's there.

$$5 \qquad 14 \qquad 13 \qquad 4$$

$$8,4,2,1 \quad 8,4,2,1 \quad 8,4,2,1 \quad 8,4,2,1 \quad = 0101\ 1110\ 1101\ 0100$$

$$0\ 1\ 0\ 1 \quad 1\ 1\ 1\ 0 \quad 1\ 1\ 0\ 1 \quad 0\ 1\ 0\ 0$$

Hex numbers are **compact and use less memory**, so more numbers can be stored in computer systems. Their small size also makes input-output handling easier compared to other numbering formats. Because it's easy to convert hexadecimal to binary and vice versa, the system is widely used in computer programming.

3.20 What decimal number does the bit pattern 0 × 0C000000 represent if it is a two's complement integer? An unsigned integer?

Answer:

Step 1: Determine the value of each bit using the following formula: bit value = 2^(bit position)
For example, the value of the leftmost (most significant) bit is 2^7 = 128, the value of the next bit is 2^6 = 64, and so on.

Step 2: Starting from the leftmost (most significant) bit, add up the values of all the 1 bits.

For the bit pattern 0 × 0C000000, the value of each bit is as follows:

bit 7: $0 \cdot 2^7 = 0$

bit 6: $1 \cdot 2^6 = 64$

bit 5: $0 \cdot 2^5 = 0$

bit 4: $0 \cdot 2^4 = 0$

bit 3: $0 \cdot 2^3 = 0$

bit 2: $0 \cdot 2^2 = 0$

bit 1: $0 \cdot 2^1 = 0$

bit 0: $0 \cdot 2^0 = 0$

The total value is 64.

Step 3: If the leftmost bit is a 1, subtract the value of the leftmost bit from the total to get the negative two's complement value.

Since the leftmost bit is a 0 in this case, the decimal value of the bit pattern in two's complement representation is 64.

Unsigned bit:
To find the decimal value of the bit pattern in unsigned integer representation, we can follow a similar process, but we do not consider the leftmost (most significant) bit to be a sign bit. Instead, we treat it as a regular bit with a value of 2^7 = 128.

Step 1: Determine the value of each bit using the following formula: bit value = 2^(bit position)
For example, the value of the leftmost (most significant) bit is 2^7 = 128, the value of the next bit is 2^6 = 64, and so on.

Step 2: Starting from the leftmost (most significant) bit, add up the values of all the 1 bits.

For the bit pattern 0 × 0C000000, the value of each bit is as follows:

bit 7: $0 \cdot 2^7 = 0$

bit 6: $1 \cdot 2^6 = 64$

bit 5: $0 \cdot 2^5 = 0$

bit 4: $0 \cdot 2^4 = 0$

bit 3: $0 \cdot 2^3 = 0$

bit 2: $0 \cdot 2^2 = 0$

bit 1: $0 \cdot 2^1 = 0$

bit 0: $0 \cdot 2^0 = 0$

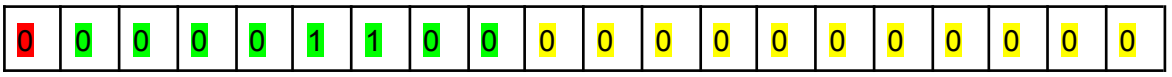
The total value is 128 + 64 = 192.
The decimal value of the bit pattern in unsigned integer representation is 192.

3.22 What decimal number does the bit pattern 0 × 0C000000 represent if it is a floating point number? Use the IEEE 754 standard.

Answer:

To figure out the decimal number represented by a bit pattern using the IEEE 754 standard, we need to follow these steps:

1. Determine the sign bit: In this case, the sign bit is 0, which means the number is positive.



Convert the Hexadecimal into binary number:

12

8 4 2 1

1 1 0 0

Then insert all the zeros we have 0000 1100 0000 0000 0000 0000 0000 0000

2. Determine the exponent: The exponent is determined by the next 8 bits, which in this case is 00001100. This represents the exponent 12 in binary.

3. Determine the mantissa: The mantissa is determined by the remaining bits, which in this case is 00000000. This represents the mantissa 0 in binary.

4. Calculate the number: Using the signed bit, the exponent, and the mantissa, we can now calculate the decimal number.
   The formula for this is (-1)^s * 2^(e-127) * (1 + m),
   where s is the sign bit (0 for positive, 1 for negative), e is the exponent, and m is the mantissa.

Plugging in the values from our bit pattern, we get
$(-1)^{0} \cdot 2^{(12-127)} \cdot (1+0) = 2.4074124e^{-35}$
This is a very small number, so we can round it down to -3.

3.23 Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

Answer

First we see that it's the single precision format, and the decimal number is 63.25.

Step 1. the signed bit is 0 because the number is positive

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

There are only 20 bit columns but it should be 32 bits.

step 2. Then we have to convert 63.25 into a binary number.

first we do 63, 63-32 = 31

31 - 16 = 15

15 - 8 = 7

7 - 4 = 3

3 - 2 = 1

1 - 1 = 0

$128 \quad 64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1$

$0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1$

know we find the binary number for .25

$0.5 \quad 0.25 \quad 0.125 \quad 0.0625 \quad 0.03125$

$0 \quad 1 \quad 0 \quad 0 \quad 0$

Step 3:

Then we move the . so it goes when there is only a one in the front.

00111111.0100

001.111110100

Step 4:

We moved the . 5 times so from our table under IEEE 754, we see that we should add 5 to 127.

$5 + 127 = 132 = 10000100$ now we want to find 132 in binary

$132 - 128 = 4$

$4 - alle\,tal = 0$

$4 - 4 = 0$

The rest is also 0

$128\ \ 64\ \ 32\ 16\ \ 8\ \ 4\ \ 2\ \ 1$

$1\ \ \ 0\ \ \ 0\ \ \ 0\ \ \ 0\ 1\ 0\ \ 0$

this is the exponent

Step 5:

Remove the leading 1 in the mantissa

$1.1111101 = 1111101$

and insert the missing 0.

3.24 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

Answer

Step 1:

the number is positive so the signed bit is 0

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

There are only 20 bit columns but it should be 64 bits.

step 2. Then we have to convert 63.25 into a binary number.

first we do 63, 63-32 = 31

31 - 16 = 15

15 - 8 = 7

7 - 4 = 3

3 - 2 = 1

1 - 1 = 0

$128 \;\; 64 \;\; 32 \;\; 16 \;\; 8 \;\; 4 \;\; 2 \;\; 1$

$0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1$

know we find the binary number for .25

$0.5 \;\; 0.25 \;\; 0.125 \;\; 0.0625 \;\; 0.03125$

$0 \quad 1 \quad 0 \quad 0 \quad 0$

Step 3:

Then we move the . so it goes when there is only a one in the front.

00111111.0100

001.111110100

Step 4:

We moved the . 5 times so from our table under IEEE 754, we see that we should add 5 to 1023.

$5 + 1023 = 1028 = 10000100$ now we want to find 132 in binary

$1028 - 1024 = 4$

$4 - alle\, tal = 0$

$4 - 4 = 0$

The rest is also 0

$1024\, 512 \;\; 256 \;\; 128 \;\; 64 \;\; 32 \;\; 16 \;\; 8 \;\; 4 \;\; 2 \;\; 1$

$1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \; 1 \; 0 \; 0$

this is the exponent

Step 5:

Remove the leading 1 in the mantissa

$1.1111101 = 1111101$

and insert the missing 0.

3.27 IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa is 10 bits long. A hidden 1 is assumed. Write down the bit pattern to represent $-1.5625 \times 10_{-1}$ assuming a version of this format, which uses an excess-16 format to store the exponent. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

Answer:

The bit pattern for representing $-\ 1.5625 \cdot 10^{-1}$ in a 16-bit floating point format would be:

1 01111 00000 00000

The sign bit is 1 because the number is negative. The exponent is 01111, which corresponds to a value of 15 in excess-16 format. The mantissa is 00000, which represents the hidden 1.

This 16-bit floating point format has a smaller range and lower accuracy compared to the single precision IEEE 754 standard, which uses a 32-bit format. The single precision format has a 23-bit mantissa, which allows for greater precision in representing decimal values. It also has a larger exponent range, which allows it to represent a wider range of values overall.

Memory hierarchy and caching COD 5.1-5.4

You are sitting at a desk in a library with a collection of books that you have pulled from the shelves and are examining. You find that several of the important computers that you need to write about are described in the books you have, but there is nothing about the EDSAC. Therefore, you go back to the shelves and look for an additional book. You find a book on early British computers that covers the EDSAC. Once you have a good selection of books on the desk in front of you, there is a high probability that many of the topics you need can be found in them, and you may spend most of your time just using the books on the desk without returning to the shelves. Having several books on the desk in front of you saves time compared to having only one book there and constantly having to go back to the shelves to return it and take out another.

The same principle allows us to create the illusion of a large memory that we can access as fast as a very small memory.

This *principle of locality* underlies both the way in which you did your work in the library and the way that programs operate. There are two different types of locality:

Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon. If you recently brought a book to your desk to look at, you will probably need to look at it again soon.

Temporal locality refers to the phenomenon where recently accessed data is likely to be accessed again in the near future. A common example of temporal locality in C is when a program repeatedly iterates through a large array or buffer.

Here's an example of a program in C that uses temporal locality:

```c
#include <stdio.h>

int main() {
    int buffer[100000];

    // initialize the buffer with some values
    for (int i = 0; i < 100000; i++) {
        buffer[i] = i;
    }

    // repeatedly access the buffer in a loop
    for (int i = 0; i < 10000; i++) {
        for (int j = 0; j < 100000; j++) {
            int x = buffer[j];
            // do something with x
        }
    }

    return 0;
}
```

In this example, the program repeatedly iterates through the buffer array in a nested loop. This creates a high degree of temporal locality, as each element of the array is accessed multiple times in succession. The processor's cache can make use of this temporal locality by keeping recently accessed data in a small, fast cache, so that if the same data is accessed again, it can be quickly retrieved from the cache instead of having to fetch it from main memory.

This makes access to the buffer faster and it increases the overall performance of the application.

Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon. For example, when you brought out the book on early English computers to learn about the EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you likewise brought back that book and, later on, found something useful in that book. Libraries put books on the same topic together on the same shelves to increase spatial locality. We'll see how memory hierarchies use spatial locality a little later in this chapter.

Spatial locality refers to the phenomenon where data that is stored near each other in memory is likely to be accessed together. A common example of spatial locality in C is when a program iterates through a large two-dimensional array by row or by column.

Here's an example of a program that uses spatial locality:

```c
#include <stdio.h>

int main() {
    int grid[1000][1000];

    // initialize the grid with some values
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 1000; j++) {
            grid[i][j] = i * 1000 + j;
        }
    }

    // iterate through the grid by row
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 1000; j++) {
            int x = grid[i][j];
            // do something with x
        }
    }

    return 0;
}
```

In this example, the program iterates through the grid array in a nested loop, first by row and then by column. This creates a high degree of spatial locality, as each element of the array is accessed immediately after its neighbors in the same row. The processor's cache can take advantage of this spatial locality by keeping recently accessed data in a small, fast cache

and since the data is accessed in a linear order, the chances of having the next accessed value in the cache are high. This can greatly improve the performance of the application.

Alternatively, the program could also iterate through the grid by column:

```
for (int j = 0; j < 1000; j++) {
    for (int i = 0; i < 1000; i++) {
        int x = grid[i][j];
        // do something with x
    }
}
```

The principle is the same, though the program will now have spatial locality along the columns, instead of the rows.

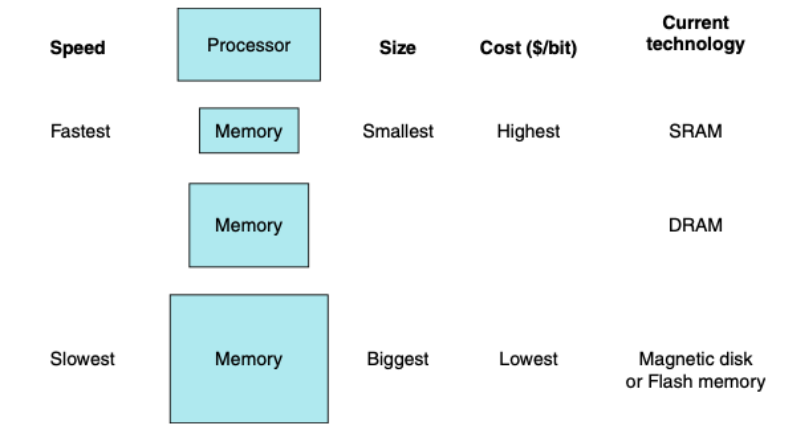


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

For example, most programs contain loops, so instructions and data are likely to be access repeatedly, showing large temporal locality. Since instructions are normally accessed sequentially, programs also show high spatial locality. Access to data also exhibits a natural spatial locality.

For example, sequential access to elements of an array or a record will naturally have high degrees of spatial locality.

We take advantage of the principle of locality by implementing the memory of a computer as a **memory hierarchy**.

Memory hierarchy: A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase while the cost per bit decreases.

A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller.

A memory hierarchy can consist of multiple levels, but data are copied between only two adjacent levels at a time, so we can focus our attention on just two levels.

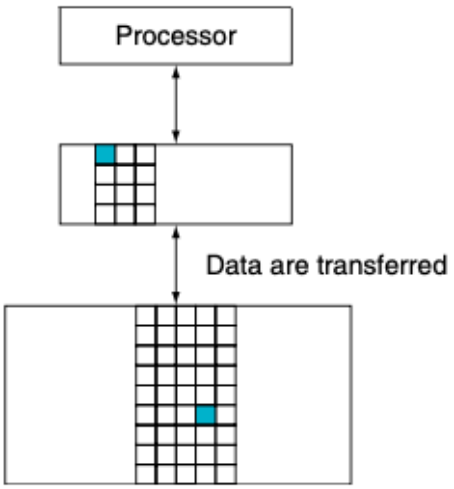


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

The upper level—the one closer to the processor—is smaller and faster than the lower level, since the upper level uses technology that is more expensive. Figure 5.2 shows that the minimum unit of information that can be either present or not present in the two-level hierarchy is called a **block** or a **line**; in our library analogy, a block of information is one book.

Block (or line): The minimum unit of information that can be either present or not present in a cache.

If the data requested by the processor appear in some block in the upper level, this is called a hit. (if you found the information in one of the books on your desk). If the data are not found in the upper level, the request is called a miss. The lower level in the hierarchy is then accessed to retrieve the block containing the requested data. (Continuing our analogy, you go from your desk to the shelves to find the desired book.)

Hit rate: The fraction of memory accesses found in a level of the memory hierarchy. it is often used as a measure of the performance of the memory hierarchy.

Miss rate: (1−hit rate) The fraction of memory accesses not found in a level of the memory hierarchy.

Since performance is the major reason for having a memory hierarchy, the time to service hits and misses is important

Hit time: The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss. (that is, the time needed to look through the books on the desk).

Miss penalty: The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.

Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty. (The time to examine the books on the desk is much smaller than the time to get a new book from the shelves.)

Programs exhibit both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items. Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.

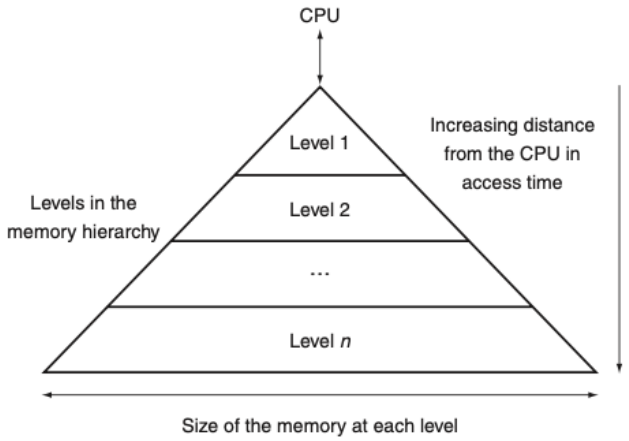


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although local storage is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

Memory Technologies

There are four primary technologies used today in memory hierarchies. Main memory is implemented from DRAM (*dynamic random access memory*), while levels closer to the processor (caches) use SRAM (*static random access memory*). DRAM is less costly per bit than SRAM, although it is substantially slower.

The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon; the speed difference arises from several factors

The third technology is flash memory. This nonvolatile memory is the secondary memory in Personal Mobile Devices.

The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 2020.

Memory technology	Typical access time	$ per GiB in 2020
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$3–$6
Flash semiconductor memory	5,000–50,000 ns	$0.06–$0.12
Magnetic disk	5,000,000–20,000,000 ns	$0.01–$0.02

SRAM Technology

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write.

SRAMs have a fixed access time to any datum, though the read and write access times may differ.

SRAM unlike DRAM doesn't need to be refreshed, this results in better performance and a lower power use. But it requires more space and is therefore more expensive, this also means that we don't use it for the computer's main memory. In a SRAM, as long as power is applied, the value can be kept indefinitely.

SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read.

In the past, most PCs and server systems used separate SRAM chips for either their primary, secondary, or even tertiary caches. Today, thanks to Moore’s Law, all levels of caches are integrated onto the processor chip, so the market for independent SRAM chips has nearly evaporated.

DRAM Technology

Dynamic RAM, or DRAM is a form of random access memory, RAM which is used in many processor systems to provide the working memory.

DRAM is widely used in digital electronics where low-cost and high-capacity memory is required.

Dynamic RAM, DRAM is used where very high levels of memory density are needed, although against this it is quite power hungry so this needs to be considered if it is to be used.

It stores each bit of data on a small capacitor within the memory cell. In a *dynamic RAM* (DRAM), the value kept in a cell is stored as a charge in a capacitor. The capacitor can be either charged or discharged and this provides the two states, "1" or "0" for the cell. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only one transistor per bit of storage, they are much denser and cheaper per bit than SRAM.

Since the charge within the capacitor leaks, it is necessary to refresh each memory cell periodically. As DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed. This refresh requirement gives rise to the term dynamic - static memories do not have a need to be refreshed. To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds.

Fortunately, DRAMs use a two-level decoding structure, and this allows us to refresh an entire *row* (which shares a word line) with a read cycle followed immediately by a write cycle.

The advantage of a DRAM is the simplicity of the cell - it only requires a single transistor compared to around six in a typical static RAM, SRAM memory cell. In view of its simplicity, the costs of DRAM are much lower than those for SRAM, and they are able to provide much higher levels of memory density. However the DRAM has disadvantages as well, and as a result, most computers use both DRAM technology and SRAM, but in different areas.

To improve the interface to processors further, DRAMs added clocks and are properly called synchronous DRAMs or SDRAMs. The advantage of SDRAMs is that the use of a clock eliminates the time for the memory and processor to synchronize.

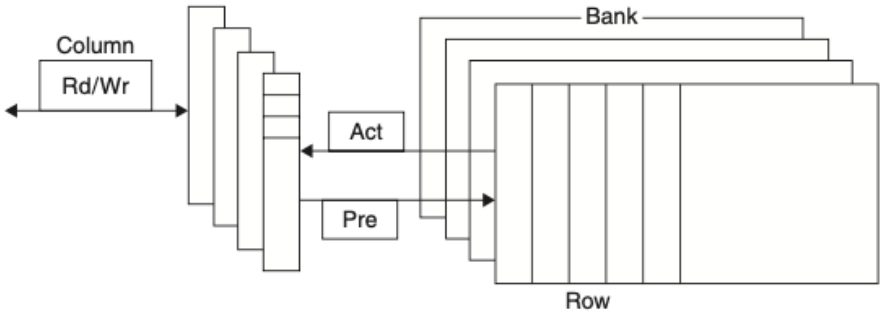


FIGURE 5.4 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR4. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR4 or by specifying a block transfer and the starting address. Each command, as well as block transfers, is synchronized with a clock.

Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$6,480,000	250 ns	150 ns
1983	256 Kibibit	$1,980,000	185 ns	100 ns
1985	1 Mebibit	$720,000	135 ns	40 ns
1989	4 Mebibit	$128,000	110 ns	40 ns
1992	16 Mebibit	$30,000	90 ns	30 ns
1996	64 Mebibit	$9,000	60 ns	12 ns
1998	128 Mebibit	$900	60 ns	10 ns
2000	256 Mebibit	$840	55 ns	7 ns
2004	512 Mebibit	$150	50 ns	5 ns
2007	1 Gibibit	$40	45 ns	1.25 ns
2010	2 Gibibit	$13	40 ns	1 ns
2012	4 Gibibit	$5	35 ns	0.8 ns
2015	8 Gibibit	$7	30 ns	0.6 ns
2018	16 Gibibit	$6	25 ns	0.4 ns

FIGURE 5.5 DRAM size increased by multiples of four approximately once every 3 years until 1996, and thereafter considerably slower. The improvements in access time have been slower but continuous, and cost roughly tracks density improvements, although cost is often affected by other issues, such as availability and demand. The cost per gibibyte is not adjusted for inflation. Price source is https://jcmit.net/memoryprice.htm.

The speed advantage of synchronous DRAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits. Instead, the clock transfers the successive bits in a burst. It is called *Double Data Rate* (DDR) SDRAM.

The name means data transfers on both the rising *and* falling edge of the clock, thereby getting twice as much bandwidth as you might expect based on the clock rate and the data width.

Sustaining that much bandwidth requires clever organization *inside* the DRAM. Instead of just one faster row buffer, the DRAM can be internally organized to read or write from multiple *banks*, with each having its own row buffer. Sending an address to several banks permits them all to read or write simultaneously. For example, with four banks, there is just one access time and then accesses rotate between the four banks to supply four times the bandwidth. This rotating access scheme is called *address interleaving*.

Flash memory

Flash memory is a type of *electrically erasable programmable read-only memory* (EEPROM).

Flash memory is used for easy and fast information storage in computers, digital cameras and home video game consoles. It is used more like a hard drive than as RAM. In fact, flash memory is known as a solid state storage device, meaning there are no moving parts -- everything is electronic instead of mechanical.

It has a grid of columns and rows with a cell that has two transistors at each intersection.

Unlike disks and DRAM, but like other EEPROM technologies, writes can wear out flash memory bits. To cope with such limits, most flash products include a controller to spread the writes by remapping blocks that have been written many times to less trodden blocks. This technique is called *wear leveling*. With wear leveling, personal mobile devices are very unlikely to exceed the write limits in the flash.

Disk Memory

As Figure 5.6 shows, a magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute. The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or videotape.

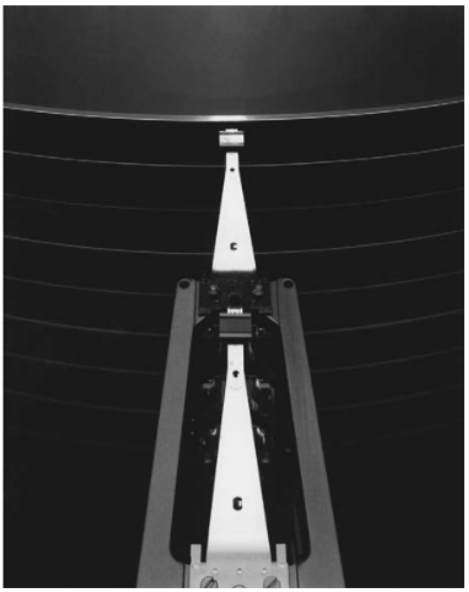


FIGURE 5.6 A disk showing 10 disk platters and the read/write heads. The diameter of today's disks is 2.5 or 3.5 inches, and there are typically one or two platters per drive today.

To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a *read-write head* is located just above each surface. The entire

drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface.

Each disk surface is divided into concentric circles, called **tracks**. There are typically tens of thousands of tracks per surface. Each track is in turn divided into **sectors** that contain the information; each track may have thousands of sectors.

Track: One of thousands of concentric circles that make up the surface of a magnetic disk.

Sector: One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk. Sectors are typically 512 to 4096 bytes in size.

The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code (see Section 5.5), a gap, the sector number of the next sector, and so on.

To access data, the operating system must direct the disk through a three-stage process. The first step is to position the head over the proper track. This operation is called a **seek**, and the time to move the head to the desired track is called the *seek time*.

Seek: The process of positioning a read/write head over the proper track on a disk.

Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the **rotational latency** or **rotational delay**. The average latency to the desired information is halfway around the disk. Disks rotate at 5400 RPM to 15,000 RPM. The average rotational latency at 5400 RPM is

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

Rotational latency: Also called **rotational delay**. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

The last component of a disk access, *transfer time*, is the time to transfer a block of bits. The transfer time is a function of the sector size, the rotation speed, and the recording density of a track. Transfer rates in 2020 were between 150 and 250 MB/ sec.

The Basics of Caches

In our library example, the desk acted as a cache—a safe place to store things (books) that we needed to examine. *Cache* was the name chosen to represent the level of the memory

hierarchy between the processor and main memory in the first commercial computer to have this extra level.

Today, although this remains the dominant use of the word *cache*, the term is also used to refer to any storage managed to take advantage of locality of access.

Before the request, the cache contains a collection of recent references X_1, X_2, ..., X_{n-1}, and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought from memory into the cache.

The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the *address* of the word in memory. This cache structure is called **direct mapped**, since each memory location is mapped directly to exactly one location in the cache. The typical mapping between addresses and cache locations for a direct-mapped cache is usually simple. For example, almost all direct-mapped caches use this mapping to find a block:

Direct-mapped cache: A cache structure in which each memory location is mapped to exactly one location in the cache.

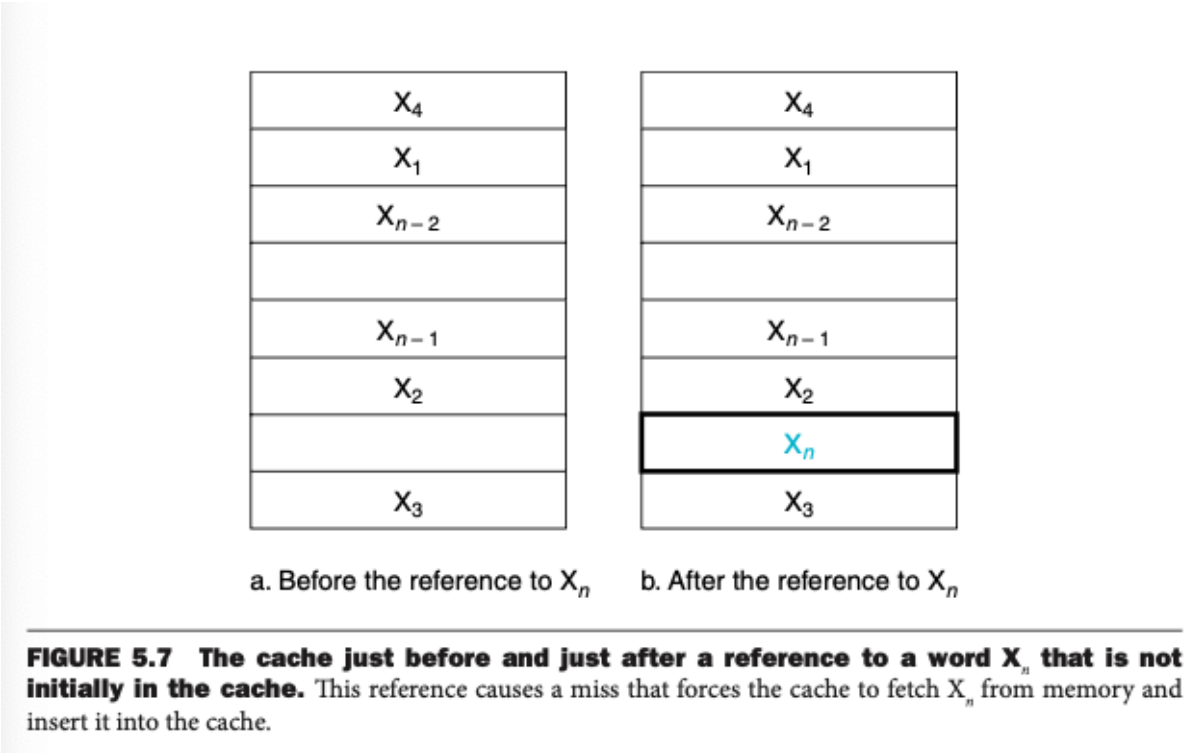


FIGURE 5.7 The cache just before and just after a reference to a word X_n that is not initially in the cache. This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache.

Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word? That is, how do we know whether a requested word is in the cache or not? We answer this question by adding a set of **tags** to the cache. The tags contain the address information required to identify whether a word in the cache corresponds to the requested word. The tag needs just to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache.

Tag: A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

We also need a way to recognize that a cache block does not have valid information. For instance, when a processor starts up, the cache does not have good data, and the tag fields will be meaningless. Even after executing many instructions, some of the cache entries may still be empty, as in Figure 5.7. Thus, we need to know that the tag should be ignored for such entries. The most common method is to add a **valid bit** to indicate whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block

Valid bit: A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

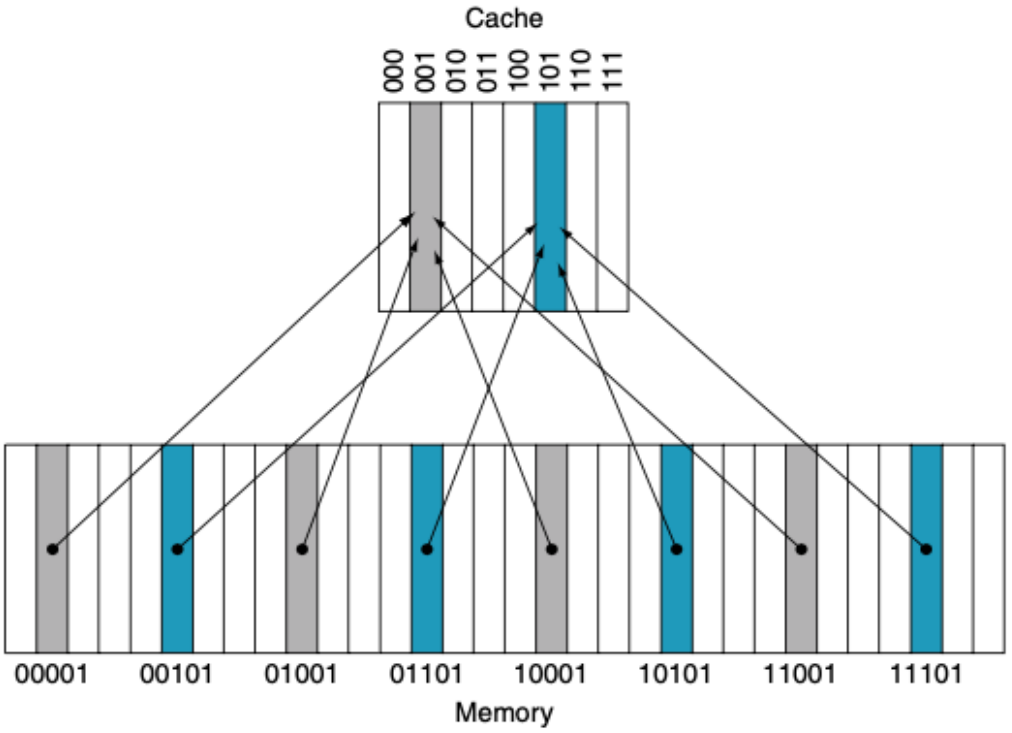


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

Caching is perhaps the most important example of the big idea of **prediction**. It relies on the principle of locality to try to find the desired data in the higher levels of the memory hierarchy, and provides mechanisms to ensure that when the prediction is wrong it finds and uses the proper data from the lower levels of the memory hierarchy. The hit rates of the cache prediction on modern computers are often above 95%

Accessing a Cache

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.9b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (5.9c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (5.9d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (5.9e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (5.9f)	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

Since the cache is empty, several of the first references are misses; the caption of Figure 5.9 describes the actions for each memory reference. On the eighth reference we have conflicting demands for a block. The word at address 18 (10010_{two}) should be brought into cache block 2 (010_{two}). Hence, it must replace the word at address 26 (11010_{two}), which is already in cache block 2 (010_{two}). This behavior allows a cache to take advantage of temporal locality: recently referenced words replace less recently referenced words.

This situation is directly analogous to needing a book from the shelves and having no more space on your desk—some book already on your desk must be returned to the shelves. In a direct-mapped cache, there is only one place to put the newly requested item and hence just one choice of what to replace.

We know where to look in the cache for each possible address: the low-order bits of an address can be used to find the unique cache entry to which the address could map. Figure 5.10 shows how a referenced address is divided into

- A tag field, which is used to compare with the value of the tag field of the cache
- A cache index, which is used to select the block

The index of a cache block, together with the tag contents of that block, uniquely specifies the memory address of the word contained in the cache block.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

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

b. After handling a miss of address (10110_{two})

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

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 5.9 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary for the sequence of addresses on page 402. The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block i with tag field j for this cache is j×8+ i, or equivalently the concatenation of the tag field j and the index i. For example, in cache f above, index 010_{two} has tag 10_{two} and corresponds to address 10010_{two}.

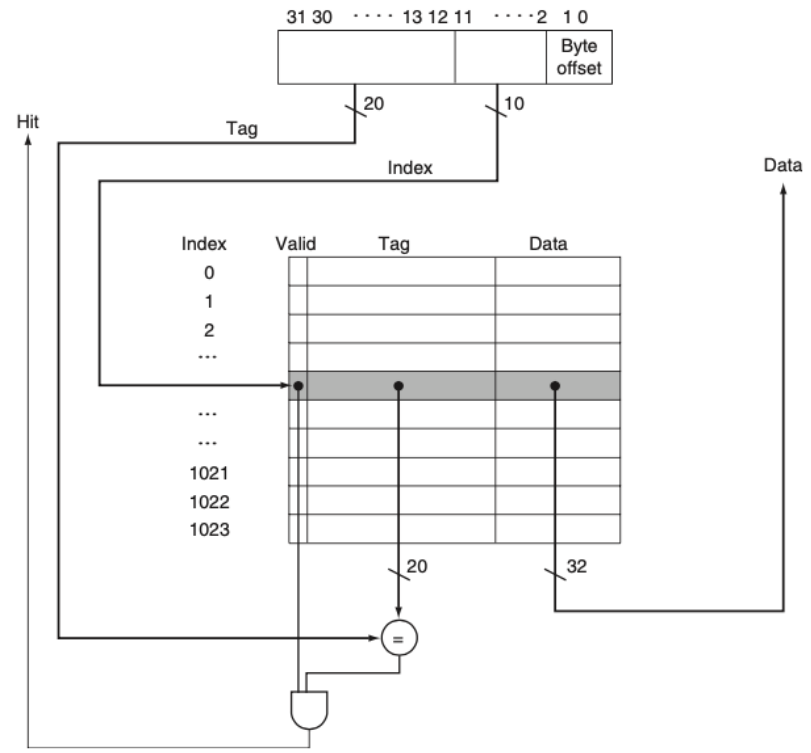


FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KiB. Unless noted otherwise, we assume 32-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving $32 - 10 - 2 = 20$ bits to be compared against the tag. If the tag and upper 52 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

The total number of bits needed for a cache is a function of the cache size and the address size, because the cache includes both the storage for the data and the tags. The size of the block above was one word (4 bytes), but normally it is several. For the following situation:

- 32-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks, so n bits are used for the index
- The block size is 2^m words (2^{m+2} bytes), so m bits are used for the word within the block, and two bits are used for the byte part of the address

The size of the tag field is:

$$32 - (n + m + 2)$$

The total number of bits in a direct-mapped cache is:

$$2^n \cdot (block\ size + tag\ size + valid\ field\ size)$$

Since the block size is 2^m words (2^m bits), and we need 1 bit for the valid field, the number of bits in such a cache is

$$2^n \cdot (2^m \cdot 32 + (32 - n - m - 2) + 1) = 2^n \cdot (2^m \cdot 32 + 21 - n - n)$$

Although this is the actual size in bits, the naming convention is to exclude the size of the tag and valid field and to count only the size of the data. Thus, the cache in Figure 5.10 is called a 4 KiB cache even though it actually contains 1.375 KiB of tags and valid bits plus 4 KiB of data.

Bits in cache Example:

How many total bits are required for a direct-mapped cache with 16 KiB of data and four-word blocks, assuming a 64-bit address?

We know that 16 KiB is 4096 (2^{12}) words. With a block size of four words (2^{2}), there are 1024 (2^{10}) blocks. Each block has 4 × 32 or 128 bits of data plus a tag, which is 64 − 10 − 2 − 2 bits, plus a valid bit. Thus, the complete cache size is

$$2^{10} \cdot (4 \cdot 32 + (54 - 10 - 2 - 2) + 1) = 2^{10} \cdot 179 = 179\ kibibits \times$$

or 22.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.4 times as many as needed just for the storage of the data.

Mapping an Address to a Multiword Cache Block

Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

We saw the formula on page 399. The block is given by

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

where the address of the block is

$$\frac{\text{Byte address}}{\text{Bytes per block}}$$

Notice that this block address is the block containing all addresses between

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block}$$

and

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

which maps to cache block number (75 modulo 64) = 11. In fact, this block maps all addresses between 1200 and 1215.

Larger blocks exploit spatial locality to lower miss rates. As Figure 5.11 shows, increasing the block size usually decreases the miss rate.

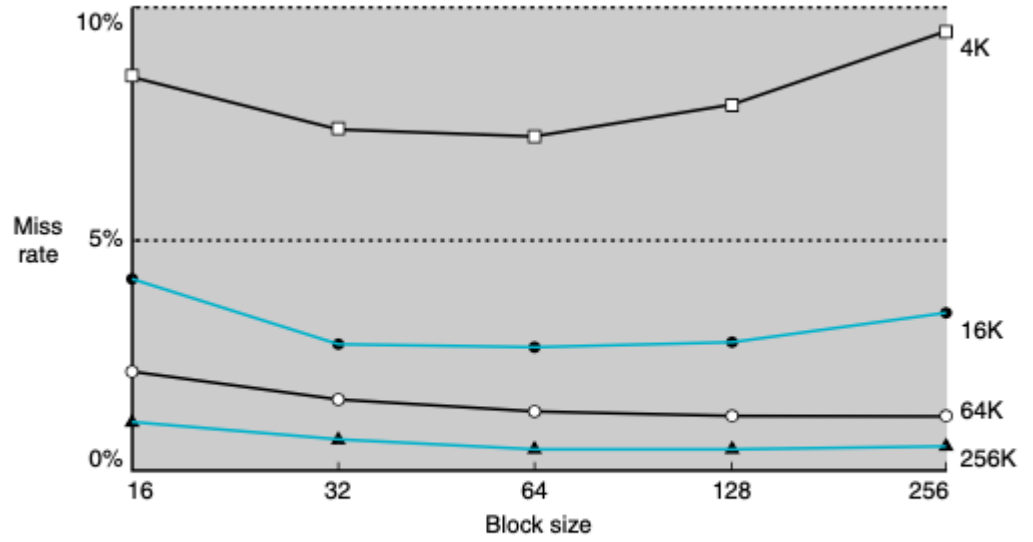


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.)

A more serious problem associated with just increasing the block size is that the cost of a miss rises. The miss penalty is determined by the time required to fetch the block from the next lower level of the hierarchy and load it into the cache. The time to fetch the block has two parts: the latency to the first word and the transfer time for the rest of the block.

Handling cache misses

Exercises:

5.1 In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously. Assume each word is a 64-bit integer.

$for\ (I = 0;\ I < 8;\ I\ ++)$

$for\ (J = 0;\ j < 8000;\ J\ ++)$

$A[I][J] = B[I][0] + A[j][I]$

5.1.1 How many 64-bit integers can be stored in a 16-byte cache block?

Answer:

Here are the steps to determine how many 64-bit integers can be stored in a 16-byte cache block:

1. Determine the size of the data type you want to store in the cache block. In this case, we want to store 64-bit integers, which take up 8 bytes of memory each.
2. Divide the size of the cache block by the size of the data type. In this case, the cache block is 16 bytes, so we divide 16 by 8, which gives us 16 / 8 = 2.
3. The result of this division tells us how many 64-bit integers can fit in the 16-byte cache block. In this case, 2 64-bit integers can fit in the 16-byte cache block.

So In short, 16 bytes cache block can store 2x64-bit integers

5.1.2 Which variable references exhibit temporal locality?

Answer:

The variables I and J exhibit temporal locality, because here the program repeatedly iterates through the matrix in a nested loop. This creates a high degree of temporal locality, as each element of the array is accessed multiple times in succession. because we access them often once in the for loop and then again in the end. There is also B[I][0] because we acces array i and 0

5.1.3 Which variable references exhibit spatial locality?

Answer:

The variables A[I] and A[J] exhibits spatial locality. In this example, the program iterates through the matrix in a nested loop, first by row and then by column. This creates a high degree of spatial locality, as each element of the array is accessed immediately after its

neighbors in the same row. The memory address A[|] and A[J] are visited after each other and are neighbors. because we acces i and then j every time the two values lays close to one another - because of the column major order

Locality is affected by both the reference order and data layout. The same computation can also be written below in Matlab, which differs from C in that it stores matrix elements within the same column contiguously in memory.

$for\ I = 1{:}\ 8$

$for\ J = 1{:}\ 8000$

$A(I,J) = B(I,0) + A(J,I);$

end

end

5.1.4 Which variable references exhibit temporal locality?

Answer:

The variables I and J exhibits temporal locality, but also B(I,0)

5.1.5 Which variable references exhibit spatial locality?

Answer:

The variables A(I,J), B(I,0) and A(J,I) exhibit spatial locality, because the memory are neighbors.

5.1.6 How many 16-byte cache blocks are needed to store all 64-bit matrix elements being referenced using Matlab's matrix storage? How many using C's matrix storage? (Assume each row contains more than one element.)

In Matlab, matrices are stored in column-major order, which means that elements in the same column are stored in contiguous memory locations. In contrast, C uses row-major order, which means that elements in the same row are stored in contiguous memory locations.

To determine the number of cache blocks needed to store all elements of a matrix, we need to know the size of the cache block (also called the cache line size) and the number of elements in the matrix. Let's assume that the cache block size is 16 bytes, and we have a 64-bit matrix with m rows and n columns.

For Matlab:

In this case, each element of the matrix takes up 8 bytes (since it is 64-bit), so one cache block can hold 2 matrix elements. To store all the elements of the matrix, we will need (m * n) / 2 cache blocks.

For C:

In this case, we have n elements in one row, and m rows so we have n*m elements and same as before 8 bytes per element. so we have (m * n * 8) / 16 = (m * n) / 2 cache blocks

So both language uses same approach as they have elements stored in different order but since the number of elements are same it doesn't affect the number of cache blocks.

5.2 Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 64-bit memory address references, given as word addresses.

0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5, 0x2c, 0xba, 0xfd

5.2.1 For each of these references, identify the binary word address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list whether each reference is a hit or a miss, assuming the cache is initially empty.

To convert the memory addresses from hexadecimal to binary, we can use the following method:

- Convert each hexadecimal digit to its corresponding four-bit binary value.
- Concatenate the binary values to form the full binary memory address.

For example, the hexadecimal address "0x03" would be converted to binary as follows:

- 0x03 = 0011 (in binary)

Assuming a direct-mapped cache with 16 one-word blocks and a block size of 1 word, we can calculate the tag, index, and offset of each memory address as follows:

- The cache has 2^4 = 16 blocks, so the index field is 4 bits wide.
- The offset is always 0, since the block size is 1 word.
- The tag field is the remaining bits.

Therefore, each address will be 64 bits, 4 bits index, and remaining will be Tag.

Assuming the cache is initially empty, each of the memory references will result in a cache miss.

And Below are the listing of each memory references with its binary value, tag value, index value, and cache hit/miss status:

Here is an example of how to identify the tag, index, and hit/miss for a memory reference of 0x03:

- Binary representation of 0x03: 00000000000000000000000000000011
- The last 4 bits (11) represent the index, which maps to cache block 3.
- The first 60 bits (000000000000000000000000000000000000000000000000) represent the tag.
- Since the cache is initially empty, this is a cache miss.

Here are the results for all memory references provided:

Memory Address	Binary Word Address	Tag	Index	Hit/Miss
0x03	000000000000000000000000000011	000000000000000000000000000000	11 (3)	Miss
0xb4	10110100	1011	0100 (4)	Miss
0x2b	00101011	001010	1011 (11)	Miss

0x02	0000000000000010	00000000000000000000000000000000	0010 (2)	Miss
0xbf	10111111	1011	1111 (15)	Miss
0x58	01011000	0101	1000 (8)	Miss
0xbe	10111110	1011	1110 (14)	Miss
0x0e	00001110	0000	1110 (14)	Miss
0xb5	10110101	1011	0101 (5)	Miss
0x2c	00101100	0010	1100 (12)	Miss
0xba	10111010	1011	1010 (10)	Miss
0xfd	11111101	1111	1101 (13)	Miss

5.2.2 For each of these references, identify the binary word address, the tag, the index, and the offset given a direct-mapped cache with two-word blocks and a total size of eight blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

Given a direct-mapped cache with two-word blocks and a total size of eight blocks, we can calculate the tag, index, offset of each memory address as follows:

- The cache has 2^3 = 8 blocks, so the index field is 3 bits wide.
- The block size is 2 words, so the offset field is 1 bits wide.
- The tag field is the remaining bits.

Therefore, each address will be 64 bits, 3 bits index, 1 bit offset and remaining will be Tag.

Assuming the cache is initially empty, each of the memory references will result in a cache miss.

And Below are the listing of each memory references with its binary value, tag value, index value, offset value, and cache hit/miss status:

Memory Address (Hex)	**Binary Address**	**Tag (binary)**	**Index (binary)**	**Offset (binary)**	**Hit/Miss**
0x03	0000000000000011	00000000000000	000	1	Miss
0xb4	10110100	1011010	000	0	Miss

0x2b	00101011	0010101	011	1	Miss
0x02	00000010	000000	000	0	Miss
0xbf	10111111	1011111	111	1	Miss
0x58	01011000	01011	100	0	Miss
0xbe	10111110	1011111	110	0	Miss
0x0e	00001110	000011	110	0	Miss
0xb5	10110101	1011010	101	1	Miss
0x2c	00101100	001011	100	0	Miss
0xba	10111010	101110	010	0	Miss

0xfd	11111101	111111	101	1	Miss

5.2.3 You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of eight words of data:

- **C1 has 1-word blocks,**
- **C2 has 2-word blocks, and**
- **C3 has 4-word blocks.**

In order to optimize the cache design for the given references, we can analyze the cache miss rate for each of the three possible cache designs: C1, C2, and C3.

- A direct-mapped cache with 1-word blocks (C1) will have the highest cache miss rate among the three designs. This is because there is a higher probability of cache conflicts, as each block can only hold one word, and there are only 8 blocks in total. As a result, multiple memory references may map to the same cache block, leading to cache evictions and miss.
- A direct-mapped cache with 2-word blocks (C2) will have a lower cache miss rate compared to C1. By doubling the block size, we reduce the number of cache conflicts and increase the chances of a cache hit. However, still the miss rate may be higher than the last design
- A direct-mapped cache with 4-word blocks (C3) will have the lowest cache miss rate among the three designs. With this design, we have only two cache blocks and the likelihood of multiple memory references mapping to the same block is reduced, resulting in a lower cache miss rate. However, this design will also use up more memory resources.

In conclusion, the optimal cache design for the given references is C3 with 4-word blocks as it will have the lowest cache miss rate among the three designs.

It is worth noting that this estimation and conclusion is based on the assumption that the memory references follow a certain pattern or temporal locality and the above analysis for a single round of access. Also, other factors like size of the cache and the number of memory references should also be taken into consideration for the optimal cache design for the specific case.

5.5 For a direct-mapped cache design with a 32-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
63–10	9–5	4–0

5.5.1 What is the cache block size (in words)?

The cache block size is typically determined by the offset field in the memory address. The offset field indicates the byte within a cache block at which the memory access is directed. The size of the offset field, in bits, is equal to the log2 of the cache block size in bytes. So, if the offset field is 6 bits, the cache block size is 2^6 = 64 bytes.

$2^4 = 16\ bytes$

Given the tag, index, and offset fields of a memory address, the cache block size can be determined by finding the log2 of the number of bytes per block using the offset value. This is the block size that is used to determine the block the data should be stored.

It is important to note that, in practice, the block size is often a parameter that is set when configuring the cache and not something that is directly inferred from the address fields.

5.5.2 How many blocks does the cache have?

To find the number of blocks in a cache, you can use the following formula:

Number of blocks = Total cache size / Block size

$32/16 = 2$ bytes

Where the total cache size is usually measured in bytes, and the block size is also measured in bytes.

Once you have the number of blocks in the cache, you can use the index field of the memory address to determine which block a memory access is directed to. The number of bits in the index field is equal to log2 of the number of blocks in the cache. Given the tag, index, offset fields, and the block size, it is possible to calculate the number of blocks in the cache.

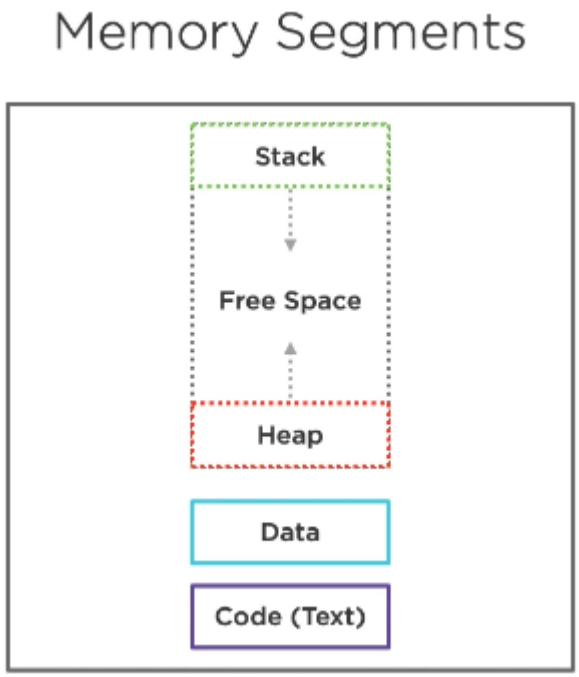
It is important to note that, in practice, the number of blocks in a cache is often a parameter that is set when configuring the cache and not something that is directly inferred from the address fields, the block size is the parameter that often help to determine the number of blocks in a cache, usually the cache block number is a power of 2 and this help to determine the number of blocks directly from the block size.

Dynamic memory

Der er fire memory segments i C.

Det første er kode (text) delen.
Så er der data delen som “stores” alle ens variable
Så kommer der stack segmentet som er stacken.
i stack segmenten ligger det vi kalder heap, som er der vores memory opbevares.

Når antallet af ting i stacken vokser, fylder den pladsen ud ned mod heap og når heap fyldes op med ting fylder den op mod stacken.



Dynamic vs static memory allocation.

Static memory

- Opbevares i stacken og bliver styret af compileren
- Hukommelsen bliver styret af compileren, den finder selv ud af hvor meget “plads” der skal bruges for at alt kan være der.
- Man kan ikke frigøre mere hukommelse

Dynamic memory

- Opbevares i heapen og bliver styret af udvikleren
- Hukommelse allocated at runtime

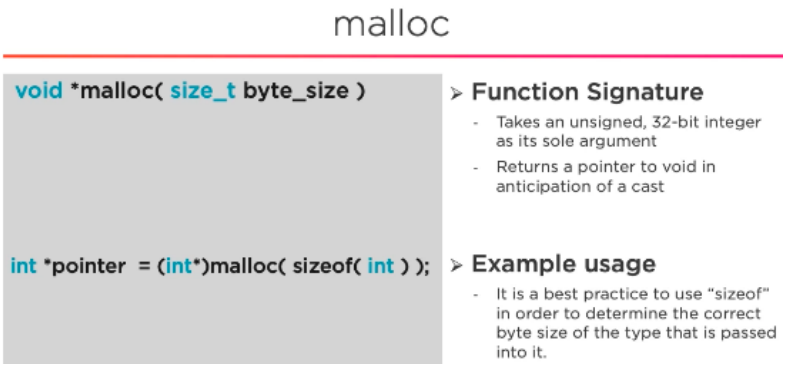
- Hukommelse kan blive frigjort af udvikleren.

C Standard Library Core Memory Functions

- <stdlib.h>

malloc()

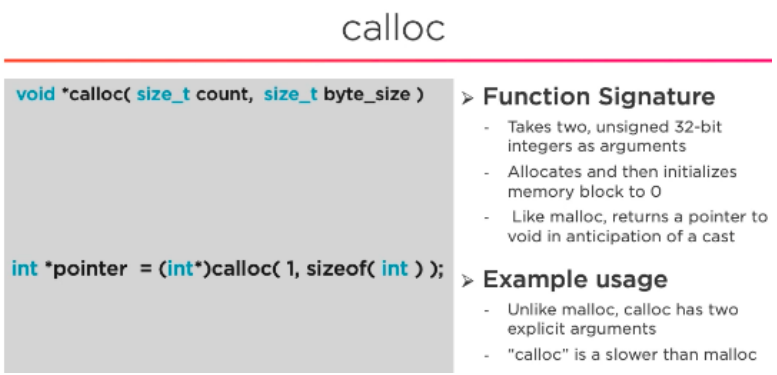
- As arguments takes size and byte size
- returns a pointer to void anticipation of a cast



- Man kan bruge exemplet til at finde ud af den rette byte size

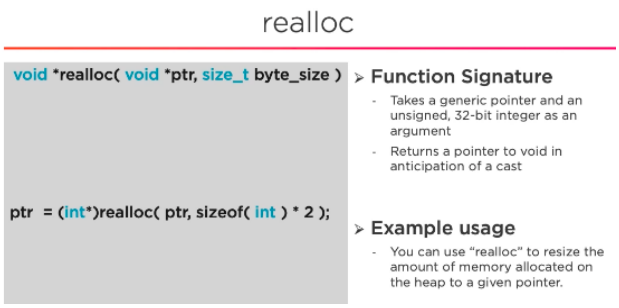
calloc()

- Reminds of malloc
- As arguments takes size count (som er den type du gerne vil allocate for), size_t and byte_size.



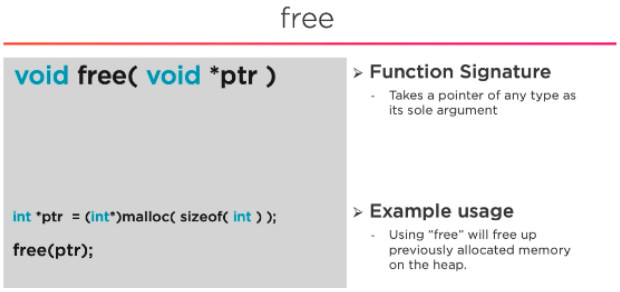
realloc()

- As arguments void *ptr (void pointer, advarsel det skal være en pointer som før har været allocated til noget af malloc eller calloc), size_t, byte_size
- returns a pointer to void anticipation of a cast



free()

- As arguments void *ptr
- Kan undgå memory leaks fordi den frigøre mere plads i heapen.



Operating systems

An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is the most important type of system software in a computer system. It manages the computer's memory, processes, and all of its software and hardware. It also allows you to communicate with the computer without knowing how to speak the computer's language.

Processes

The **process** is the major OS abstraction of a running program. At any point in time, the process can be described by its state: the con- tents of memory in its **address space**, the contents of CPU registers (including the **program counter** and **stack pointer**, among others), and information about I/O (such as open files which can be read or written).

The **processAPI** consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.

Though we defer discussion of a real process API until a subsequent chapter, here we first give some idea of what must be included in any interface of an operating system. These APIs, in some form, are available on any modern operating system.

Create: An operating system must include some method to cre- ate new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated.

Destroy: As there is an interface for process creation, systems also provide an interface to destroy processes forcefully. Of course, many processes will run and just exit by themselves when complete; when they don’t, however, the user may wish to kill them, and thus an in- terface to halt a runaway process is quite useful.

Wait: Sometimes it is useful to wait for a process to stop running; thus some kind of waiting interface is often provided.

Miscellaneous Control: Other than killing or waiting for a process, there are sometimes other controls that are possible. For example, most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (con- tinue it running).

Status: There are usually interfaces to get some status information about a process as well, such as how long it has run for, or what state it is in.

Process states

Processes exist in one of many different **process states**, including running, ready to run, and blocked. Different events (e.g., getting scheduled or descheduled, or waiting for an I/O to complete) transition a process from one of these states to the other.

Running: In the running state, a process is running on a processor. This means it is executing instructions.

Ready: In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.

Blocked: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

A **process list** contains information about all processes in the sys- tem. Each entry is found in what is sometimes called a **process control block** (**PCB**), which is really just a structure that contains information about a specific process.

The Kernel

Technically operating systems encompass lots of parts, shell, GUI, C library etc.

Kernel is always running, it holds the most basic and important code of the computer. It services the hardware and manages processes.

The kernel takes over for the CPU when its interrupted for example if we press a key, or tries to access an invalid memory access. Then when it has handled that it jumps back such that the CPU runs the code again.

Supervisor mode: Also called **kernel mode**. A mode indicating that a running process is an operating system process.

System call: A special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process.

Context switching:

Each processes things that they are “alone” in the CPU, but it's not.

Only one process runs at a time, but we regularly switch between available processes.

We take the process and save its entire state, and then we can run some other process based on that process's earlier save.

System calls (RISC-V)

A request by a process that the kernel carries out some operation on its behalf

The *ecall* (environment call) instruction transfers control to the kernel.

- The kernel then inspects registers (mostly a0-a7) to see that it has been asked to do.

PrintInt - prints to the console what's written in a0
ReadInt - reads int from console writes whats in a0

System calls (C)

fork()

- is used to create a new process.

When we run *fork()* the process prints out the code in the program. But it also gives us its process identifier (also called PID). The PID is used to name a process if you would want to do something with the process, feks. stop it from running.

When the “new process” is created, then the process created is almost an exact copy of the running program. For the Operating System it would look like we are running two exactly identical processes and both are about to return the *fork()* system call.

The first/original process is called the parent and the “new process” is called the child.

The “new process” does not start to run at the main() function in the beginning of the program but this is not the case, rather it just comes into life as if it had called *fork()* itself.

The Child “new process” is not an exact copy of the parents. It has its own copy of the address space (its own memory), its own register, own PC and so on, but the value it returns to the caller of *fork()* is different.

While the parent receives the PID of the newly-created child, the child receives a return code of zero. This differentiation is useful, because it is simpler then to write the code that handles the two different cases (as above).

Because we have two active processes running at once, it can sometimes happen that if we have a single CPU (for simplicity) then either the child or parent might run at one time, and which one is shown in the terminal can differentiate.

Example fork() and wait()

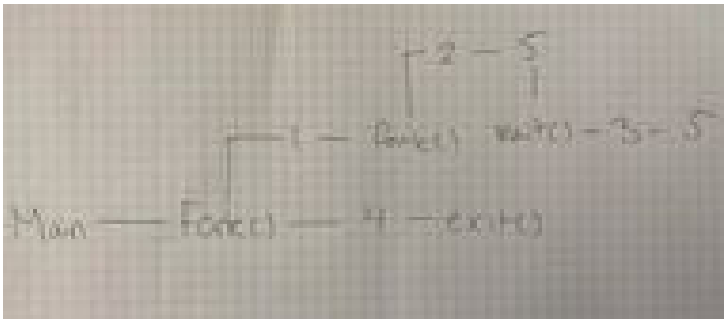
Question 2.1.3:

Consider the C program below. For space reasons, we are not checking error return codes, so assume that all functions return normally.

```
int main () {
  if (fork() == 0) {
    printf("1");
    if (fork() == 0) {
      printf("2");
    } else {
      pid_t pid; int status;
      if ((pid = waitpid(pid, NULL, 0)) > 0) {
          printf("3");
      }
    }
  } else {
    printf("4");
    exit(0);
  }
  printf("5");
}
```

Which of the following strings is a possible output of the program? Place any number of marks.

- [x] **a)** 125354
- [x] **b)** 412535
- [x] **c)** 124535
- [ ] **d)** 512345
- [ ] **e)** 412355



wait()

- makes it possible for the parent to wait for the child process to finish what it has been doing.

This call is more complete if we call *waitpid()*.

With this call we can ensure that the child is always printed first. It can still happen that the parent run first, but it will be stopped and paused by the *wait()* call until the child is done, then wait() returns, and then the parent prints its message.

exec()

- This system call is useful when you want to run a program that is different from the calling program.

When we run *fork()* it will always run a copy of the running program, but if you dont want to do that you can use *exec()*.

When running *execvp()* the output will tells how many lines, word and bytes there are found in the file.

What it does: given the name of an executable, and some arguments, it **loads** code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized.

The Operation system simply runs the new process passing in the arguments argv of that process. It does not create a new process, it transforms the currently running program into a different running program.

After the *exec()* in the child, it is almost as if the old/current process never ran; a successful call to *exec()* never returns.

Virtual memory - hardware perspective COD 5.7

Virtual memory: A technique that uses main memory as a “cache” for secondary storage.

In earlier sections, we saw how caches provided fast access to recently-used portions of a program’s code and data. Similarly, the main memory can act as a “cache” for the secondary storage, traditionally implemented with magnetic disks. This technique is called **virtual memory**.

Historically, there were two major motivations for virtual memory: to allow efficient and safe sharing of memory among several programs, such as for the memory needed by multiple virtual machines for Cloud computing, and to remove the programming burdens of a small, limited amount of main memory. It's still the same reason today. 🙂

If we want to allow multiple virtual machines to work together, we need to protect them from each other. This means that each virtual machine can only read and write the portion of main memory that is assigned to it.

Virtual memory implements the translation of a program’s address space to **physical addresses**. This translation process enforces **protection** of a program’s address space from other virtual machines.

physical address: An address in main memory.

protection: Aset of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or unintentionally, with one another by reading or writing each other’s data. These mechanisms also isolate the operating system from a user process.

There is two reasons to use virtual memory

1. To allow efficient and sage sharing of memory among several programs, such as for the memories needed by multiple virtual machines for cloud computing.
2. To allow a single-user program to exceed the size of primary memory.

1. For doing this it's important that we protect the two virtual machines from each other making sure they only use the main memory that has been assigned to them.

Because we can’t know which virtual machine will share the memory with other virtual machines when we compile them, we want to compile each program into its own address. In fact, the virtual machines sharing the memory change dynamically while they are running. Because of this dynamic interaction, we would like to compile each program into its own *address space*—a separate range of memory locations accessible only to this program.

2. Virtual memory, automatically manages the two levels of the memory hierarchy represented by main memory (sometimes called physical memory) and secondary storage.

Page fault: An event that occurs when an accessed page is not present in main memory. If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. The OS then must find the page in the next level of the hierarchy and decide where to place the requested page in the main memory.

Swap space: The space on the disk reserved for the full virtual memory space of a process.

Virtual address: An address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.

Address translation: Also called **address mapping**. The process by which a virtual address is mapped to an address used to access memory.

Page table: The table containing the virtual to physical address translations in a virtual memory system. The table, which is stored in memory, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.

Segmentation: A variable-size address mapping scheme in which an address consists of two parts: a segment number, which is mapped to a physical address, and a segment offset.

Exception enable: Also called interrupt enable. A signal or action that controls whether the process responds to an exception or not; necessary for preventing the occurrence of exceptions during intervals before the processor has safely saved the state needed to restart.

Restartable instruction: An instruction that can resume execution after an exception is resolved without the exception's affecting the result of the instruction.

TLB - Making address translation fast

Accordingly, modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a **translation-lookaside buffer (TLB)**, although it would be more accurate to call it a translation cache.

Translation-lookaside buffer (TLB): A cache that keeps track of recently used address mappings to try to avoid an access to the page

The TLB corresponds to that little piece of paper we typically use to record the location of a set of books we look up in the card catalog; rather than continually searching the entire catalog, we record the location of several books and use the scrap of paper as a cache of Library of Congress call numbers.

Some typical values for a TLB might be

- TLB size: 16–512 entries
- Block size: 1–2 page table entries (typically 4–8 bytes each)

- Hit time: 0.5–1 clock cycle
- Miss penalty: 10–100 clock cycles
- Miss rate: 0.01%–1%

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

FIGURE 5.32 The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, page table hit, cache miss) but never detected.

A Common Framework for Memory Hierarchy COD 5.8

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large block could increase miss rate

FIGURE 5.36 Memory hierarchy design challenges.

Where Can a Block Be Placed?

One place (direct mapped), a few places (set associative), or any place (fully associative).

We have seen that block placement in the upper level of the hierarchy can use a range of schemes, from direct mapped to set associative to fully associative. As mentioned above, this entire range of schemes can be thought of as variations on a set-associative scheme where the number of sets and the number of blocks per set varies:

Scheme name	Number of sets	Blocks per set
Direct mapped	Number of blocks in cache	1
Set associative	$\frac{\text{Number of blocks in the cache}}{\text{Associativity}}$	Associativity (typically 2–16)
Fully associative	1	Number of blocks in the cache

Feature	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Total size in blocks	250–2000	2500–25,000	16,000–250,000	40–1024
Total size in kilobytes	16–64	125–2000	1,000,000–1,000,000,000	0.25–16
Block size in bytes	16–64	64–128	4000–64,000	4–32
Miss penalty in clocks	10–25	100–1000	10,000,000–100,000,000	10–1000
Miss rates (global for L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

FIGURE 5.33 The key quantitative design parameters that characterize the major elements of memory hierarchy in a computer. These are typical values for these levels as of 2020. Although the range of values is wide, this is partially because many of the values that have shifted over time are related; for example, as caches become larger to overcome larger miss penalties, block sizes also grow. While not shown, server microprocessors today also have L3 caches, which can be 4 to 50 MiB and contain many more blocks than L2 caches. L3 caches lower the L2 miss penalty to 30 to 40 clock cycles.

The advantage of increasing the degree of associativity is that it usually decreases the miss rate. The improvement in miss rate comes from reducing misses that compete for the same location.

How Is a Block Found?

There are four methods: indexing (as in a direct-mapped cache), limited search (as in a set-associative cache), full search (as in a fully associative cache), and a separate lookup table (as in a page table).

The choice of how we locate a block depends on the block placement scheme, since that dictates the number of possible locations. We can summarize the schemes as follows:

Associativity	Location method	Comparisons required
Direct mapped	Index	1
Set associative	Index the set, search among elements	Degree of associativity
Full	Search all cache entries	Size of the cache
	Separate lookup table	0

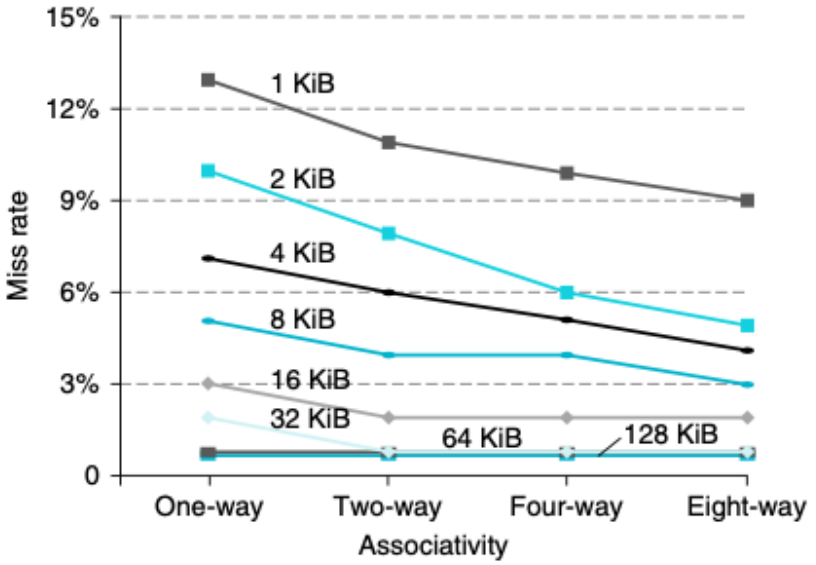


FIGURE 5.34 The data cache miss rates for each of eight cache sizes improve as the associativity increases. While the benefit of going from one-way (direct mapped) to two-way set associative is significant, the benefits of further associativity are smaller (e.g., 1–10% improvement going from two-way to four-way versus 20–30% improvement going from one-way to two-way). There is even less improvement in going from four-way to eight-way set associative, which, in turn, comes very close to the miss rates of a fully associative cache. Smaller caches obtain a significantly larger absolute benefit from associativity because the base miss rate of a small cache is larger. Figure 5.16 explains how these data were collected.

The choice among direct-mapped, set-associative, or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware.

In virtual memory systems, a separate mapping table—the page table—is kept to index the memory. In addition to the storage needed for the table, using an index table requires an extra memory access. The choice of full associativity for page placement and the extra table is motivated by these facts:

1. Full associativity is beneficial, since misses are very expensive.
2. Full associativity allows software to use sophisticated replacement schemes that are designed to reduce the miss rate.
3. The full map can be easily indexed with no extra hardware and no searching required.

Which Block Should Be Replaced on a Cache Miss?

Typically, either the least recently used or a random block.

When a miss occurs in an associative cache, we must decide which block to replace.

In a fully associative cache, all blocks are candidates for replacement. If the cache is set associative, we must choose among the blocks in the set. Of course, replacement is easy in a direct-mapped cache because there is only one candidate.

There are the two primary strategies for replacement in set-associative or fully associative caches:

Random: Candidate blocks are randomly selected, possibly using some hardware assistance.

Least recently used (LRU): The block replaced is the one that has been unused for the longest time.

In practice, LRU is too costly to implement for hierarchies with more than a small degree of associativity (two to four, typically), since tracking the usage information is expensive.

What Happens on a Write?

Each level in the hierarchy can use either write-through or write-back.

A key characteristic of any memory hierarchy is how it deals with writes. We have already seen the two basic options:

Write-through: The information is written to both the block in the cache and the block in the lower level of the memory hierarchy (main memory for a cache). The caches in Section 5.3 used this scheme.

Write-back: The information is written just to the block in the cache. The modified block is written to the lower level of the hierarchy only when it is replaced. Virtual memory systems always use write-back, for the reasons discussed in Section 5.7.

Both write-back and write-through have their advantages. The key advantages of write-back are the following:

- Individual words can be written by the processor at the rate that the cache, rather than the memory, can accept them.
- Multiple writes within a block require only one write to the lower level in the hierarchy.
- When blocks are written back, the system can make effective use of a high-bandwidth transfer, since the entire block is written.

Write-through has these advantages:

- Misses are simpler and cheaper because they never require a block to be written back to the lower level.
- Write-through is easier to implement than write-back, although to be realistic, a write-through cache will still need to use a write buffer.

The Three Cs: An Intuitive Model for Understanding the Behavior of Memory Hierarchies

Three Cs model: A cache model in which all cache misses are classified into one of three categories: compulsory misses, capacity misses, and conflict misses.

Compulsory miss: Also called **cold-start miss**. A cache miss caused by the first access to a block that has never been in the cache.

Capacity miss: A cache miss that occurs because the cache, even with full associativity, cannot contain all the blocks needed to satisfy the request.

Conflict miss: Also called **collision miss**. A cache miss that occurs in a set-associative or direct- mapped cache when multiple blocks compete for the same set and that are eliminated in a fully associative cache of the same size.

Heap

External fragmentation: External fragmentation occurs in a heap when there are small gaps of free memory between allocated blocks, causing the memory to be less efficiently used.

An example of external fragmentation on a heap would be as follows:

1. A program requests memory for a block of size 100 bytes, the heap allocates this block and marks the next 100 bytes as used.
2. The program then frees this block of memory, leaving 100 bytes of free memory in the heap.
3. The program then requests memory for a block of size 50 bytes, but since there is not a single free block of memory that is 50 bytes or larger, the heap must allocate a new block of memory, leaving the previous 100 bytes of free memory unused.
4. This process continues, with the program allocating and freeing blocks of memory, resulting in multiple small fragments of free memory scattered throughout the heap.

As a result, there's now a lot of free memory that is unusable because it is not contiguous, and it can't be used to satisfy a new memory request.

For example, Block 1 is allocated and then freed, leaving a gap of unused memory. Then Block 2 is allocated and then freed, leaving another gap of unused memory. Finally, Block 3 is allocated. As a result, there are two fragments of unused memory that can't be used to satisfy a new memory request.

Internal fragmentation: Internal fragmentation occurs when the memory allocated for a block is larger than the amount of memory actually needed for the data stored in that block, again resulting in less efficient use of memory. (if the header and footer is next to each other and we need to use more space)

Exercises:

5.16 As described in Section 5.7, virtual memory uses a page table to track the mapping of virtual addresses to physical addresses. This exercise shows how this table must be updated as addresses are accessed. The following data constitute a stream of virtual byte addresses as seen on a system. Assume 4 KiB pages, a four- entry fully associative TLB, and true LRU replacement. If pages must be brought in from disk, increment the next largest page number.

Decimal	4669	2227	13916	34587	48870	12608	49225
hex	0x123d	0x08b3	0x365c	0x871b	0xbee6	0x3140	0xc049

TLB

Valid	Tag	Physical Page Number	Time Since Last Access
1	0xb	12	4
1	0x7	4	1
1	0x3	6	3
0	0x4	9	7

Page table

Index	Valid	Physical Page or in Disk
0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
a	1	3
b	1	12

5.16.1 For each access shown above, list

- **whether the access is a hit or miss in the TLB,**
- **whether the access is a hit or miss in the page table,**
- **whether the access is a page fault,**
- **the updated state of the TLB.**

		Tag	Index	Hit/miss

Eksamenopgaver

Realloc i heap:
Først finder jeg det der skal realloc (markeret med gul), i det her tilfælde kan headeren kun være den nederste blok(i 0x12bffc) da den adresse med gul skal være i mellem header og footer. Footteren findes så ved den næste adresse som indeholder samme værdi som headeren (grøn). Så laver vi de grønne værdier om til binær 00100011, så aflæser vi blokken er denne blok allokeret og er den forrige blok allokeret. Svaret er ja til begge ting da der er 1 taller på de to sidste bits i header/footers binære tal. Nu skal vi finde ud af hvad størrelsen af denne blok er: det gøres ved at sætte de sidste 3 bits til 0. = 00100 000 dette svarer til 32 bit. Blokken fylder altså 32 bits. Der er 4 bytes pr række (32 bit = 4 bytes). Blokken fylder altså 32/4 = 8 rækker. (det passer med afstanden fra header til fotter :).
Nu skal vi finde den nye værdi til headeren. Først ser vi at realloc(adres, 8) har en payload (det antal blocks i mellem header og footer) på 8 blocks. Størrelsen på den allokerede blok bliver 8 (payload) + 4 bytes (for header fordi der pr række er 4 bytes) + 4 bytes (for footer) = 16 bytes

så skal vi tjekke om 16 bytes går op i 8. da der i opgaveteksten står (has a size that is a multiple of eight bytes). For at finde ud af hvor langt der er i mellem den nye header og footer siger vi nu 16/4 bytes (pr række) = 4 rækker.
 Så skriver vi den nye størrelse i bit (16): 00010 000
 og indsætter værdierne på de tre sidste bits. Vi ved fra før og funktionskaldet at forrige blok er allokeret og denne blok er allokeret så: 00010011, så der skal 1 taller på de to første pladser. Dette laves om til hex som er 0x13 det sættes ind som den nye headers værdi (samme sted som gamle header) også tæller vi 4 rækker op og skirver samme værdi i footeren. Så skal vi finde ud af hvad der skal stå i de 4 bloks oven over.
Vi ved at den over footeren er fri nu, da den er allokeret (pga 1 tallet forrige bin).
De to øverste finder vi ud af om er frie ved at lave 0x12 om til binær. = 0010010, da de har 0 som sidste bit ved vi at der er frie, og vi kan derfor slå denne (adresse 0x12c028) og forrige blok (adresse 12c00c) sammen (give dem samme værdi).

Hvad er størrelsen af blokken:
8 rækker (det er der i mellem dem) * 4 bytes (pr række) = 32 bytes i alt.
Så vi skriver 32 om til bit (og de sidste 3 til 0): 00100 000, hvad skal der stå på de tre sidste 0. Vi ved at forrige blok er allokeret (fra tidligere) og at denne blok er fri. så 00100010 dette i hex er 0x22, så det skriver vi ind på de to blokke. De to sidste pladser vi mangler der skal der bare stå det samme som før.
Nu skal vi free adressen 0x12c000, det gør vi ved at tage den ny header og fotters værdi 0x13 og laver den om til binær. 0010011, vi kan se at denne og forrige blok er allokeret, og at det giver 16 bytes i decimal (4 rækker).
Nu finder vi den nye værdi: næste blok er fri vi immediate coalescing
(vi freer den), forrige vol er stadig allokeret. Den nye størrelse er nu 12 rækker. det giver os 12 (rækker) * 4 bytes = 48 bit, den nye værdi i binær er 48 = 00110 000 (med de 3 nuller 0). Da vi freer og forrige blok er allokeret. Hedder det nye binære tal 00110010, hex = 0x32. Det sætter vi ind i den nye header (samme sted) og den nye footer (12 rækker op). Alt mellem header og footer skal bare beholde samme værdi.

0010011

Reeksamen 21/22
maskinarkitektur
question 4.1.1
Led efter stedder som hoppper op i koden igen til L1, eller L3 fx (bne, bltz). Pas på stedet hvor at den bare tester om et loop skal kører fx bge. Her er det ummidelbart i L1 og L3

question 4.1.2
Pointers er der hvor der står lw og sw, det der står i parenteserne er fx a4 pointeren.
Hvis noget så bliver rykket ind i a4 et andet sted fx
addi a4, a4, 1
slli a4, a4, 0x2
add a4, a1, a4 // a4= a1 + 4 = det vil sige at når a1 bliver gemt i a4, så er a1 nok også
// en pointer.
Når man ligger et tal til et sted, altså et offset (bites flytning) så er det nok en pointer.

question 4.1.3
Man tjekker det ved at starte i toppen a koden og kører ned igennem og se hvilke registers der ikke er blevet sat.
li a5, 1
bge a5, a0, L4 // a0 er ikke sat, så det er en argumenter
addi a6, a1, 4 // a1 er ikke sat, så det er en argumenter
slli a0, a0, 0x2

J L2 // fx så springer den til L2 og der bliver a4 sat inden at vi bruger den i L1

Husk at kigge efter jumps, vi kører ikke slavisk igennem, vi følger jumpsene fordi registeret kan blive sat et andet sted!!

question 4.1.4
Find alle slags sorts og hav dem i c og risc V

question 4.1.5

Skriv alt hvad du har noteret om koden fx a0 er offset fordi det er blevet bitshiftet

Pipeline and instruction level parallelism
maskinarkitektur

Hvad er IPC? (antal clock cycles) den er antal a wb divideret med antal cyclusser)

Cache and locality

Concurrency i COD 6.1 - 6.2

The goal is to create more powerful computers simply by connecting many existing smaller ones. Then the more processors a person could get, then they get better performance.

Replacing large inefficient processors with many smaller, efficient processors can deliver better performance per Joule both in the large and in the small, if software can efficiently use them. Therefore, improved energy efficiency joins scalable performance in the case for multiprocessors.

Since multiprocessor software should scale, some designs support operation in the presence of broken hardware; that is, if a single processor fails in a multiprocessor with n processors, these systems would continue to provide service with n – 1 processors. Hence, multiprocessors can also improve availability (see Chapter 5).

High performance can mean greater throughput for independent tasks, called **task-level parallelism** or **process-level parallelism**. These tasks are independent single-threaded applications, and they are an important and popular use of multiple processors.

This approach contrasts with running a single job on multiple processors. We use the term **parallel processing program** to refer to a single program that runs on multiple processors simultaneously.

There have long been scientific problems that have needed much faster computers, and this class of problems has been used to justify many novel parallel computers over the decades. Some of these problems can be handled simply today, using a **cluster** composed of

microprocessors housed in many independent servers (see Section 6.7). In addition, clusters can serve equally demanding applications outside the sciences, such as search engines, Web servers, email servers, and databases.

Multiprocessor: A computer system with at least two processors. This computer is in contrast to a uniprocessor, which has one, and is increasingly hard to find today.

Task-level parallelism or process-level parallelism: Utilizing multiple processors by running independent programs simultaneously.

Parallel processing program: A single program that runs on multiple processors simultaneously.

Cluster: A set of computers connected over a local area network that function as a single large multiprocessor.

A cluster could for example be search engines, web servers, email servers and databases.

As described in Chapter 1, multiprocessors have been shoved into the spotlight because the energy problem means that future increases in performance must come from some place other than much higher clock rates or vastly improved CPI. As we said in Chapter 1, they are called **multicore microprocessors** instead of multiprocessor microprocessors, presumably to avoid redundancy in naming. Hence, processors are often called *cores* in a multicore chip. The number of cores is expected to increase with improved hardware technology. These multicores are almost always **Shared Memory Processors (SMPs)**, as they usually share a single physical address space. We'll see SMPs more in Section 6.5.

Multicore microprocessor: A microprocessor containing multiple processors ("cores") in a single integrated circuit. Virtually all microprocessors today in desktops and servers are multicore.

Shared memory multiprocessor (SMP): A parallel processor with a single physical address space.

The state of technology today means that programmers who care about performance must become parallel programmers (See Section 6.13).

The tall challenge facing the industry is to create hardware and software that will make it easy to write correct parallel processing programs that will execute efficiently in performance and energy as the number of cores per chip scales.

Speed-up Challenge: Bigger Problem

Suppose you want to perform two sums: one is a sum of 10 scalar variables, and one is a matrix sum of a pair of two-dimensional arrays, with dimensions 10 by 10. For now, let's assume only the matrix sum is parallelizable; we'll see soon how to parallelize scalar sums. What speed-up do you get with 10 versus 40 processors? Next, calculate the speed-ups assuming the matrices grow to 20 by 20.

If we assume performance is a function of the time for an addition, t, then there are 10 additions that do not benefit from parallel processors and 100 additions that do. If the time for a single processor is 110 t, the execution time for 10 processors is

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

$$\text{Execution time after improvement} = \frac{100t}{10} + 10t = 20t,$$

so the speed-up with 10 processors is $110t/20t = 5.5$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{100t}{40} + 10t = 12.5t,$$

so the speed-up with 40 processors is $110t/12.5t = 8.8$. Thus, for this problem size, we get about 55% of the potential speed-up with 10 processors, but only 22% with 40.

Look what happens when we increase the matrix. The sequential program now takes $10t + 400t = 410t$. The execution time for 10 processors is

$$\text{Execution time after improvement} = \frac{400t}{10} + 10t = 50t,$$

so the speed-up with 10 processors is $410t/50t = 8.2$. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{400t}{40} + 10t = 20t,$$

so the speed-up with 40 processors is $410t/20t = 20.5$. Thus, for this larger problem size, we get 82% of the potential speed-up with 10 processors and 51% with 40.

These examples show that getting good speed-up on a multiprocessor while keeping the problem size fixed is harder than getting good speed-up by increasing the size of the problem. This insight allows us to introduce two terms that describe ways to scale up.

Strong scaling means measuring speed-up while keeping the problem size fixed. **Weak scaling** means that the problem size grows proportionally to the increase in the number of processors.

Strong scaling: Speed- up achieved on a multiprocessor without increasing the size of the problem.

Weak scaling: Speed- up achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.

Hardware Multithreading COD 6.4

A related concept to MIMD, especially from the programmer's perspective, is **hardware multithreading**. While MIMD relies on multiple **processes** or **threads** to try to keep many processors busy, hardware multithreading allows multiple threads to share the functional units of a *single* processor in an overlapping fashion to try to utilize the hardware resources efficiently. To permit this sharing, the processor must duplicate the independent state of each thread. For example, each thread would have a separate copy of the register file and the program counter.

There are two main approaches to hardware multithreading. **Fine-grained multithreading** switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on expensive stalls, such as last-level cache misses. This change relieves the need to have thread switching be extremely fast and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall.

Hardware multithreading: Increasing utilization of a processor by switching to another thread when one thread is stalled.

Fine-grained multithreading: A version of hardware multithreading that implies switching between threads after every instruction.

Coarse-grained multithreading: A version of hardware multithreading that implies switching between threads only after significant events, such as a last-level cache miss.

Simultaneous multithreading (SMT) is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled **pipelined** processor to exploit thread-level parallelism at the same time it exploits instruction- level parallelism (see Chapter 4).

Simultaneous multithreading (SMT): A version of multithreading that lowers the cost of multithreading by utilizing the resources needed for multiple issue, dynamically scheduled microarchitecture.

Multicore and other shared memory multiprocessor COD 6.5

While hardware multithreading improved the efficiency of processors at modest cost, a big challenge of the last decade has been to deliver on the traditional performance of Moore's Law by efficiently programming the increasing number of processors per chip.

Given the difficulty of rewriting old programs to run well on parallel hardware, a natural question is: what can computer designers do to simplify the task? One answer was to provide a single physical address space that all processors can share, so that programs need not concern themselves with where their data are, merely that programs may be executed in parallel. In this approach, all variables of a program can be made available at any time to any processor. The alternative is to have a separate address space per processor that requires that sharing must be explicit; we'll describe this option in the Section 6.8. When the physical address space is common then the hardware typically provides cache coherence to give a consistent view of the shared memory (see Section 5.8).

Single address space multiprocessors come in two styles. In the first style, the latency to a word in memory does not depend on which processor asks for it. Such machines are called **uniform memory access (UMA)** multiprocessors.

In the second style, some memory accesses are much faster than others, depending on which processor asks for which word, typically because main memory is divided and attached to different processors or to different memory controllers on the same chip. Such machines are called **nonuniform memory access (NUMA)** multiprocessors.

Uniform memory access (UMA): A multiprocessor in which latency to any word in main memory is about the same no matter which processor requests the access.

Nonuniform memory access (NUMA): A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data; otherwise, one processor could start working on data before another is finished with it. This coordination is called **synchronization**, which we saw in Chapter 2. When sharing is supported with a single address space, there must be a separate mechanism for synchronization. One approach uses a **lock** for a shared variable. Only one processor at a time can acquire the lock, and other processors interested in shared data must wait until the original processor unlocks the variable.

Synchronization: The process of coordinating the behavior of two or more processes, which may be running on different processors.

Lock: A synchronization device that allows access to data to only one processor at a time.

Lecture concurrency

Concurrency means that we have two or more events or circumstances happening or existing at the same time.

From a programming perspective, *concurrency* means multiple *logical control flows* executing simultaneously.

Logical control flow: A stream of execution where the choice of what to do next is made by the code itself (i.e. this is pretty much all code you've written so far).

It is for example used in video games, when rendering graphics and computing physics - opdatering af skærmen imens at der sker andre ting.

CPU core: A piece of hardware that executes instructions—contains registers and one or more levels of cache.

Classic problems of concurrent programming

Races - Outcome depends on arbitrary scheduling decisions elsewhere.

- Example: who gets the last seat on the airplane?

Deadlock - Resource allocation prevents forward progress.

- Example: traffic gridlock. (And programs generally cannot reverse!)

Starvation: External events or scheduling prevents forward progress.

- Example: someone always jumping ahead in line.
- Also known as *livelock* or *fairness*.

Race conditions

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of the shared data are unpredictable and can lead to unexpected behavior. In C, race conditions can occur when using shared global variables or shared memory without proper synchronization. To avoid race conditions, we can use synchronization techniques such as mutexes, semaphores, or atomic operations to ensure that only one thread can access the shared data at a time.

Example of Race conditions

Here's an example of a race condition in C:

```c
#include <stdio.h>
#include <pthread.h>

int shared_variable = 0;

void *increment_thread(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        shared_variable++;
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, increment_thread,
NULL);
    pthread_create(&thread2, NULL, increment_thread,
NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final value of shared variable: %d\n",
shared_variable);
    return 0;
}
```

Here there is a race condition because the two threads access *shared_variable* at the same time. They do this because *shared_variable* is not in a mutex lock. This can also be prevented if the threads create and join are in pairs like this. There is no race condition in this code because the shared variable x is only accessed and modified by one thread at a time. The pthread_create() function creates a new thread, and the pthread_join() function waits for the thread to finish execution. In the main thread, the code first creates and runs a worker thread, then waits for it to finish before creating and running another worker thread. This means that the second worker thread does not start executing until the first worker thread has completed, so there is no overlap in their access to the shared variable x. As a result, the final value of x will be 2000, and it will not be unpredictable.
Here is two examples of no race conditions:

```c
#include <unistd.h>
#include <stdio.h>

int x = 0;

void* worker(void* p) {
  for (int i = 0; i < 1000; i++)
    x++;
  return NULL;
}

int main() {
  pthread_t t;
  pthread_create(&t, NULL, worker, NULL);
  pthread_join(t);
  pthread_create(&t, NULL, worker, NULL);
  pthread_join(t);
  printf("%d\n", x);
}
```

```c
int shared_variable = 0;
pthread_mutex_t mutex;

void *increment_thread(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&mutex);
        shared_variable++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    pthread_mutex_init(&mutex, NULL);
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, increment_thread,
NULL);
    pthread_create(&thread2, NULL, increment_thread,
NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final value of shared variable: %d\n",
shared_variable);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

The unix model of concurrency

A process consists of three parts:

1. A *virtual memory space*.
- Contains stack, code, heap, data, etc.

2. A *kernel context*.
- PID, open files, signal mask, parent PID, list of children, etc.

3. An *execution context*.
- Registers (including special ones like the program counter).

A multi-threaded process model:

A process still consists of three parts:

1. A *virtual memory space*.

- Contains stack, code, heap, data, etc.

2. A *kernel context*.

- PID, open files, signal mask, parent PID, list of children, etc.

3. One or more *threads*, each containing.

3.1 An *execution context*

- Registers(includingspecialoneslikeetheprogramcounter).

3.2 A *kernel thread context*.

- ThreadID(TID),andafewotherthingsthatdonotmatter.

Processes consist of one or more threads!

What is a thread?

Threads in the same process:

- Each thread has its own logical control flow.
- Each thread has its own stack.
- Threads share open files.
- Threads share the same virtual memory space.
- They are peers, there is no “main thread”.

Threads contra processes:

Similarities

- Each has its own logical control flow.
- Each can run concurrently with others (possibly also parallel).
- Each is context switched.

Differences

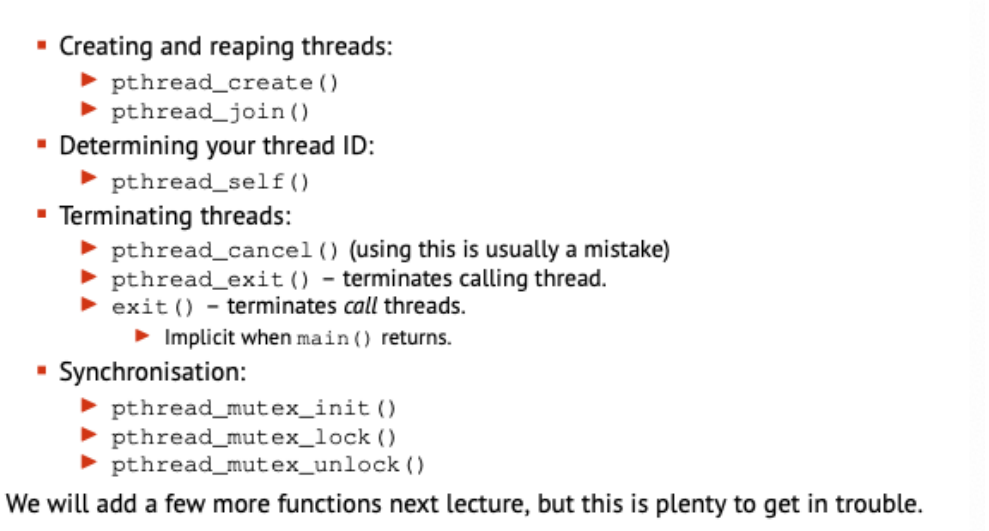
- Threads share code and data.
    - Processes typically do not.
- Threads are cheaper to create and maintain than processes.
    - Take with a grain of salt; both are plenty fast on Linux.
    - But switching between threads within a process does not require switching to a new virtual address space.

Each process has one or more threads.

Each thread belongs to exactly one process.

Threads in code C

POSIX threads - standard thread interface on UNIX.



Semaphores - More under network

A semaphore is a mechanism for synchronizing access to a shared resource. It is typically implemented as a variable or a data structure and is used to control access to a shared resource by multiple processes or threads. Semaphores can be used to implement critical sections, mutual exclusion, and other synchronization constructs. They can also be used to implement blocking or non-blocking operations, depending on the specific implementation.

Explain, in pseudocode or prose, how semaphores can be implemented in terms of mutexes and integer variables. Show how to initialize a semaphore with initial value v, how to implement the P operation, and how to implement the V operation. Could your implementation be made more efficient by also using condition variables?

Semaphores can be implemented using a combination of mutexes and integer variables. Here is an example of how this can be done:

Initialization:

1. To initialize a semaphore with an initial value of v, a mutex and an integer variable are used. The mutex is used to protect the semaphore's value from concurrent access, and the integer variable is used to store the current value of the semaphore. The semaphore is initialized by setting the integer variable to v and initializing the mutex.

```
semaphore s;
s.value = v;
s.mutex = PTHREAD_MUTEX_INITIALIZER;
```

2. P (wait) operation:

The P operation is used to decrement the value of the semaphore. Before the value is decremented, the mutex is acquired to ensure that the value is not being accessed by another process. If the value of the semaphore is greater than 0, it is decremented and the mutex is released. If the value of the semaphore is 0, then the process will block until the semaphore is incremented.

```
void P(semaphore *s) {
     pthread_mutex_lock(&(s->mutex));
     while (s->value <= 0) {
         pthread_cond_wait(&(s->cond), &(s->mutex));
     }
     s->value--;
     pthread_mutex_unlock(&(s->mutex));
}
```

3. V (signal) operation:

The V operation is used to increment the value of the semaphore. The mutex is acquired to ensure that the value is not being accessed by another process. The value of the semaphore is incremented and the mutex is released. If there are any processes blocked on the semaphore, one of them is awakened.

```
void V(semaphore *s) {
     pthread_mutex_lock(&(s->mutex));
     s->value++;
     pthread_cond_signal(&(s->cond));
     pthread_mutex_unlock(&(s->mutex));
}
```

The implementation could be made more efficient by also using condition variables. By using condition variables, the P operation can block the thread only when the semaphore's value is 0, instead of busy waiting.

Additionally, by using condition variable, the V operation can signal only the threads that are waiting, instead of signaling all the threads. This can make the system more responsive and less prone to unnecessary context switching.

Locks OSTEP 28

Locks

Programmers annotate source code with locks, putting them around critical sections, and thus ensure that any such critical section executes as if it were a sin- gle atomic instruction.

A lock is just a variable, and thus to use one, you must declare a lock variable of some kind.

As an example we have a peace of code that is very simple:

```
balance = balance + 1;
```

This is our critical section, and we therefore want to use a lock on it, to do that we add this code around it:

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  unlock(&mutex);
```

Here the lock variable is *mutex.*

This lock variable (or just “lock” for short) holds the state of the lock at any instant in time. It is either **available** (or **unlocked** or **free**) and thus no thread holds the lock, or **acquired** (or **locked** or **held**), and thus exactly one thread holds the lock and presumably is in a critical section. We could store other information in the data type as well, such as which thread holds the lock, or a queue for ordering lock acquisition, but information like that is hidden from the user of the lock.

Calling *lock()* the routine tries to acquire the lock - if no other thread is holding the lock (means that it's free) then we will acquire the lock, this is sometimes called the owner of the lock.

If another thread now tries to call *lock()* on the same look then this new thread will be prevented from entering the lock.

Once the owner of the lock calls *unlock()* then the lock is now available (free) again. If there was another thread that called *lock()* while it was already taken by another thread (and thereby is “waiting” in line to that lock) it will then acquire it and can enter its critical section. If no thread was waiting then the state of the lock is changed to available.

Threads are partly controlled by the operating system (by the OS choicng the priority of the threads) By using locks the programmer gets some of the control back, because we can schedule the execution of the threads better.

Steps for checking the locks

- The first is whether the lock does its basic task, which is to provide **mutual exclusion**. Basically, does the lock work, preventing multiple threads from entering a critical section?

- The second is **fairness**. Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: does any thread contending for the lock **starve** while doing so, thus never obtaining it?

- The final criterion is **performance**, specifically the time overheads added by using the lock.

Lock-based Concurrent Data Structures

Adding locks to a data structure to make it us- able by threads makes the structure **thread safe.**

Concurrent Counters

One of the simplest data structures is a counter. It is a structure that is commonly used and has a simple interface.

perfect scaling - Ideally, you'd like to see the threads complete just as quickly on multiple processors as the single thread does on one.

Approximate counter - works by representing a single logical counter via numerous *local* physical counters, one per CPU core, as well as a single *global* counter.

```
1  typedef struct __counter_t {
2      int             value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Figure 29.2: **A Counter With Locks**

The basic idea of approximate counting is as follows. When a thread running on a given core wishes to increment the counter, it increments its local counter; access to this local counter is synchronized via the corre- sponding local lock. Because each CPU has its own local counter, threads across CPUs can update local counters without contention, and thus updates to the counter are scalable.

However, to keep the global counter up to date (in case a thread wishes to read its value), the local values are periodically transferred to the global counter, by acquiring the global lock and incrementing it by the local counter's value; the local counter is then reset to zero.

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$5 \rightarrow 0$	1	3	4	5 (from L_1)
7	0	2	4	$5 \rightarrow 0$	10 (from L_4)

Figure 29.3: **Tracing the Approximate Counters**

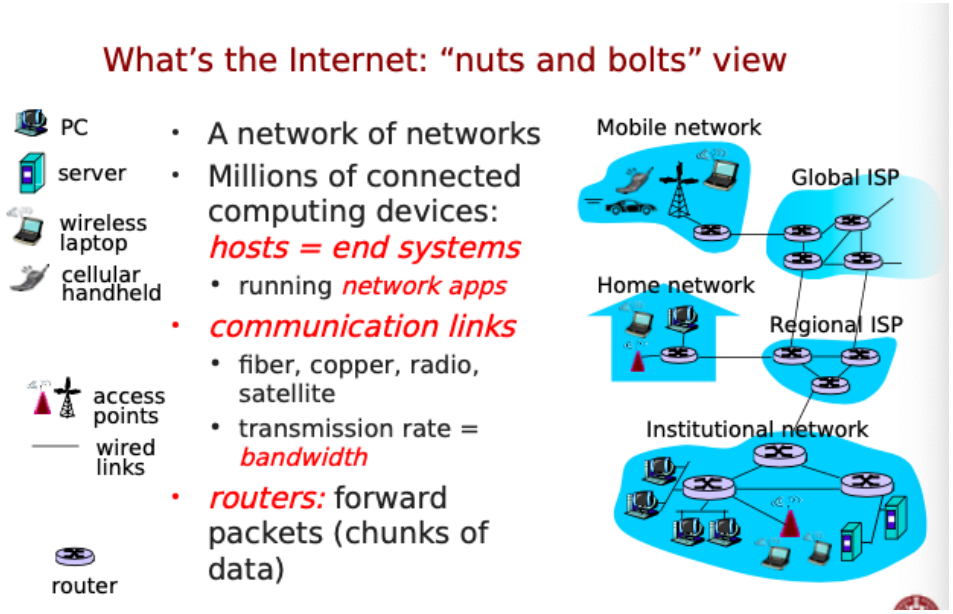
How often this local-to-global transfer occurs is determined by a thresh- old S. The smaller S is, the more the counter behaves like the non-scalable counter above; the bigger S is, the more scalable the counter, but the fur- ther off the global value might be from the actual count. One could sim- ply acquire all the local locks and the global lock (in a specified order, to avoid deadlock) to get an exact value, but that is not scalable.

To make this clear, let's look at an example (Figure 29.3). In this ex- ample, the threshold S is set to 5, and there are threads on each of four CPUs updating their local counters L1 ... L4. The global counter value (G) is also shown in the trace, with time increasing downward. At each time step, a local counter may be incremented; if the local value reaches the threshold S, the local value is transferred to the global counter and the local counter is reset.

Computer Networking

Introduction to computer networking

First, we can describe the nuts and bolts of the Internet, that is, the basic hardware and software components that make up the Internet. Second, we can describe the Internet in terms of a network- ing infrastructure that provides services to distributed applications.



End systems are connected together by a network of **communication links** and **packet switches**.

There are many types of communication links, which are made up of different types of physical media, including coaxial cable, copper wire, optical fiber, and radio spectrum. Different links can transmit data at different rates, with the **transmission rate** of a link measured in bits/second.

Transmission rate:Transmission rate, also known as data rate or bit rate, is a measure of the amount of data that is transmitted over a network in a given period of time. It is usually measured in bits per second (bps) or bytes per second (Bps). The transmission rate can affect the overall performance of a network, as a higher transmission rate allows for faster data transfer and improved communication. However, it is important to note that the transmission rate is not the only factor that affects network performance, and other factors such as network congestion and network distance can also play a role.

Protocols:

How we determine, how we can communicate. Language etc.

- Speaking the same language

- Syntax and semantics

Layering = functional Abstraction:

Network of networks, we can divide the network into different layers, and if we we move around in the same layer it shouldn't matter. Some make the cables, some make the routers, and so on. So one layer doesnt do everything.

- Standing on the shoulders of giants
- A key to managing complexity

Sub-divide the problem

- Each layer relies on services from layer below
- Each layer exports services to layer above

Resource allocation:

Where does the different request go

- Dividing scarce resources among competing parties
- Memory, link bandwidth, wireless spectrum, paths

Network Core

Circuit Switching:

Traditional architecture, Physically making the connection, like the first telephones switching the lines when people wanted to talk to each other.

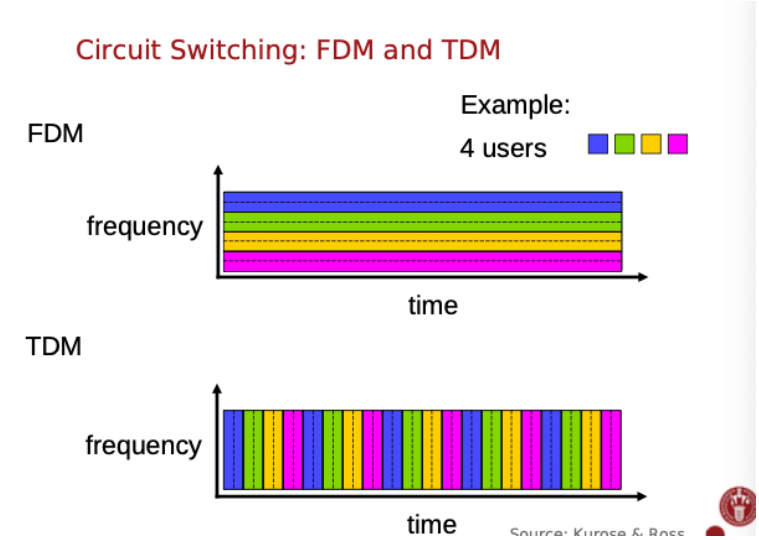
You can't share the connection. There needs to be a physical connection from the caller to the responder, fx via cable. Very efficient for the one user, because they can use all of the network by themselves.

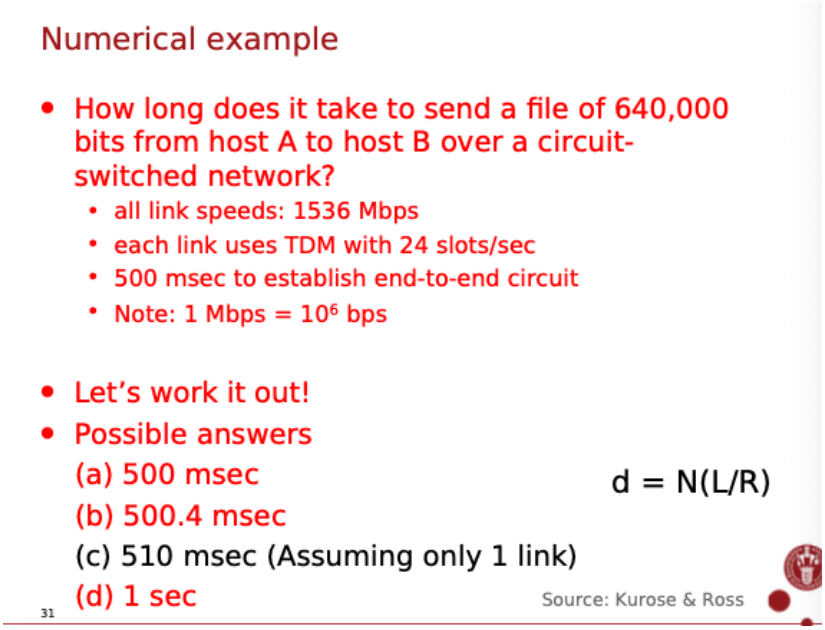
We don't use this kind of switching anymore.

Circuit switching: is a method of transmitting data in a network in which a dedicated physical connection is established between the sender and the receiver for the duration of the communication. An example of circuit switching is the traditional telephone system. It guarantees a dedicated and constant bandwidth but also has some disadvantages such as the waste of resources when the connection is not in use, the difficulty to scale and the lack of flexibility to adapt to changing traffic patterns.

FDM = frequency

TDM = Time





$1536/0,64/24 \;=\; 100 * 0,1 \;=\; 10$

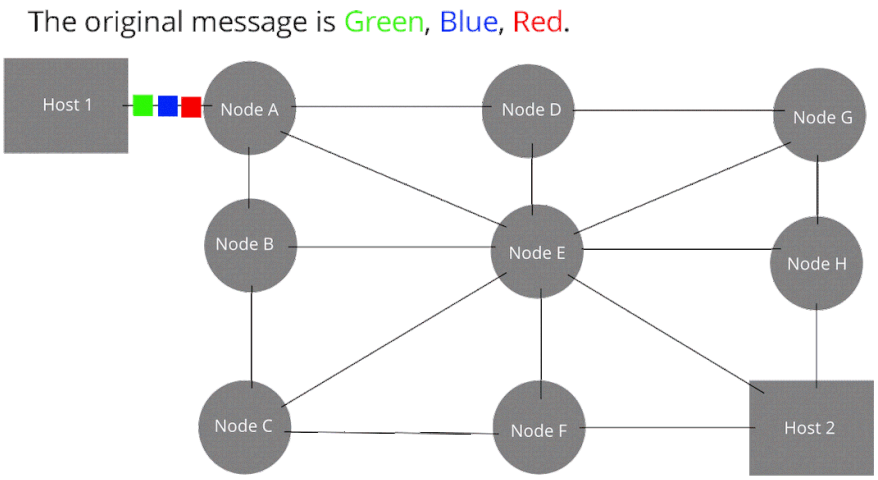
$10 + 500 \;=\; 510\, msec$

Packet Switching:

the sharing of network resources among multiple users and adapts to changing traffic patterns, it also requires more complex routing and error-checking mechanisms. Packet switching can be done at the link-layer or at the network layer, in link-layer packet switching,

the switch uses the destination MAC address, in network-layer packet switching, the router uses the destination IP address.

Packet switching is a method of transmitting data across a network in which data is divided into small units called packets, and each packet is sent independently through the network to its destination, unlike circuit switching, packets are sent through the network using the best available path and are not guaranteed to follow the same path or arrive in the same order, each packet contains the necessary information to be transmitted across the network such as destination and source address, and it is the responsibility of each device, such as routers and switches, to forward the packets to their destination.



Packet switches come in many shapes and flavors, but the two most prominent types in today's Internet are **routers** and **link-layer switches**

Routing: is a method of packet switching that uses routing tables and protocols to determine the best path for a packet to take to reach its destination. Routing packets through the network is done by routers that read the destination address in the packet header and forward it to the next hop based on the routing table.

Link-layer switching: also known as data link switching, is a method of packet switching that operates at the data link layer of the OSI model. It uses a switch to forward packets based on the destination MAC address, which is present in the packet header. Link-layer switches are faster than routers as they work on the link layer which is faster than network layer.

End systems/hosts access the Internet through **Internet Service Providers (ISPs)**

Internet Service Providers (ISPs): are companies that provide internet access to customers through various technologies such as cable, DSL, fiber-optic, satellite, and wireless. ISPs are responsible for connecting customers to the internet and providing them with a connection that allows them to access the World Wide Web, send and receive emails, and use other online services.

End systems, packet switches, and other pieces of the Internet run **protocols** that control the sending and receiving of information within the Internet. The **Transmission Control Protocol (TCP)** and the **Internet Protocol (IP)** are two of the most impor- tant protocols in the Internet.

TCP is a transport layer protocol: that ensures that data is transmitted reliably and in the correct order. It establishes a virtual connection between two devices and ensures that data is transmitted error-free by using flow control and error checking mechanisms.

IP is a network layer protocol: that is responsible for routing data packets across networks. It assigns unique IP addresses to devices and uses them to identify the source and destination of data packets. IP packets can be routed through multiple networks, which allows for the internet to be a global network of networks.

End systems attached to the Internet provide a **socket interface** that specifies how a program running on one end system asks the Internet infrastructure to deliver data to a specific destination program running on another end system.

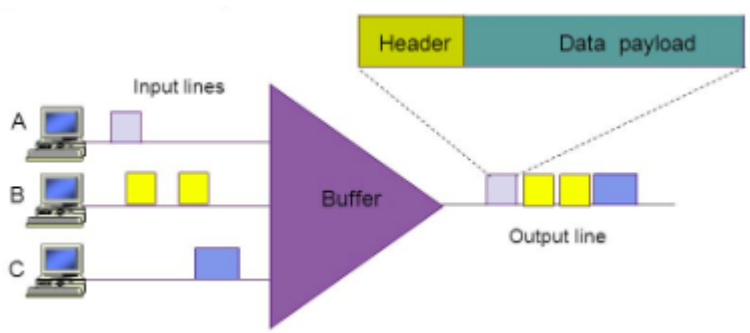
A socket is an endpoint of a two-way communication link between two devices in a network. It is a combination of an IP address and a port number, and it is used to identify a specific process running on a device. The socket interface is a set of programming instructions that allow applications to communicate over a network using the TCP/IP protocols.

Hosts are sometimes further divided into two categories: **clients** and **servers**.

A client: is a device or application that requests services or resources from a server. The client typically initiates the communication and sends a request to the server. The server then responds to the client's request by providing the requested services or resources. Examples of clients include web browsers, email clients, and instant messaging applications.

A server: is a device or application that provides services or resources to clients. It is responsible for processing requests from clients and returning the requested information. Servers typically have more powerful hardware and software than clients and are designed to handle multiple requests simultaneously. Examples of servers include web servers, email servers, and file servers.

Statistical Multiplexing:



1. **Statistical multiplexing** is performed by switching systems in communication networks that merge data packets from multiple input lines and forward them to multiple outputs in a first come first serve or other scheduling discipline. In this way, many data flows can share capacity on a common transmission path. The aggregation of flows in a multiplexer is governed by **statistical** laws, such that the entire flow usually shows a smoothed rate variability as compared to single flow components.

2. **Statistical multiplexing** is a method of making the most efficient use of the bandwidth available. Different flows share the same static resource (e.g. the bandwidth of a central link) and the idea is to allocate the bandwidth to each flow in order to prevent peaks from occurring at the same time on all the flows, which would result in a packet loss.

Most packet switches use **store-and-forward transmission** at the inputs to the links

Store-and-forward:

- Entire packet must arrive at the router before it can be transmitted on the next link.
- It takes L/R seconds to transmit a packet of L bits on to link at R bps.

This will cause some delays.

Store-and-forward transmission: is a method used by packet switches to forward data packets in a network. In this method, the packet switch receives a packet on one of its input ports, stores the entire packet in memory, and then checks for errors before forwarding it to the appropriate output port. This allows the packet switch to detect and discard any corrupted packets before they are transmitted further into the network.

The main advantage of store-and-forward transmission is that it ensures the integrity of the data being transmitted by checking for errors before forwarding the packet. This can prevent the spread of corrupted packets throughout the network and improve overall network performance. However, it can also cause a small delay in transmission, as the packet switch must wait for the entire packet to be received before forwarding it.

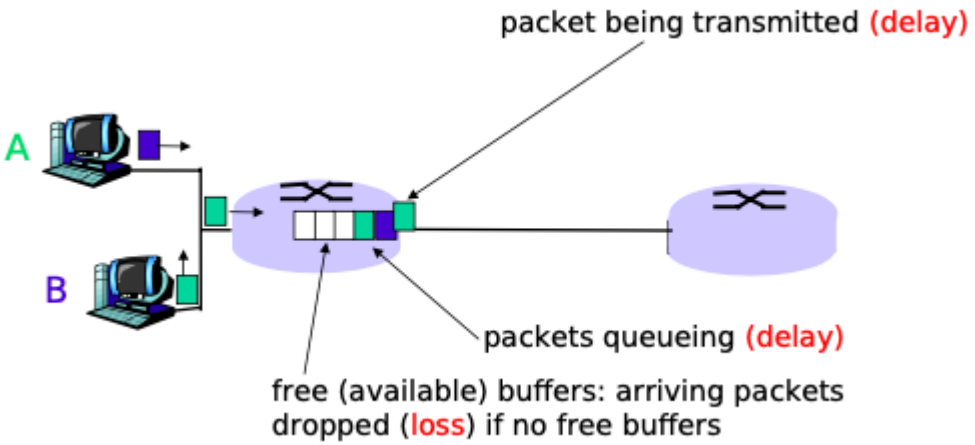
Cut-through switching: is a method used by packet switches to forward data packets in a network, in this method, the switch reads the destination address of the packet as soon as it is received and immediately begins forwarding the packet to the appropriate output port, the main advantage of cut-through switching is that it reduces the delay in transmission but it also has some drawbacks, it cannot check for errors before forwarding the packet, which can lead to the spread of corrupted packets throughout the network.

Loss and delay:

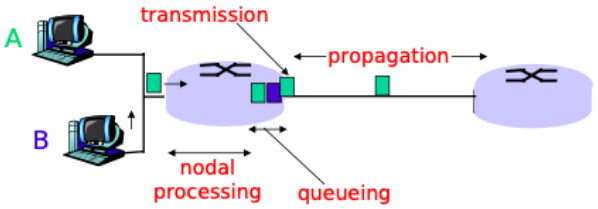
If the packets are sent into the router buffer faster than the router buffer can send it out then we lose the packet, because of fx overwriting of already waiting packets.

- packets queue in router buffers
- packet arrival rate to link exceeds output link capacity
- packets queue, wait for turn - At the queue, the packet experiences a **queuing delay** as it waits to be transmitted onto the link.
- Assuming that packets are transmitted in a first-come-first-served manner, as is common in packet-switched networks, our packet can be transmitted only after all the packets that have arrived before it have been transmitted.
- The time required to examine the packet's header and determine where to direct the packet is part of the **processing delay**
- Propagation delay is the time it takes for a packet to travel through the physical medium of the network.

End-to-end delay: is a measure of the time it takes for a packet of data to travel from its source to its destination in a network, it is the sum of all the delays that a packet of data experiences in the network, including transmission delay, queuing delay, processing delay, and propagation delay, end-to-end delay can be affected by several factors such as network congestion, network distance, and network topology, it is desired to keep the end-to-end delay as low as possible as it affects the overall performance of the network and the user experience.



Four sources of packet delay



$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$

d_{trans}: transmission delay

- L: packet length (bits)
- R: link bandwidth (bps)
- $d_{trans} = L/R$

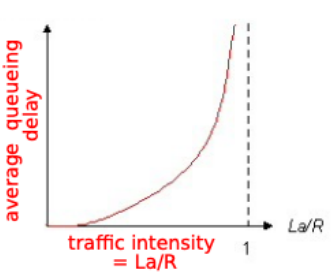
d_{prop}: propagation delay

- d: length of physical link
- s: propagation speed in medium (~$2x10^8$ m/sec)
- $d_{prop} = d/s$

d_{trans} and d_{prop} *very* different

Queuing delay

- R: link bandwidth (bps)
- L: packet length (bits)
- a: average packet arrival rate



- La/R ~ 0: avg. queuing delay small
- La/R → 1: avg. queuing delay large
- La/R > 1: more "work" arriving than can be serviced, average delay infinite!

Packet switching VS Circuit switching:

Packet switching and circuit switching are two methods of transmitting data across a network. They differ in how they handle and transmit data:

Circuit switching:

- Establishes a dedicated connection, or circuit, between the sender and receiver for the duration of the communication.

- The circuit is reserved exclusively for the use of the sender and receiver and is not shared with any other devices.
- Guarantees a dedicated and constant bandwidth for the duration of the connection.
- Ensures that the data is transmitted in the correct order.
- Allows for real-time communication.
- However, it also has some disadvantages, such as the waste of resources when the connection is not in use, the difficulty to scale and the lack of flexibility to adapt to changing traffic patterns.

Packet switching:

- Divides data into small packets and sends them through the network to their destination through multiple paths.
- Allows for the sharing of network resources among multiple users and adapts to changing traffic patterns.
- Requires more complex routing and error-checking mechanisms.
- More efficient use of network resources.
- However, packets may take different routes through the network, and they may arrive at the destination out of order.

In summary, Circuit switching establishes a dedicated connection between sender and receiver while packet switching divides data into small packets and sends them through the network to their destination through multiple paths, Circuit switching guarantees a dedicated and constant bandwidth but Packet switching allows for the sharing of network resources among multiple users and adapts to changing traffic patterns.

Throughput:

Throughput: is a measure of the amount of data that can be transmitted over a network in a given period of time, it is typically measured in bits per second or bytes per second, it is a key metric that is used to evaluate the performance of a network, a higher throughput indicates better network performance, as it means that more data can be transmitted in a given period of time. It can be calculated by measuring the actual data rate of a network over a period of time or by measuring the capacity of a network.

Throughput: Rate (bit/time unit) at which bits transferred between sender/retriever. Before problems will happen.

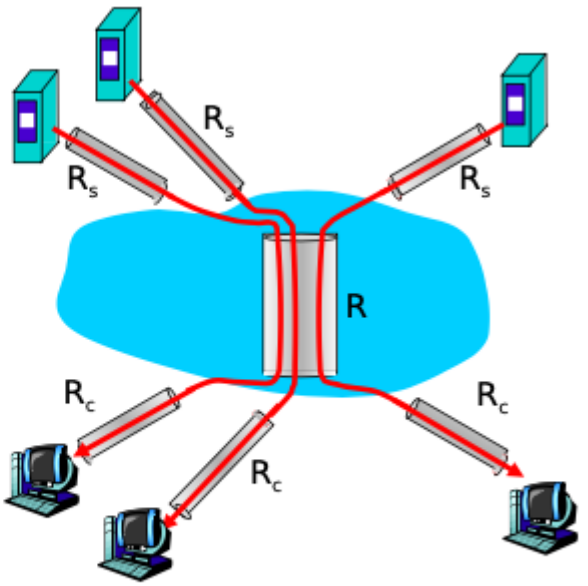
Instantaneous: rate at given point in time

Average: rate over longer period of time

Throughput: Internet scenario

There will always be a bottleneck somewhere in a system, which is the weakest link.

- per-connection end-end throughput: $\min(R_c, R_s, R/10)$
- in practice: R_c or R_s is often bottleneck



10 connections (fairly) share backbone bottleneck link R bits/sec

A bottleneck: is a point in a network or system where the capacity is lower than the demand, causing a reduction in overall performance, it is a constraint that limits the overall performance of a system. In a network, a bottleneck can occur when a device or link has a lower capacity than the other devices or links, in a computer system, a bottleneck can occur when a component such as the CPU, memory, or storage device has a lower performance than the other components in the system. The slower component can limit the performance of the faster components.

Opgaver i Computer networking

R13. Suppose users share a 2 Mbps link. Also suppose each user transmits contin- uously at 1 Mbps when transmitting, but each user transmits only 20 percent of the time. (See the discussion of statistical multiplexing in Section 1.3.)

1. When circuit switching is used, how many users can be supported?
    - Since each user only uses 1 mpbs when transmitting, then if there is 2 mpbs then there can be 2 users transmitting at the same time.

2. For the remainder of this problem, suppose packet switching is used. Why will there be essentially no queuing delay before the link if two or fewer users transmit at the same time? Why will there be a queuing delay if three users transmit at the same time?
    - If we have 2 users then they use all of the 2 Mpbs bandwidth, because they use 1 mbps each. And if there are fewer, then it only uses 1 mpbs. Then if there are 3 users they would require 3 Mpbs for them to transmit at the same time. But there is only 2 mpbs and therefore there would acquire queuing delays, and the last user would have to stay in the queue until one of the others are done.

3. Find the probability that a given user is transmitting.
    - The probability must be 0.2, because each user is transmitting 20 percent of the time or ⅕ of the time.

4. Suppose now there are three users. Find the probability that at any given time, all three users are transmitting simultaneously. Find the fraction of time during which the queue grows.
    - $P^3 = (0,2)^3 = 0,008$ because we take the probability for one user and takes it in 3 because now its three users.

R14. Why will two ISPs at the same level of the hierarchy often peer with each other? How does an IXP earn money?

- If the two ISPs do not peer with each other, then when they send traffic to each other they have to send the traffic through a provider ISP (intermediary), to which they have to pay for carrying the traffic. By peering with each other directly, the two ISPs can reduce their payments to their provider ISPs. An Internet Exchange Points (IXP) (typically in a standalone building with its own switches)is a meeting point where multiple ISPs can connect and/or peer together. An ISP earns its money by charging each of the ISPs that connect to the IXP a relatively small fee, which may depend on the amount of traffic sent to or received from the IXP.

P4. Consider the circuit-switched network in Figure 1.13. Recall that there are four circuits on each link. Label the four switches A, B, C, and D, going in the clockwise direction.

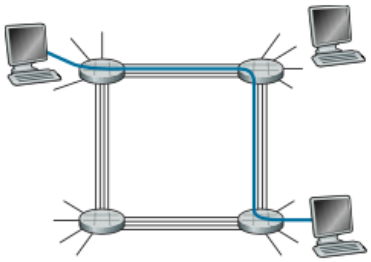


Figure 1.13 ♦ A simple circuit-switched network consisting of four switches and four links

1. What is the maximum number of simultaneous connections that can be in progress at any one time in this network?
    - There can be 4 times 4 connections simultaneously = 16

2. Suppose that all connections are between switches A and C. What is the maximum number of simultaneous connections that can be in progress?
    - There can be 4 times 2 connections simultaneously = 8

3. Suppose we want to make four connections between switches A and C, and another four connections between switches B and D. Can we
route these calls through the four links to accommodate all eight connections?
    - Yes, its not at the same time A to C sends two connections through B, and B sends two through A to D. therefore all 4 links are used.

P13. Suppose *N* packets arrive simultaneously to a link at which no packets are currently being transmitted or queued. Each packet is of length *L* and the link has transmission rate *R*.

1. What is the average queuing delay for the *N* packets?
    - The queuing delay is 0 for the first package, L/R for the second package, and in generally $(n-1)L/R$ for the nth package.

Now suppose that *N* such packets arrive to the link every *LN/R* seconds.

2. What is the average queuing delay of a packet?

- It takes LN/R seconds to transmit the N packets. Thus, the buffer is empty when each batch of N packets arrive. Thus, the average delay of a packet across all batches is the average delay within one batch, i.e., $(N-1)L/2R$.

UNIX I/O

- A file is a sequence of m bytes

Cool fact: All I/O devices are represented as files:

/dev/tty (the current terminal)

/dev/sda2 (a disk partition)

/dev/tty2 (some other terminal)

Even the kernel is represented as a file:

/boot/vmlinuz-3.13.0-55-generic (kernel image)

/proc (process information)

/sys (kernel data structures)

Use ***htonl***, ***htons***, ***ntohl*** and ***ntohs*** functions for network byte order conversions

Htonl = host long

Htons = host short

In UNIX, I/O (input/output) operations are performed through the use of file descriptors. A file descriptor is a small integer value that is used to identify a specific file or device that the program wants to read from or write to. UNIX provides a set of system calls that can be used to create, manipulate, and close file descriptors.

Some of the most commonly used UNIX I/O system calls include:

- open() - opens a file or device and returns a file descriptor
- read() - reads data from a file descriptor into a buffer
- write() - writes data from a buffer to a file descriptor
- lseek() - moves the file pointer to a specific position within a file
- close() - closes a file descriptor

UNIX also provides a set of higher-level I/O functions called "streams" that are built on top of the file descriptor interface. These functions provide a more convenient way to perform I/O operations and include functions such as fopen(), fread(), fwrite(), and fclose().

Internet connections

Clients and servers communicate by sending streams of bytes over connections. Each connection is:

- Point-to-point: connects a pair of processes.
- Full-duplex: data can flow in both directions at the same time,
- Reliable: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.

A socket is an endpoint of a connection

- Socket address is an IPaddress:port pair

A port is a 16-bit integer that identifies a process:

- **Ephemeral port:** Assigned automatically by client kernel when client makes a connection request.
- **Well-known port:** Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

The Sockets Interface

The socket interface: provides a common set of functions that can be used to create, manage, and terminate network connections. These functions include:

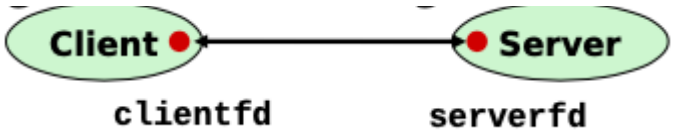
- socket() - creates a new socket
- bind() - assigns a local address and port to a socket
- listen() - sets a socket to listen for incoming connections
- accept() - accepts an incoming connection on a socket
- connect() - establishes a connection to a remote socket
- send() and recv() - used to send and receive data over a socket
- close() - closes a socket

What is a socket?

- To the kernel, a socket is an endpoint of communication
- To an application, a socket is a file descriptor that lets the application read/write from/to the network

Remember: All Unix I/O devices, including networks, are modeled as files

Clients and servers communicate with each other by reading from and writing to socket descriptors



Host and Service Conversion: getaddrinfo

getaddrinfo is a way to convert string representations of hostnames, hostadresses, ports and service names to socket address structures.

Advantages

- can be safely used by threaded programs
- allows us to write portable protocol-independent code

Disadvantages

- Somewhat complex
- Fortunately, a small number of usage patterns suffice in most cases.

Socket programming example

Echo server and client

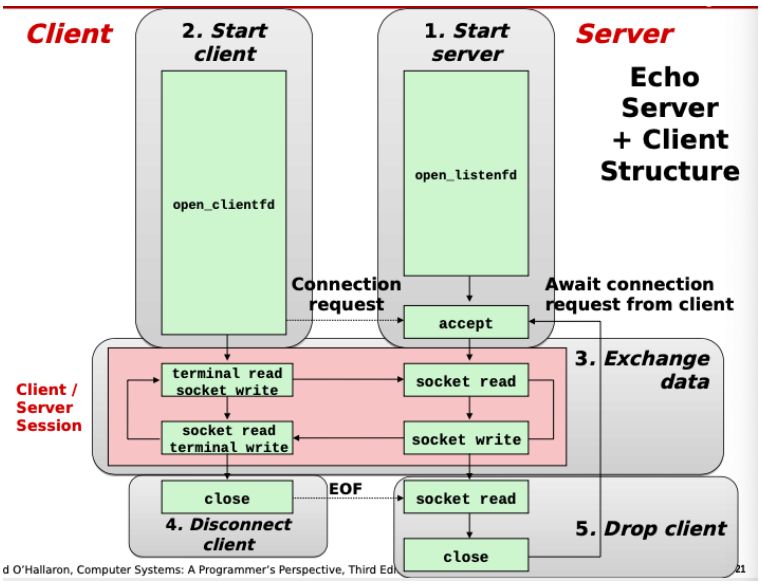
Server

- Accepts connection request
- Repeats back lines as they are typed

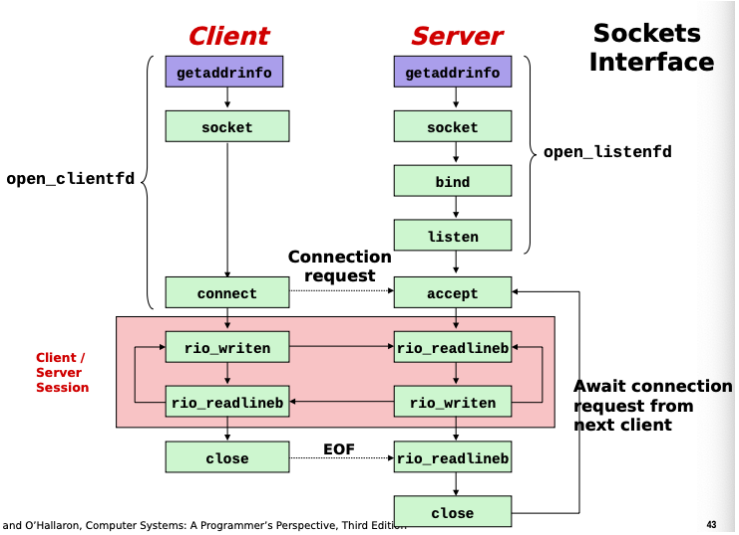
Client

Requests connection to server repeatedly:

- Read line from terminal
- Send to server
- Read reply from server
- Print line to terminal



This is what happens when a client and server are connected.



The same connection, but more broken down into pieces of what happens.

Socket:

Clients and servers use the socket function to create a socket descriptor.

Bind:

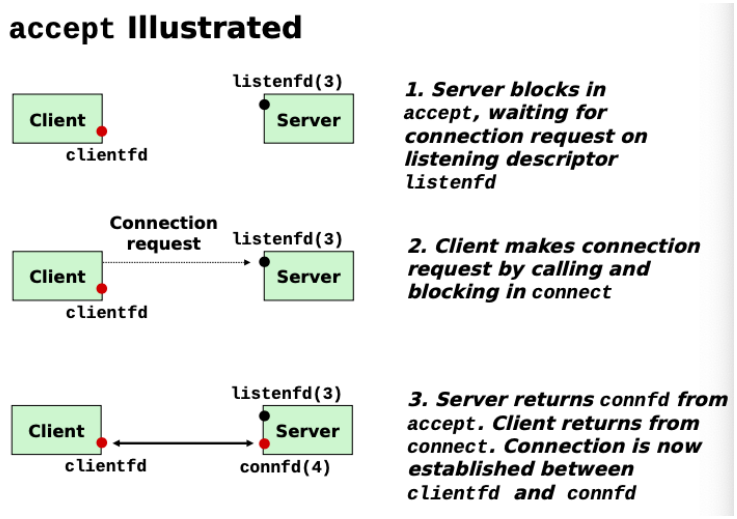
A server uses bind to ask the kernel to associate the server's socket address with a socket descriptor.

Listen:

A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client.

Accept:

Servers wait for connection requests from clients by calling accept.



Connect:

A client establishes a connection with a server by calling connect.

Short Counts

Short counts often occurs in these situations:

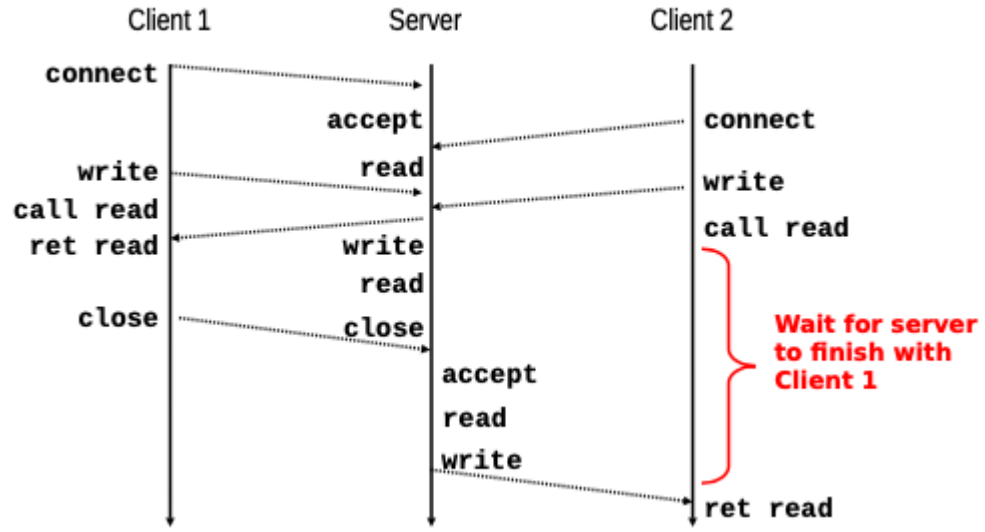
- Encountering (end-of-file) EOF on reads
- Reading text lines from a terminal
- Reading and writing network sockets

Best practice is to always allow for short counts.

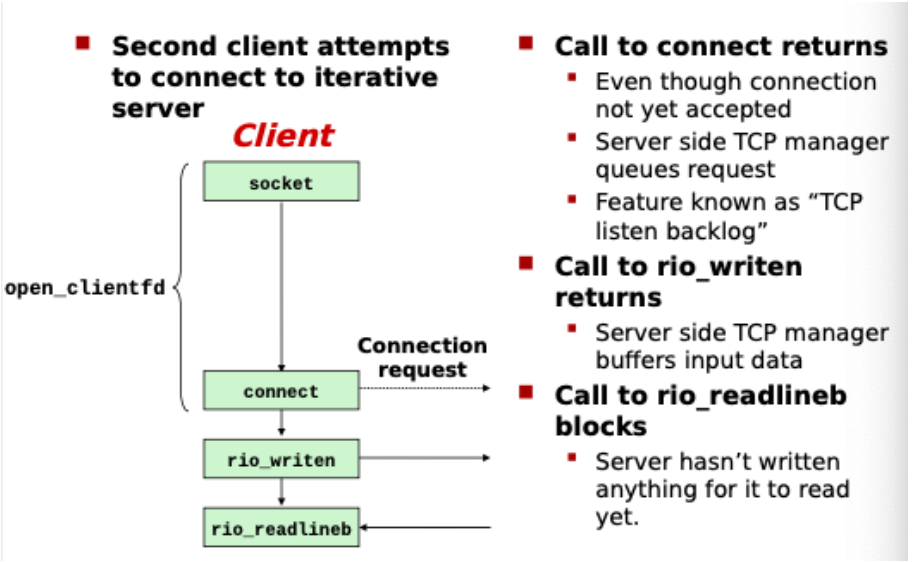
Non-blocking network programming and introducing security: Hashes and Salt

Iterative Servers

- iterative servers process one request at a time



Where does second client block?



Writing a concurrent servers

- uses multiple concurrent flows to serve multiple clients at the same time.
- It allows servers to handle more than one client at the same time.

Process-based servers

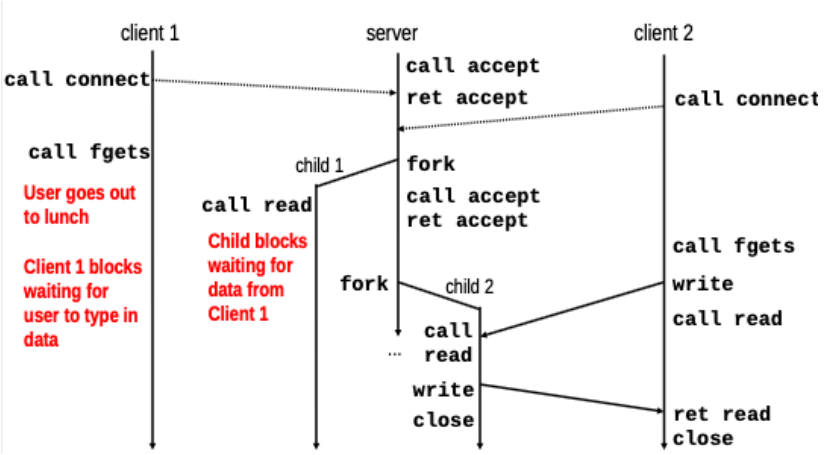
- spawn separate process for each client
- Handle multiple connnections concurrently
- clean sharing model
    - descriptors (no)
    - file tables (yes)
    - global variables (no)

- simple and straightforward

problems:

- listening server process must reap zombie children to avoid fatal memory leak
- Parent process must *close* its copies of *connfd*
    - kernel keeps reference count for each socket/open file
    - after fork, *refcnt(connfd) = 2*
    - Connection will not be closed until *refnt(connfd) = 0*

- Nontrivial to share data between processes



Event-based servers

- one logical control flow
- can single-step with debugger
- no process or thread control overhead
- server maintains set of active connections
    - array of *connfd´s*

problems

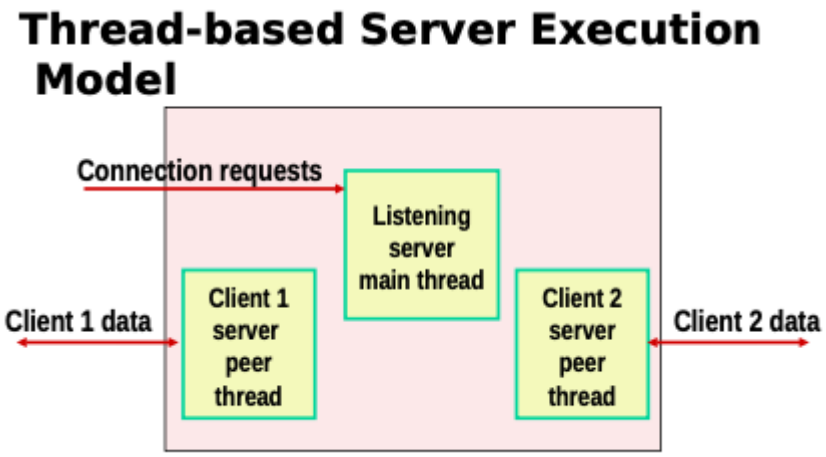
- significantly more complex to code than process or thread based designs.
- Hard to provide fine-grained concurrency
- can't take advantage of multi-core

Thread-based Servers

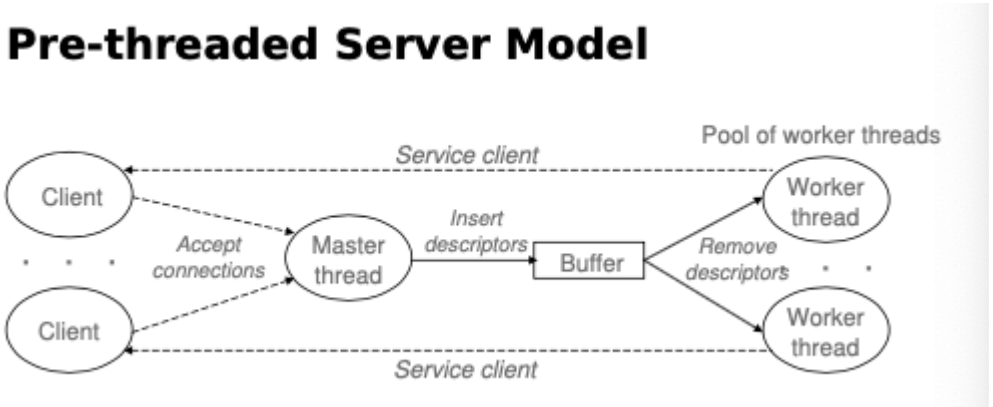
- Very similar to process-based, just uses threads instead
- Threads are more efficient that processes
- easy to share data structures between threads
    - logging information, file cache

problems

- can give problems if more than one thread wants to access information at the same time.
- Unintentional sharing can introduce subtle and hard-to-reproduce errors
    - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
    - Hard to know which data shared & which private
    - Hard to detect by testing, difficult to debug



- each client handled by individual peer thread
- threads share all process state except TID
- each thread has a separate stack for local variables



- clients handled using thread-pool architecture
- bounded/unbounded buffer can be used for synchronization

Introducing security

This lecture will not cover all of security, but we are introducing it now for context, and will expect you to consider it in both networking courseworks

- Everyone talks about security
- Everyone says its important
- Everyone agrees that its a problem

There is no single simple, or even complex answer here, so it is often forgotten (See the course reading for an example)

Internet design: Insecure

- design for simplicity
- “on by default design
- Readily available zombie machines
- attacks look like normal traffic
- internet´s federated operation obstructs cooperation for diagnosis/mitigation

Security properties

Confidentiality: Concealment of information or resources: Ensuring that sensitive information is only accessible to authorized users.

Authenticity: Identification and assurance of origin of info: Verifying the identity of users trying to access a system or network.

Integrity: Trustworthiness of data or resources in terms of preventing improper and unauthorized changes: Ensuring that data and systems are protected against unauthorized modification or tampering.

Availability: Ability to use desired info or resource **Non-repudiation:** Offer of evidence that a party indeed is sender or a receiver of certain information : Ensuring that systems and services are accessible to authorized users when needed.

Access control: Facilities to determine and enforce who is allowed access to what resources (host, software, network, ...)

Non-repudiation: Ensuring that a user cannot deny having performed a particular action.

Anonymity: Allowing users to remain anonymous while accessing a system or network.

Auditing: Keeping records of system and user activity for later review.

Encryption: Converting data into a code to protect it from unauthorized access.

Firewall: A barrier that controls the flow of incoming and outgoing traffic to a network.

Intrusion detection/prevention: Monitoring a system for signs of unauthorized access or attack.

Disaster recovery: Having plans and procedures in place to restore a system or network after a disaster or outage.

Hash functions

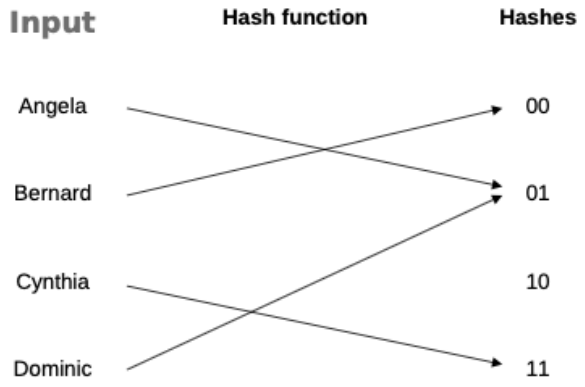
- Function for deterministicly computing a fixed-length output from some variable length input
- Minimise output collisions (eg two inputs producing the same output)
- Idealy is fast to compute, but time consuming to reverse engineer

---

A note on cryptography

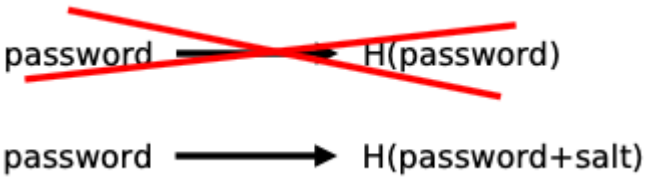
- You may encounter the term ‘cryptographic hash functions’
- These are a subset of hash functions, but we will not really discuss them yet. They are much slower and conversely much more difficult to reverse engineer
- We will return to these a few times, later in the course

Hash tables

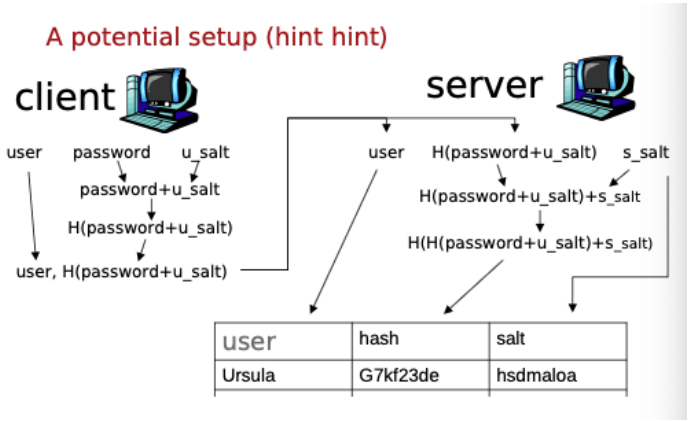


Using Salt

- Because hashing is deterministic, the same input will have the same output. Definitely a problem when users can’t be relied upon to have unique passwords
- Second problem is that rainbow tables of hashes of common passwords definitely exist and are used



- Salts should ideally be long, random, and unique to ensure minimum chances of collisions
- All the rainbow tables in the world won’t help if our passwords are salted before being hashed
- Note the salt is stored as plaintext as we need it to verify our salted hash, but that isn’t a problem really
- This doesn't make it impossible to still brute force, but each hash needs to compute independently



(this is for the assignment)

- If a user-remembered password gets compromised, the attacker still needs to find out the salt to gain access
- Password tables become much less effective
- Randomly generating salt means rainbow tables won't help either
- If the message gets intercepted the user-remembered password is still obscured
- Repeated hashes of salted user passwords are extremely unlikely
- If the user database gets compromised, hashes of salted passwords are obscured
- User passwords never even make it into the server

DNS Peer to Peer and UDP

DNS (Domain Name System) is a hierarchical (When DNS is referred to as hierarchical, it means that it is organized into a tree-like structure, with different levels or domains of authority. At the top level, there are the root servers, which are responsible for managing the top-level domains such as .com, .org, .edu, etc. The root servers provide a starting point for resolving domain names.) , distributed naming system for computers, services, or any resource connected to the Internet or a private network. It translates human-friendly domain names, such as www.example.com, into IP addresses that are used by devices to communicate with each other.

There are two main types of DNS: traditional client-server DNS and Peer-to-Peer (P2P) DNS.

In traditional client-server DNS, a client sends a request to a DNS server, which then responds with the requested information. The client-server model is centralized and requires a reliable connection to the DNS server for proper resolution.

Peer-to-Peer (P2P) DNS, on the other hand, is a distributed model in which DNS information is shared among a group of peers rather than relying on a centralized server. In P2P DNS, each peer acts as both a client and a server, allowing for more robust and decentralized resolution. P2P DNS systems can be more resistant to failure and censorship, and they may also offer improved privacy and security.

UDP (User Datagram Protocol) is a transport layer protocol that is commonly used in conjunction with DNS. DNS queries and responses are typically sent over UDP because it is a connectionless and lightweight protocol that is well-suited for the small, short-lived transactions that are typical of DNS. However, DNS can also use TCP (Transmission Control Protocol) for larger or more critical transfers.

In summary, DNS is a hierarchical, distributed naming system for computers, services, or any resource connected to the Internet or a private network, there are two main types of DNS: traditional client-server DNS and Peer-to-Peer (P2P) DNS, the first relies on a centralized server while the second is distributed among a group of peers, UDP (User Datagram Protocol) is a transport layer protocol that is commonly used in conjunction with DNS, DNS queries and responses are typically sent over UDP because it is a connectionless and lightweight protocol that is well-suited for the small, short-lived transactions that are typical of DNS.

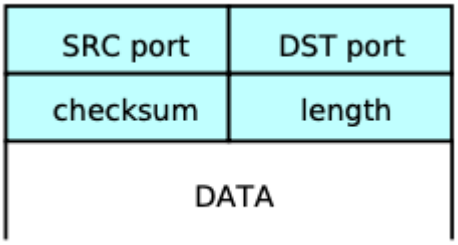
User datagram protocol (UDP)

Datagram messaging service

- Demultiplexing of messages: port numbers
- Detecting corrupted messages: checksum

Lightweight communication between processes

- Send messages to and receive them from a socket
- Avoid overhead and delays of ordered, reliable delivery



We use UDP when we want a fine control over what data is sent and when.
There is no delay for connection establishment, no buffers.
Small packet header overhead.

Separating Names and IP addresses

- Host name never change
- The IP address under the hostname changes
- Name could map to multiple IP addresses
- Map to different addresses in different places

Domain Name System (DNS)

Computer science concepts underlying DNS

- indirection: names in place of addresses
- Hierarchy: in names, addresses, and servers
- Caching: of mappings from names to/from addresses

DNS software components

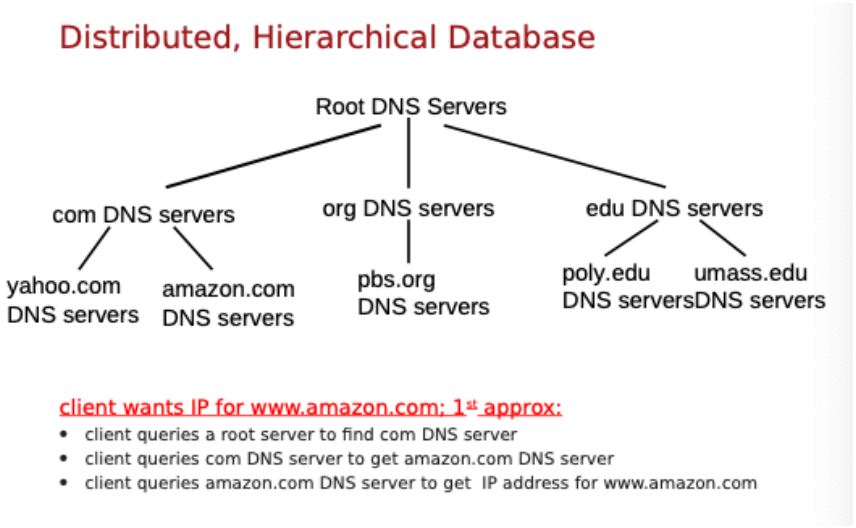
- DNS resolvers (component dealing with the client that request something)
- DNS servers (the server that translates the IP addresses into names and the other way)

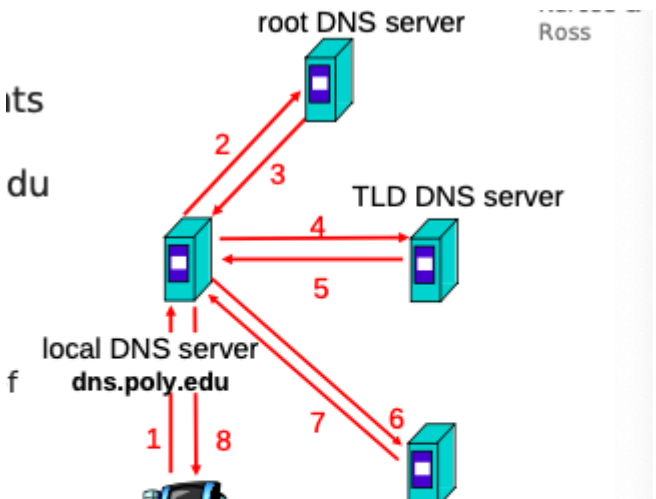
DNS queries

- Iterative queries (The client communicates directly with each DNS server involved in finding the IP address)
- recursive queries (Where one DNS server communicates with several other DNS servers to hunt down an IP address and return it to the client)
- DNS caching based on time-to-live (TTL) (TTL is how long the server should hold the information in the cache)

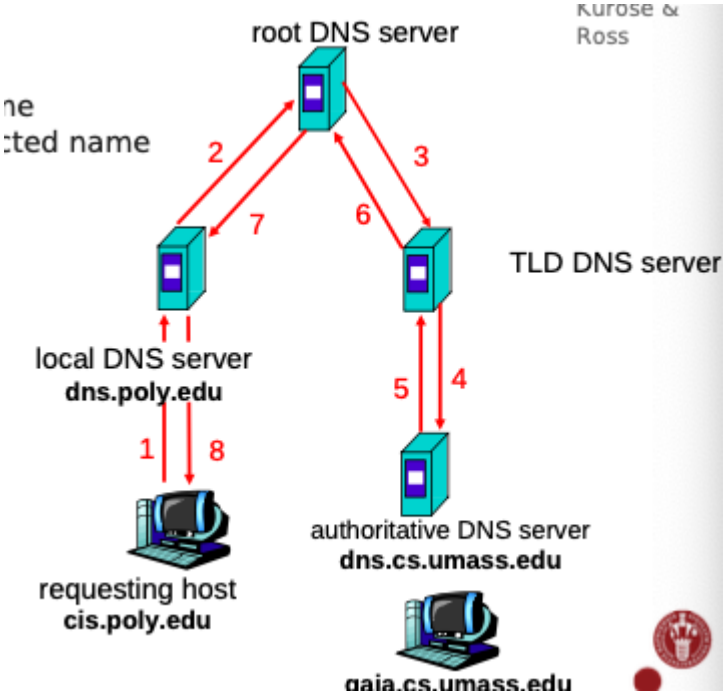
Hierarchy of DNS servers

- root servers
- Top-level domain (TLD) servers
- Authoritative DNS servers





Example on a DNS that uses iterated query



Example of DNS server using the recursive query.

Time to live and negative caching

How DNS caching works

- DNS servers cache responses to queries
- Responses include a "time to live" (TTL) field
- Server deletes the cached entry after TTL expires

Negative Caching: Remember things that do not work

- Misspellings like www.cnn.comm and www.cnnn.com
- These can take a long time to fail the first time
- Good to remember that they don't work
- ... so the failure takes less time the next time around

DNS Security

DNS cache poisoning

- ask for www.evil.com
- Additional section for (www.cnn.com, 1.2.3.4, A) -
- Thanks! I won't bother check what I asked for

DNS hijacking

- Let’s remember the domain. And the UDP ID (source port + transaction ID).
- 16 bits: 65K possible transaction IDs
  • What rate to enumerate all in 1 sec? 64B/packet
  • 64*65536*8 / 1024 / 1024 = 32 Mbps
- Prevention: Also randomize the DNS source port
  • E.g., Windows DNS alloc’s 2500 DNS ports: ~164M possible IDs • Would require 80 Gbps
  • Kaminsky attack: this source port...wasn’t random after all

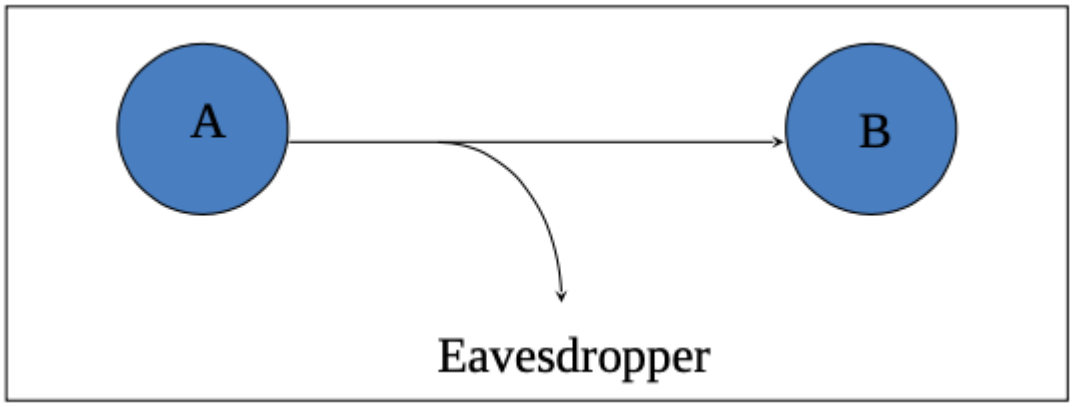
Security attacks

Eavesdropping - message interception (attack on confidentiality):

Eavesdropping is a type of attack in which an attacker listens in on communication between two parties without their knowledge or consent. This can be done in a number of ways, such as:

- **Wiretapping:** Physically accessing a telephone or network cable and listening in on the communication.
- **Sniffing:** Using software or specialized hardware to capture and analyze network traffic.

Eavesdropping can be prevented by using encryption to protect communication, implementing secure protocols for communication, and monitoring network traffic for signs of eavesdropping.



Integrity Attack - Tampering

An integrity attack is a type of attack that aims to alter, corrupt, or destroy data in a system or network without authorization. This can include:

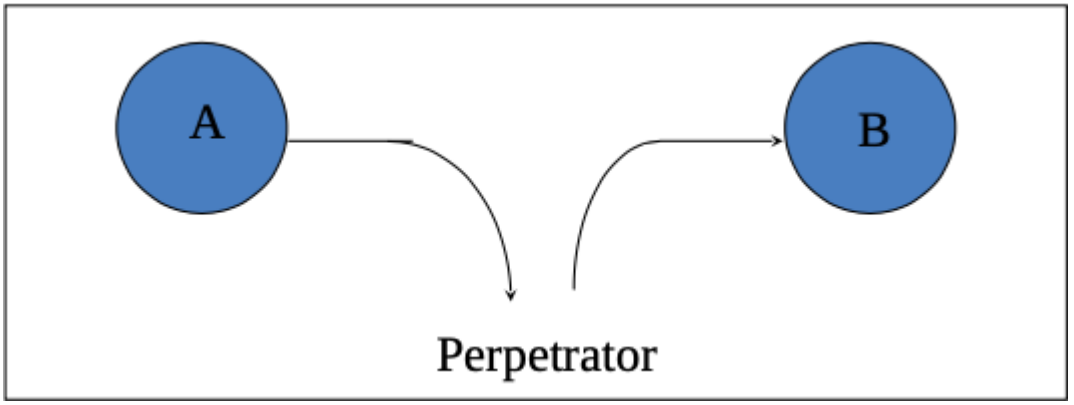
- **Data tampering:** Modifying data in a system or network without authorization. This can include changing values in a database, altering files on a server, or modifying network packets.

- **Data injection:** Inserting malicious or unauthorized data into a system or network. This can include inserting malware, inserting false information into a database, or injecting packets into a network.
- **Data deletion:** Deleting data from a system or network without authorization. This can include deleting files, wiping a hard drive, or removing data from a database.

An integrity attack can have serious consequences, such as loss of data, system downtime, or damage to the reputation of an organization.

To prevent integrity attacks, it is important to implement security controls such as access controls, data backups, network segmentation, and intrusion detection systems. Additionally, it is important to regularly monitor systems and networks for signs of integrity attacks, such as unusual changes in data or unusual network activity.

Integrity attacks can be mitigated by using cryptographic methods such as hash functions and digital signatures, these methods can ensure that the data has not been tampered with and can detect any modification on the data.



Authenticity Attack - Fabrication

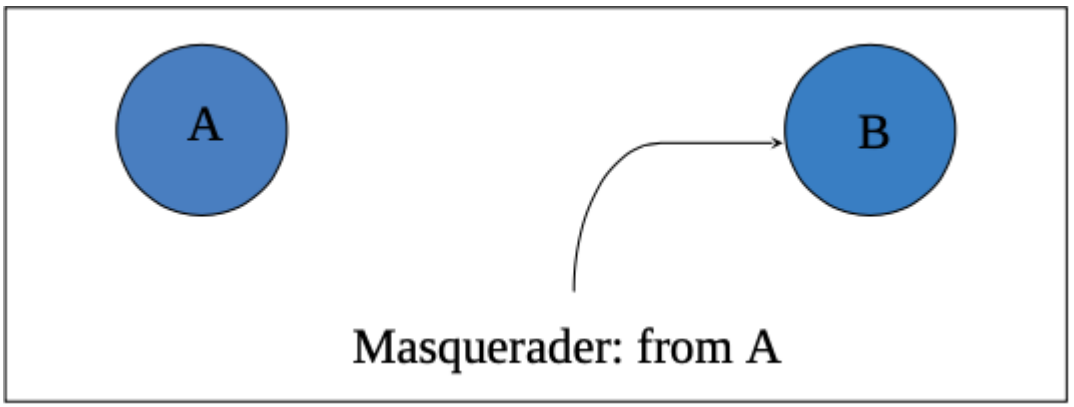
An authenticity attack is a type of attack in which an attacker attempts to impersonate an authorized user or device in order to gain access to a system or network. This can include:

- **Spoofing:** Impersonating a legitimate user or device by using their identifying information, such as an IP address, MAC address, or username.
- **Phishing:** Attempting to trick users into providing sensitive information, such as login credentials, by disguising as a trustworthy entity.
- **Man-in-the-Middle (MitM) attack:** Intercepting and potentially modifying communication between two parties without their knowledge.
- **Session hijacking:** Taking over an active session, such as a web session, by intercepting and using session cookies or tokens.

An authenticity attack can allow an attacker to gain unauthorized access to sensitive information or perform unauthorized actions, such as transferring funds or changing configurations.

Authenticity attacks can be prevented by implementing strong authentication methods such as multi-factor authentication, and by using encryption to protect communication. Additionally, it is important to be aware of the potential risks of authenticity attacks and to be cautious when providing personal information or clicking on links.

Furthermore, intrusion detection systems and firewalls can also help detect and prevent authenticity attacks by monitoring network traffic for signs of spoofing or phishing attempts. Additionally, regularly monitoring logs and network traffic can help detect and respond to authenticity attacks in progress.



Attack on availability

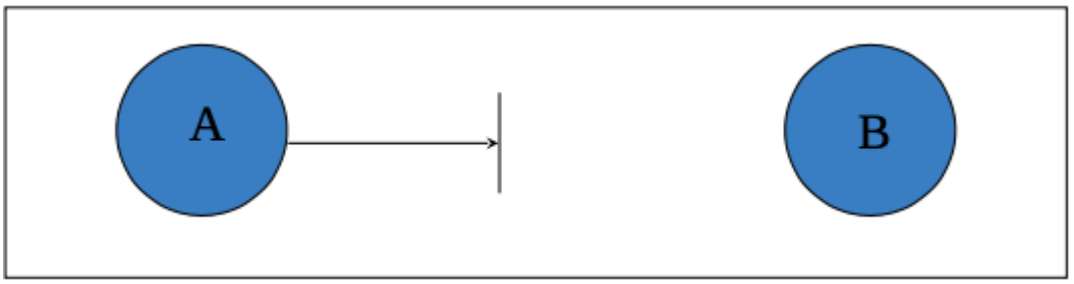
An availability attack is a type of attack that aims to make a system or network unavailable to authorized users. This can include:

- **Denial of Service (DoS) attack:** Attempting to make a system or network unavailable by overwhelming it with traffic or other means.
- **Distributed Denial of Service (DDoS) attack:** Similar to a DoS attack, but uses a network of compromised systems to launch the attack.
- **Ransomware:** Malware that encrypts a system's files and demands a ransom to be paid in order to restore access to the files.
- **Resource depletion:** Consuming resources such as memory, CPU, or disk space in a way that causes a system or network to become unavailable.
- **Physical damage:** Destroying or disabling physical infrastructure, such as servers or network equipment, to make a system or network unavailable.

An availability attack can have serious consequences, such as loss of revenue, loss of reputation, and damage to the operations of an organization.

To prevent availability attacks, it is important to implement security controls such as firewalls, intrusion detection systems, and load balancers. Additionally, it is important to regularly monitor systems and networks for signs of availability attacks, such as unusual traffic patterns or resource utilization.

Implementing a disaster recovery plan and implementing redundancy in the infrastructure can also help to mitigate the impact of availability attacks and allow an organization to quickly restore service in case of an attack.



Cryptography

Cryptographers invent secret codes to attempt to hide messages from unauthorized observers.

Encryption

Encryption is a method of converting plaintext (readable data) into ciphertext (unreadable data) using a mathematical algorithm, called a cipher. The process of converting plaintext to ciphertext is called encryption and the process of converting ciphertext back to plaintext is called decryption.

There are many different types of encryption algorithms, each with different strengths and weaknesses. Some common types of encryption include:

1. Symmetric key encryption: Uses the same key for both encryption and decryption. Examples include AES and DES.
2. Asymmetric key encryption: Uses a pair of keys, one for encryption and one for decryption. Examples include RSA and ECC.
3. Hashing: A one-way function that converts plaintext into a fixed-length output called a hash. Examples include SHA and MD5.

Encryption is used to protect sensitive information from unauthorized access or modification. Encryption can also be used to protect the privacy of communication, such as in email and instant messaging, as well as to secure online transactions, such as online banking and e-commerce.

Encryption can be applied in different ways, such as:

1. File-level encryption: Encrypts individual files or directories, such as encrypting a sensitive document.
2. Full-disk encryption: Encrypts an entire storage device, such as a hard drive or USB flash drive.
3. Network encryption: Encrypts communication sent over a network, such as SSL/TLS encryption used to secure internet traffic.
4. Database encryption: Encrypts data stored in a database, such as encrypting sensitive customer information.
5. Cloud encryption: Encrypts data stored in the cloud, such as encrypting files stored in a cloud-based storage service.

It's important to note that encryption is not foolproof and can be broken through various methods like brute force attacks, side-channel attacks, and other advanced techniques. Therefore, it's important to use strong encryption algorithms, keep encryption keys secure, and regularly update encryption methods to ensure data security.

Cryptography is not the same as hashing

Hashing

Hashing is a one-way function that converts plaintext, such as a password, into a fixed-length output called a hash. Hashes are often used to store passwords in a secure way because they cannot be easily converted back into plaintext.

There are many different types of hashing algorithms, each with different strengths and weaknesses. Some common types of hashing algorithms include:

1. MD5: A widely used hashing algorithm that produces a 128-bit hash. It is considered to be less secure because it is vulnerable to collisions and preimage attacks.
2. SHA-1: A widely used hashing algorithm that produces a 160-bit hash. It is considered to be less secure than newer algorithms because it is also vulnerable to collisions and preimage attacks.
3. SHA-2: A family of hashing algorithms that includes SHA-256 and SHA-512, which produce 256-bit and 512-bit hashes respectively. These algorithms are considered to be more secure than MD5 and SHA-1.

4. bcrypt: A key-derivation function that is specifically designed for hashing passwords. It is considered to be very secure because it is slow and computationally expensive, making it resistant to brute force attacks.
5. scrypt: A key-derivation function that is similar to bcrypt but also includes a memory-hard function, making it resistant to specialized hardware attacks.

Hashing is used to protect sensitive information such as passwords, encryption keys, and digital signatures. In many systems, the hashed password is stored instead of the plaintext password. When a user attempts to log in, the system calculates the hash of the entered password and compares it to the stored hash. If the hashes match, the entered password is correct.

It's important to note that hash functions are not designed to encrypt data, they are designed to produce a unique representation of the input data, called the digest, that can be used for comparison or indexing purposes. Additionally, it's important to use a secure and modern hashing algorithm, and also to use a salt, which is random data that is added to the input data before it is hashed, this makes it harder for attackers to use precomputed tables of hashes (rainbow tables) to crack the password.

- Hashing is the practice of transforming data to a unique (ish) value of fixed length
- In hashing it is impossible to reverse engineer (Though usually we don’t need to)
- Cryptography is the practice of altering the content of a message in some way to obscure its meaning
- Cryptogaphy it is impossible to make something that cannot be reverse engineered (Though thats actually what we want)

Three types of functions

Cryptographic hash Functions

- Zero keys
- Not sufficiently secure
- Very quick

Secret-key functions (Symmetric key)

- One key
- Very secure, very difficult to distribute
- Quick, compared to Public Key

Public-key functions (Asymmetric key)

- Two keys
- Very secure, easy to distribute
- Very slow

Salt

A salt is a random data that is added to the input data, such as a password, before it is hashed. The salt is then stored along with the hashed password, and is used during the login process to verify the entered password. The purpose of a salt is to make it harder for an attacker to use precomputed tables of hashes, called rainbow tables, to crack the password.

When a password is hashed without a salt, an attacker can precompute the hash of every possible password and store it in a lookup table. Then, when the attacker gets access to a hashed password, they can look it up in the table and find the corresponding plaintext password. However, if the password is hashed with a unique salt, the attacker would need to precompute a new lookup table for every password, which is a much more computationally expensive task.

A salt is typically a random value that is generated when a password is first hashed, it can be a string of random characters or a random number, and it should be unique for each user. It is important that the salt is stored in a secure location, such as in a separate database table, and that it is transmitted in a secure way, such as over an encrypted connection.

It is also important to use a different salt for each password and to not use a fixed or commonly known salt, otherwise it will not add much security and could be easily cracked.

In summary, the use of a salt when hashing passwords makes it much more difficult for an attacker to use precomputed tables to crack the password, thus adding an extra layer of security to the hashed password and making it more difficult for an attacker to gain access to the plaintext password.

VPN (Virtual Private Networks)

A Virtual Private Network (VPN) is a technology that creates a secure and encrypted connection over the internet. The connection is called a "tunnel," and it allows users to securely access resources on a private network, such as a corporate network, while they are connected to the internet.

VPNs can be used in a number of ways, including:

1. Remote access: Allows employees to access a company's private network from a remote location, such as from home or while traveling.
2. Site-to-site connectivity: Connects two or more private networks together, such as connecting a branch office to a corporate network.
3. Secure internet browsing: Allows users to securely access the internet from a public Wi-Fi or other untrusted network.

VPNs use various protocols such as PPTP, L2TP, SSTP, IKEv2, and OpenVPN, to establish secure and encrypted connections. These protocols use different methods of encryption, authentication, and tunneling to create the secure connection.

When a user connects to a VPN, their internet traffic is routed through the VPN's server, which acts as a middleman. This allows the VPN server to encrypt the traffic and protect it from being intercepted by an attacker.

institutions often want private networks for security.

- costly: separate routers, links, DNS infrastructure.

VPN: institution’s inter-office traffic is sent over public Internet instead

- encrypted before entering public Internet
- logically separate from other traffic

VPN Eksamensopgave

To calculate the distribution of bits in a virtual address, you need to know the page size and the size of the TLB.

In this system, the page size is 64 bytes and the TLB is 3-way set associative with 4 sets and 12 total entries.

Assuming that the page size is 2^6 bytes (=64), then 6 bits are used to indicate the offset within a page.

The remaining bits of the virtual address can be split up between the page number and the TLB set number.

The number of bits for the page number can be calculated by:

Virtual Address size - offset bits = 13 bits - 6 bits = 7 bits

The number of bits for the TLB set number can be calculated by:

log2(Number of sets) = log2(4) = 2 bits

So the distribution of bits in the virtual address is:

- 2 bits for the TLB set number (index) the first two bits in the VPN
- TLB tag is found by the remaining bits in the VPN (without the index)
- 7 bits for the page number (VPN)

- 6 bits for the offset within a page (use this if there is a physical address as the first 6 bits)

If the TLB **is not 3-way set associative**, the number of sets and the number of entries in the TLB would be different, and therefore the number of bits needed for the TLB set number would also be different.

For example, if the TLB is fully associative, meaning there is only 1 set, then no bits would be required for the TLB set number, as all entries are in the same set.

If the TLB is 2-way set associative, then only 1 bit would be required to identify the set number.

So the actual distribution of bits in the virtual address would depend on the associativity of the TLB and the number of sets available in it.

In general, the calculation of the distribution of bits in the virtual address would be:

- log2(Number of sets) bits for the TLB set number
- Virtual Address size - offset bits - log2(Number of sets) bits for the page number
- offset bits for the offset within a page

This way, you can adjust the calculations according to the associativity of the TLB.

FØrst skal man regne ud hvor mange bit hver kategori skal have i den virtuelle adresse. Derfor kan man opdele adressen i VPO (de første 5 bits) og VPN (de resterende bits). I VPN kan man finde index og tagget. I indexet kigger man på de første 2 bits af VPN og laver dem til Hex (da 2^2 =4), tagget er så resten af VPN. I første tabel skal man på index “rækken” (hvis index er 2 kigger man på set 2 i første tabel) Og ser om man kan finde tagget. Hvis man finder index værdien der så er der TLB hit og page HIT (N i page fault!!) ingen ting i PPN. Hvis man ikke finder den sammenligner man VPN i anden tabel også er der page hit hvis den er der. Hvis man får et hit så aflæser man PPN, også kan man finde den physical adress. Det gør man ved at kopiere VPO ned i starten og så skriver man PPN ind som bit i det næste stykke, også 0 for resten.

Operational Security

Operational Security (OPSEC) is a process of identifying and protecting critical information and assets in an organization. It involves identifying the critical information, analyzing how it could be compromised, and implementing security measures to protect it. The goal of OPSEC is to protect an organization's sensitive information and assets from unauthorized disclosure, disruption, or destruction.

OPSEC includes a range of activities, including:

1. Identifying critical information: Determining what information is critical to the organization and needs to be protected.
2. Analyzing threats: Identifying the potential threats to the critical information and how they could be used to harm the organization.
3. Identifying vulnerabilities: Identifying the weaknesses in the organization's current security measures.
4. Implementing countermeasures: Implementing security measures to protect the critical information and assets from the identified threats and vulnerabilities.
5. Continuously monitoring and assessing: Continuously monitoring the security of the organization and assessing the effectiveness of the security measures in place.

OPSEC is a process that is ongoing and requires regular review and updates to ensure that it is effective in protecting the organization's critical information and assets. It involves not only the IT and security teams but also the entire organization, as all employees have a role in identifying and protecting sensitive information.

Some examples of the steps that can be taken to improve operational security include:

- Implementing access controls to limit access to sensitive information
- Regularly reviewing and updating security policies and procedures
- Ensuring employees are aware of the risks and their role in protecting sensitive information
- Conducting regular security audits and vulnerability assessments
- Ensuring that systems and software are kept up to date with the latest security patches.

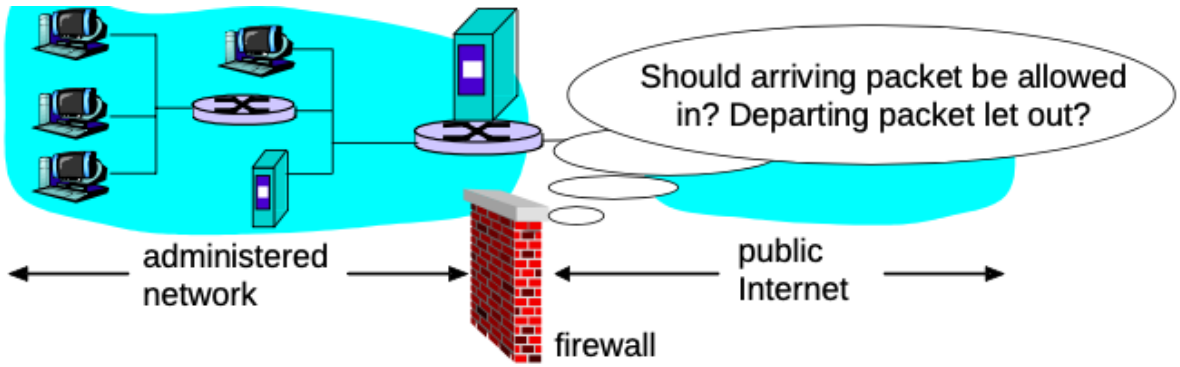
By implementing effective operational security measures, organizations can protect their sensitive information and assets from unauthorized access, and reduce the risk of data breaches, theft, and other security incidents.

Two instances today

- Firewalls
- DNS Security

Firewalls

Isolates internal net from larger Internet, allowing some packets to pass, blocking others.



Firewall filters packet-by-packet, based on:

Packet-by-packet: refers to the process of transmitting data in a network in which each packet is sent independently, one at a time, rather than as a single, continuous stream of data. In packet-by-packet transmission, each packet is treated as a separate unit of data, and it is the responsibility of the network to ensure that all packets are properly received and reassembled at the destination.

- Source/Dest IP address; Source/Dest TCP/UDP port numbers
- TCP SYN and ACK bits; ICMP message type
- Deep packet inspection on packet contents (DPI)

Packet Filtering Examples

Block all packets with IP protocol field = 17 and with either source or dest port = 23.

- All incoming and outgoing UDP flows blocked
- All Telnet connections are blocked

Block inbound TCP packets with SYN but no ACK

- Prevents external clients from making TCP connections with internal clients
- But allows internal clients to connect to outside

Block all packets with TCP port of Quake

Peers to Peers

Speeding Up the File Distribution

Increase the server upload rate

- Higher link bandwidth at the server
- Multiple servers, each with their own link

Alternative: have the receivers help

- Receivers get a copy of the data
- and redistribute to other receivers
- To reduce the burden on the server

Peer-to-peer Networks: BitTorrent

BitTorrent history

- 2002: B. Cohen debuted BitTorrent

Emphasis on efficient fetching, not searching

- Distribute same file to many peers
- Single publisher, many downloaders

Preventing free-loading

- Incentives for peers to contribute

BitTorrent: Tracker

Infrastructure node

- Keeps track of peers participating in the torrent
- Peers register with the tracker when it arrives

Tracker selects peers for downloading

- Returns a random set of peer IP addresses
- So the new peer knows who to contact for data

Can have “trackerless” system

- Using distributed hash tables (DHTs)

BitTorrent: Chunk Request Order

Which chunks to request?

- Could download in order
- Like an HTTP client does

Problem: many peers have the early chunks

- Peers have little to share with each other
- Limiting the scalability of the system

Problem: eventually nobody has rare chunks

- E.g., the chunks need the end of the file
- Limiting the ability to complete a download

Solutions: random selection and rarest first

BitTorrent: Rarest Chunk First

Which chunks to request first?

- Chunk with fewest available copies (i.e., rarest chunk)

Benefits to the peer

- Avoid starvation when some peers depart

Benefits to the system

- Avoid starvation across all peers wanting a file
- Balance load by equalizing # of copies of chunks

Bit-Torrent: Preventing Free-Riding

Peer has limited upload bandwidth

- And must share it among multiple peers
- Tit-for-tat: favor neighbors uploading at highest rate

Rewarding the top four neighbors

- Measure download bit rates from each neighbor • Reciprocate by sending to the top four peers

Optimistic unchoking

- Randomly try a new neighbor every 30 seconds
- So new neighbor has a chance to be a better partner • Compatible peers find each other

Server Performance BOH 12.5-12.7

Synchronizing Threads with Semaphores

Semaphores means synchronizing different threads based on a special type of variable.

Semphores - A special type of variable s that is a global variable with a nonnegative integer value that can only be manipulated by two special operations, called P and V;

$P(s)$: If s is nonzero, then P decrements s and returns immediately. If s is zero, then suspend the thread until s becomes nonzero and the thread is restarted by a V operation. After restarting, the P operation decrements s and returns control to the caller.

$V(s)$: The V operation increments s by 1. If there are any threads blocked at a P operation waiting for s to become nonzero, then the V operation restarts exactly one of these threads, which then completes its P operation by decrementing s.

Semaphore is a synchronization technique where we can **control number of threads to access a resource**. In lock/Mutex, only one thread can access resources at a time. But Semaphore allows multiple threads to access the same resource at a time. We can limit the number of threads that can access the same resources.

We want safe trajectories all the time and this we can do by synchronizing the threads.

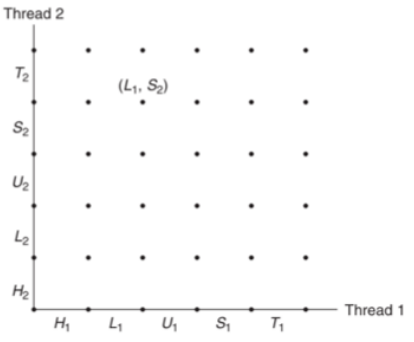
Progress graph

A progress graph models the execution of n concurrent threads as a trajectory (bane) through an n-dimensional Cartesian space.

The origin of the graph corresponds to the initial state where none of the threads has yet completed an instruction.

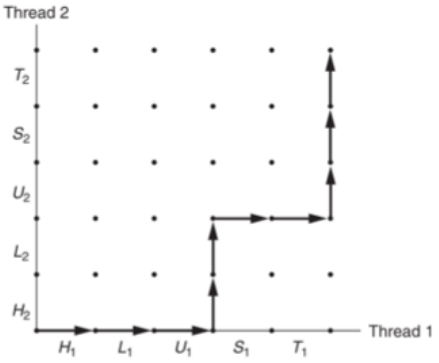
A progress graph models instructions execution as a transition from one state to another. A transition is represented as a directed edge from one point to an adjacent point. Only transitions that move to the right or up is legal, this is because a program never runs backwards for example, and two transitions instructions can´t complete at the same time.

Figure 12.19
Progress graph for the first loop iteration of `badcnt.c`.



The execution history of a program is modeled as a trajectory through the state space.

Figure 12.20
An example trajectory.



this corresponds to writting
$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2$

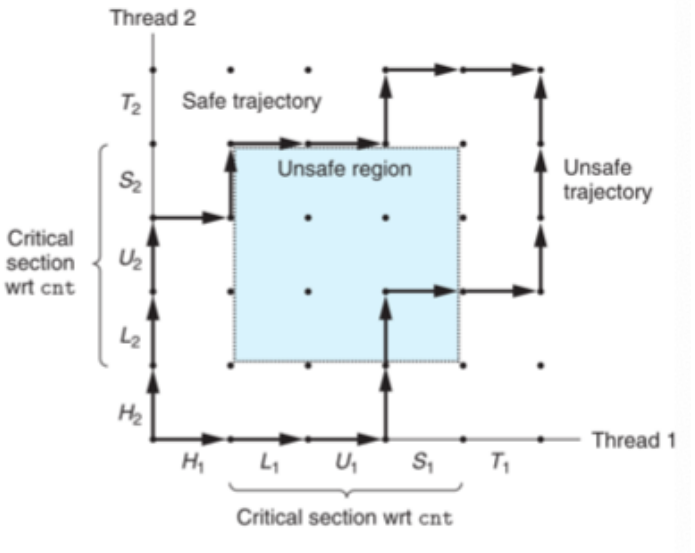
Mutual exclusion: We want to ensure that each thread has mutually exclusive access to the shared variable while it is executing the instructions in its critical section.

Unsafe region: The intersection of two critical sections defines a region of the state space.

Safe Trajectory: A trajectory that skirts the unsafe region.

Unsafe Trajectory: A trajectory that touches any part of the unsafe region.

Figure 12.21
Safe and unsafe trajectories. The intersection of the critical regions forms an unsafe region. Trajectories that skirt the unsafe region correctly update the counter variable.



Computer Architecture

Digital logic

Digital logic is a branch of electrical engineering that deals with the design and analysis of digital systems, such as computer processors, memory, and other digital circuits. It is based on the use of digital signals, which have two discrete values, typically represented as "0" and "1".

Digital logic circuits are built using basic logic gates such as AND, OR, NOT, NAND, NOR, XOR and XNOR. These gates take one or more input signals and produce a single output signal, based on a specific logical function. The output of a logic gate is determined by the input values, which are either 0 or 1.

Combinations of these basic gates can be used to create more complex circuits such as flip-flops, counters, and finite state machines. These circuits can be used to control the flow of data in a digital system and to perform specific operations such as data storage and retrieval.

Digital logic circuits can also be represented using Boolean algebra, which is a mathematical system that allows the analysis and manipulation of logical expressions. Boolean algebra is used to simplify and optimize the design of digital circuits.

Digital logic is the foundation of modern digital electronics and computer technology, and it is used in a wide range of applications such as computers, mobile phones, televisions, and industrial control systems.

It's important to note that digital logic is different from analog electronics, which deals with signals that can take any value within a range, instead of only two discrete values.

Gates, Truth Tables, and Logic Equations COD A.2

Digital electronics operate with only two voltage levels of interest: a high voltage and a low voltage. All other voltage values are temporary and occur while transitioning between the values.

Asserted signal: A signal that is (logically) true, or 1.

Deasserted signal: A signal that is (logically) false, or 0.

Logic blocks are categorized as one of two types, depending on whether they contain memory.

combinational logic: A logic system whose blocks do not contain memory and hence compute the same output given the same input.

sequential logic: A group of logic elements that contain memory and hence whose value depends on the inputs as well as the current contents of the memory.

Truth Tables

Because a combinational logic block contains no memory, it can be completely specified by defining the values of the outputs for each possible set of input values. Such a description is normally given as a *truth table*. For a logic block with *n* inputs, there are 2*n* entries in the truth table, since there are that many possible combinations of input values. Each entry specifies the value of all the outputs for that particular input combination.

Truth Tables

Consider a logic function with three inputs, A, B, and C, and three outputs, D, E, and F. The function is defined as follows: D is true if at least one input is true, E is true if exactly two inputs are true, and F is true only if all three inputs are true. Show the truth table for this function.

The truth table will contain $2^3 = 8$ entries. Here it is:

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

Boolean Algebra

Another approach is to express the logic function with logic equations

In Boolean algebra, all the variables have the values 0 or 1 and, in typical formulations, there are three operators:

- The OR operator is written as +, as in $A + B$. The result of an OR operator is 1 if either of the variables is 1. The OR operation is also called a *logical sum*, since its result is 1 if either operand is 1.

- The AND operator is written as $\cdot$, as in $A \cdot B$. The result of an AND operator is 1 only if both inputs are 1. The AND operator is also called *logical product*, since its result is 1 only if both operands are 1.

- The unary operator NOT is written as A. The result of a NOT operator is 1 only if the input is 0. Applying the operator NOT to a logical value results in an inversion or negation of the value (i.e., if the input is 0 the output is 1, and vice versa).

There are several laws of Boolean algebra that are helpful in manipulating logic equations.

- Identity law: $A + 0 = A$ and $A \cdot 1 = A$
- Zero and One laws: $A + 1 = 1$ and $A \cdot 0 = 0$
- Inverse laws: $A + \overline{A} = 1$ and $A \cdot \overline{A} = 0$
- Commutative laws: $A + B = B + A$ and $A \cdot B = B \cdot A$
- Associative laws: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive laws: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$

Any set of logic functions can be written as a series of equations with an output on the left-hand side of each equation and a formula consisting of variables and the three operators above on the right-hand side.

Logic Equations

Show the logic equations for the logic functions, D, E, and F, described in the previous example.

Here's the equation for D:

$$D = A + B + C$$

F is equally simple:

$$F = A \cdot B \cdot C$$

E is a little tricky. Think of it in two parts: what must be true for E to be true (two of the three inputs must be true), and what cannot be true (all three cannot be true). Thus we can write E as

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

We can also derive E by realizing that E is true only if exactly two of the inputs are true. Then we can write E as an OR of the three possible terms that have two true inputs and one false input:

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

Proving that these two expressions are equivalent is explored in the exercises.

gate: A device that implements basic logic functions, such as AND or OR.

For example, an AND gate implements the AND function, and an OR gate implements the OR function. Since both AND and OR are commutative and associative, an AND or an OR gate can have multiple inputs, with the output equal to the AND or OR of all the inputs. The logical function NOT is implemented with an inverter that always has a single input. The standard representation of these three logic building blocks is shown in Figure A.2.1.

FIGURE A.2.1 Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right. The signals to the left of each symbol are the inputs, while the output appears on the right. The AND and OR gates both have two inputs. Inverters have a single input.

Rather than draw inverters explicitly, a common practice is to add “bubbles” to the inputs or outputs of a gate to cause the logic value on that input line or output line to be inverted. For example, Figure A.2.2 shows the logic diagram for the function $A\ B$, using explicit inverters on the left and bubbled inputs and outputs on the right.

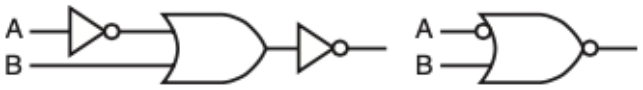


FIGURE A.2.2 Logic gate implementation of $\overline{\overline{A} + B}$ using explicit inverts on the left and bubbled inputs and outputs on the right. This logic function can be simplified to $A \cdot \overline{B}$ or in Verilog, `A & ~ B`.

Inverted gates Its main function is to **invert the input signal applied**. If the applied input is low then the output becomes high and vice versa.

The two common interesting gates, NOR and NAND gates are called *universal*, since any logic function can be built using this one gate type.

NOR gate An inverted OR gate.

NAND gate An inverted AND gate.

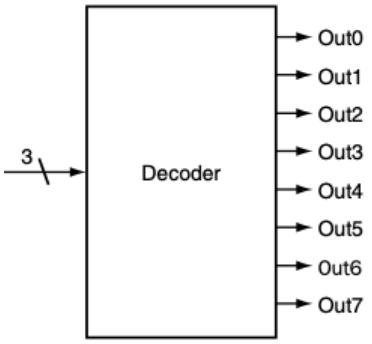
Combinational Logic COD A.3

Decoders

One logic block we will use in building larger components is a decoder.

decoder: A logic block that has an *n*-bit input and 2*n* outputs, where only one output is asserted for each input combination. The inverted decoder is called an **encoder.**

This decoder translates the *n*-bit input into a signal that corresponds to the binary value of the *n*-bit input. The outputs are thus usually numbered, say, Out0, Out1, ..., Out2*n* −1. If the value of the input is *i*, then Out*i* will be true and all other outputs will be false.



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

FIGURE A.3.1 A 3-bit decoder has three inputs, called 12, 11, and 10, and $2^3 = 8$ outputs, called Out0 to Out7. Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

Multiplexors

One basic logic function that is often used is called multiplexors. A multiplexor might more properly be called a *selector*, since its output is one of the inputs that is selected by a control.

Constructing a basic arithmetic logic unit COD A.5

The **arithmetic logic unit (ALU)** is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR.

Clocks COD A.7

Clocks are needed in sequential logic to decide when an element that contains state should be updated. A clock is simply a free-running signal with a fixed *cycle time*; the *clock frequency* is simply the inverse of the cycle time.

edge-triggered clocking: A clocking scheme in which all state changes occur on a clock edge.

clocking methodology: The approach used to determine when data are valid and stable relative to the clock.

state element: A memory element.

synchronous system: A memory system that employs clocks and where data signals are read only when the clock indicates that the signal values are stable.

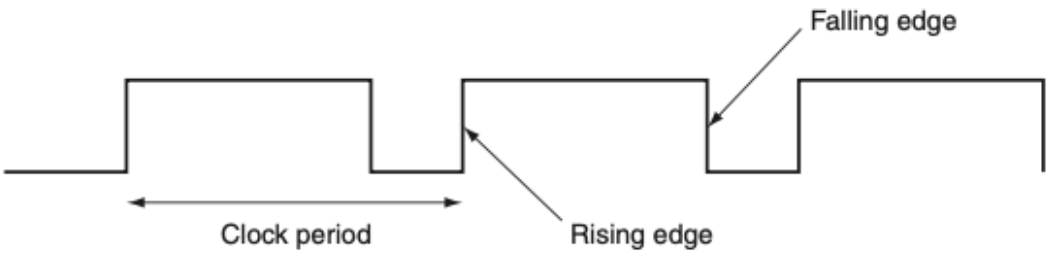


FIGURE A.7.1 A clock signal oscillates between high and low values. The clock period is the time for one full cycle. In an edge-triggered design, either the rising or falling edge of the clock is active and causes state to be changed.

As shown in Figure A.7.1, the *clock cycle time* or *clock period* is divided into two portions: when the clock is high and when the clock is low. In this text, we use only **edge-triggered clocking**. This means that all state changes occur on a clock edge. We use an edge-triggered methodology because it is simpler to explain. Depending on the technology, it may or may not be the best choice for a **clocking methodology**.

Memory elements: Flip-Flops, Latches, and registers COD A.8

All memory elements store state: the output from any memory element depends both on the inputs and on the value that has been stored inside the memory element. Thus all logic blocks containing a memory element contain state and are sequential.

Flip-flop

flip-flop A memory element for which the output is equal to the value of the stored state inside the element and for which the internal state is changed only on a clock edge.

latch A memory element in which the output is equal to the value of the stored state inside the element and the state is changed whenever the appropriate inputs change and the clock is asserted.

D flip-flop A flip-flop with one data input that stores the value of that input signal in the internal memory when the clock edge occurs.

hold time The minimum time during which the input must be valid after the clock edge.

Register Files

One structure that is central to our datapath is a *register file*. A register file consists of a set of registers that can be read and written by supplying a register number to be accessed. A register file can be implemented with a decoder for each read or write port and an array of registers built from D flip-flops. Because reading a register does not change any state, we need only supply a register number as an input, and the only output will be the data contained in that register.

For writing a register we will need three inputs: a register number, the data to write, and a clock that controls the writing into the register.

Memory elements: SRAMs and DRAMs COD A.9

Registers and register files provide the basic building blocks for small memories, but larger amounts of memory are built using either **SRAMs (static random access memories)** or *DRAMs* (dynamic random access memories). We first discuss SRAMs, which are somewhat simpler, and then turn to DRAMs.

static random access memory (SRAM): A memory where data are stored statically (as in flip-flops) rather than dynamically (as in DRAM). SRAMs are faster than DRAMs, but less dense and more expensive per bit.

SRAMs

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write. SRAMs have a fixed access time to any datum, though the read and write access characteristics often differ.

DRAMs

In a static RAM (SRAM), the value stored in a cell is kept on a pair of inverting gates, and as long as power is applied, the value can be kept indefinitely. In a dynamic RAM (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit.

Logic Design Conventions COD 4.2

combinational element: An operational element, such as an AND gate or an ALU

state element: A memory element, such as a register or a memory.

control signal A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a *data signal*, which contains information that is operated on by a functional unit.

asserted The signal is logically high or true.

deasserted The signal is logically low or false.

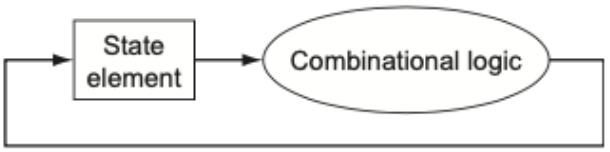


FIGURE 4.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within one clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and on structures like the one shown in this figure.

Building a datapath COD 4.3

A reasonable way to start a datapath design is to examine the major components required to execute each class of RISC-V instructions. Let's start at the top by looking at which **datapath elements** each instruction needs, and then work our way down through the levels of **abstraction**.

datapath element: A unit used to operate on or hold data within a processor. In the RISC-V implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

program counter: (PC) The register containing the address of the instruction in the program being executed.

Building the datapath:

- First we need a memory unit to store the instructions of a program and supply instructions given an address.
- Then we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU.

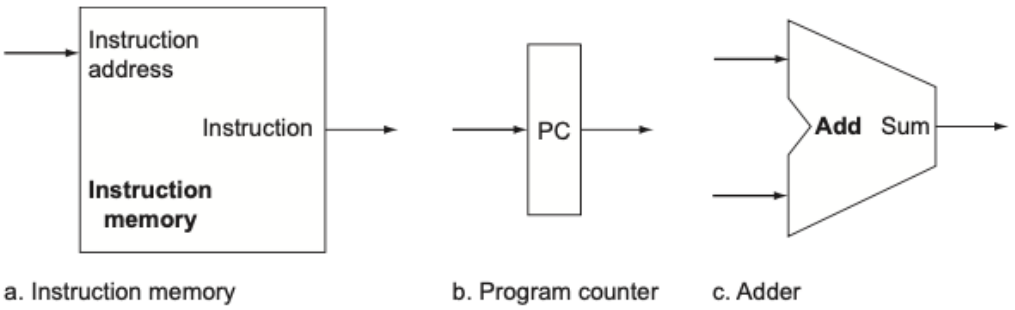


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

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

Creating a single datapath

Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation.

This simplest datapath will attempt to execute all instructions in one clock cycle. Thus, that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

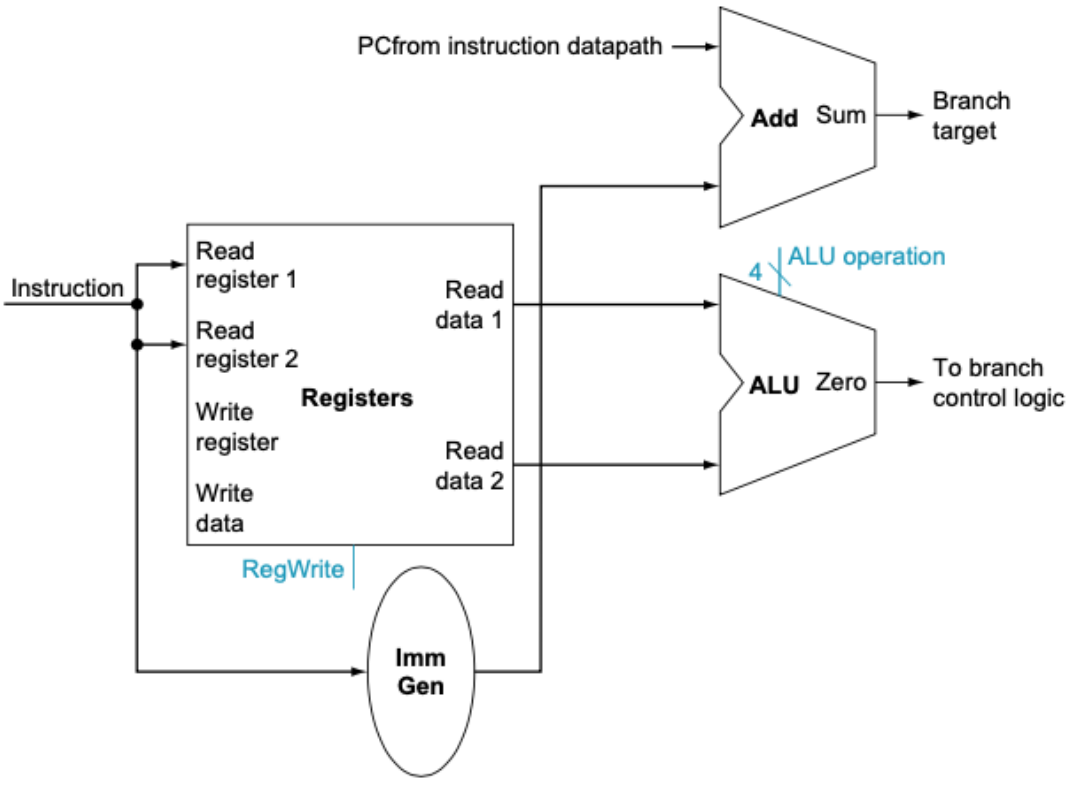


FIGURE 4.9 The portion of a datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and immediate (the branch displacement). Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

Example building a datapath

Building a Datapath

The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

- The arithmetic-logical instructions use the ALU, with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign- extended 12-bit offset field from the instruction.
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

Show how to build a datapath for the operational portion of the memory- reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

ANSWER

To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. Figure 4.10 shows the operational portion of the combined datapath.

A simple implementation scheme COD 4.4

In this section, we look at what might be thought of as a simple implementation of our RISC-V subset. We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers *load word* (lw), *store word* (sw), *branch if equal* (beq), and the arithmetic- logical instructions add, sub, and, and or.

The ALU control

The RISC-V ALU in Appendix A defines the four following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Depending on the instruction class, the ALU will need to perform one of these four functions. For load and store instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the four actions (AND, OR, add, or subtract), depending on the value of the 7-bit funct7 field (bits 31:25) and 3-bit funct3 field (bits 14:12) in the instruction.

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction. The instruction, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the funct7 or funct3 fields; in this case, we say that we "don't care" about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input. See Appendix A.

LÆS 4.4 GODT IGENNEM

Shared memory multithreading COD 6.1

As mentioned in chapter 1, energy has become the overriding issue for both microprocessors and data centers. Replacing large inefficient processors with many smaller, efficient processors can deliver better performance per Joule both in the large and in the small, if software can efficiently use them. Therefore, improved energy efficiency joins scalable performance in the case for multiprocessors.

This means that a multiprocessor would still be able to run when there is presence of a broken hardware it would just run with n-1 processor.

task-level parallelism or **process-level parallelism:** Utilizing multiple processors by running independent programs simultaneously.

parallel processing program: A single program that runs on multiple processors simultaneously.

cluster: A set of computers connected over a local area network that function as a single large multiprocessor.

multicore microprocessor: A microprocessor containing multiple processors (“cores”) in a single integrated circuit. Virtually all microprocessors today in desktops and servers are multicore.

shared memory multiprocessor (SMP): A parallel processor with a single physical address space.

The state of technology today means that programmers who care about performance must become parallel programmers.

Multicore and other shared memory multiprocessors COD 6.5

uniform memory access (UMA): A multiprocessor in which latency to any word in main memory is about the same no matter which processor requests the access.

nonuniform memory access (NUMA): A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

synchronization: The process of coordinating the behavior of two or more processes, which may be running on different processors.

lock: A synchronization device that allows access to data to only one processor at a time.

Example of a program that is a parallel programming system, which offers a portable, scalable, and simple programming model for shared memory multiprocessors. Its primary goal is to parallelize loops and perform reductions.

OpenMP: An API (Application Programmer Interface) for shared memory multiprocessing in C, C++, or Fortran that runs on UNIX and Microsoft platforms. It includes compiler directives, a library, and runtime directives.

Parallelism and memory hierarchy: Cache coherence COD 5.10

Given that a multicore multiprocessor means multiple processors on a single ship, these processors very likely share a common physical address space.

The first aspect, called *coherence*, defines *what values* can be returned by a read. The second aspect, called *consistency*, determines *when* a written value will be returned by a read.

Write serialization is when ensure that all writers to the same location (address) are seen in the identical order, which means that we see that first write first, and the second second and so on, and therefore only saves the latest written.

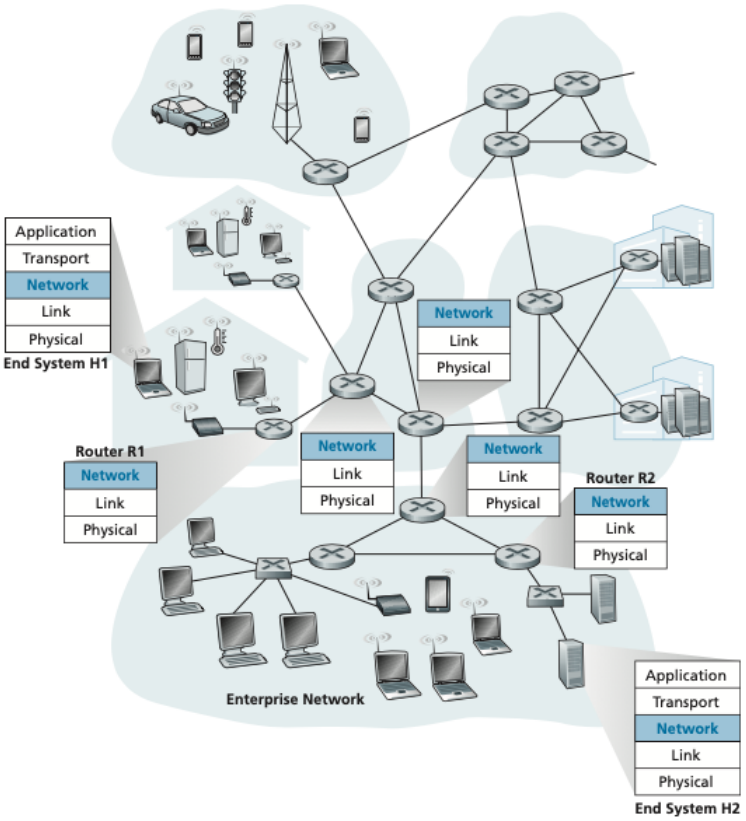
The network layer: data plane KR 4.1- 4.24

Forwarding and routing: the data and control planes

The primary role of the network layer is deceptively simple to move packets from a sending host to a receiving host. To do so, two important network-layer functions can be identified:

Forwarding: When a packet arrives at a router's input link, the router must move the packet to the appropriate output link. For example, a packet arriving from Host H1 to Router R1 in Figure 4.1 must be forwarded to the next router on a path to H2. As we will see, forwarding is but one function (albeit the most common and important one!) implemented in the data plane. In the more general case, which we'll cover in Section 4.4, a packet might also be blocked from exit- ing a router (for example, if the packet originated at a known malicious sending host, or if the packet were destined to a forbidden destination host), or might be duplicated and sent over multiple outgoing links.

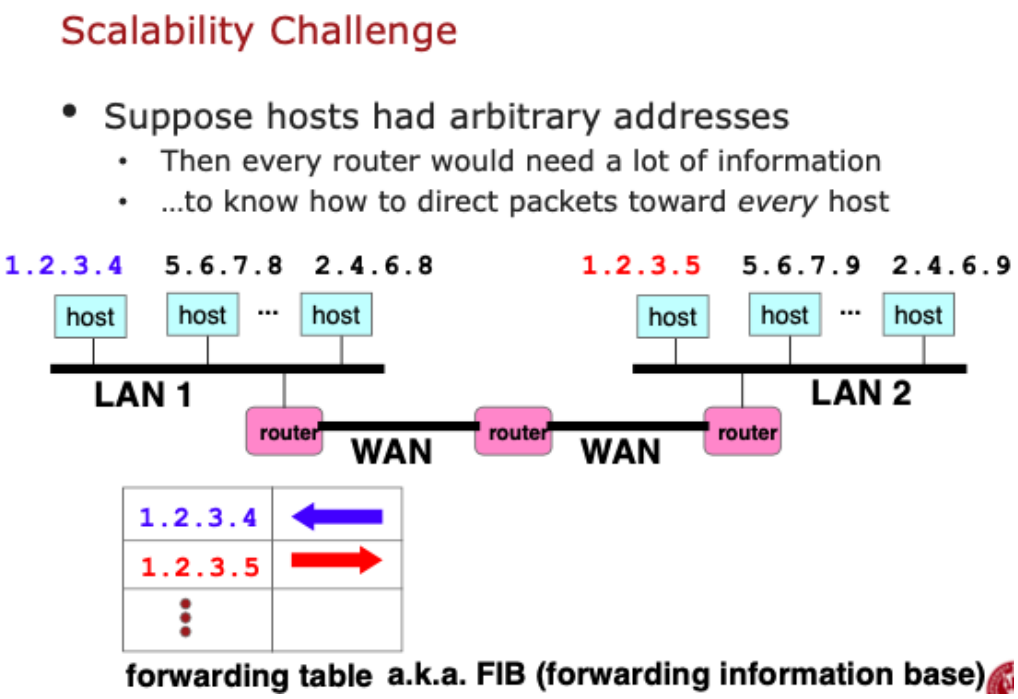
Routing: The network layer must determine the route or path taken by packets as they flow from a sender to a receiver. The algorithms that calculate these paths are referred to as **routing algorithms**. A routing algorithm would determine, for example, the path along which packets flow from H1 to H2 in Figure 4.1. Routing is implemented in the control plane of the network layer.



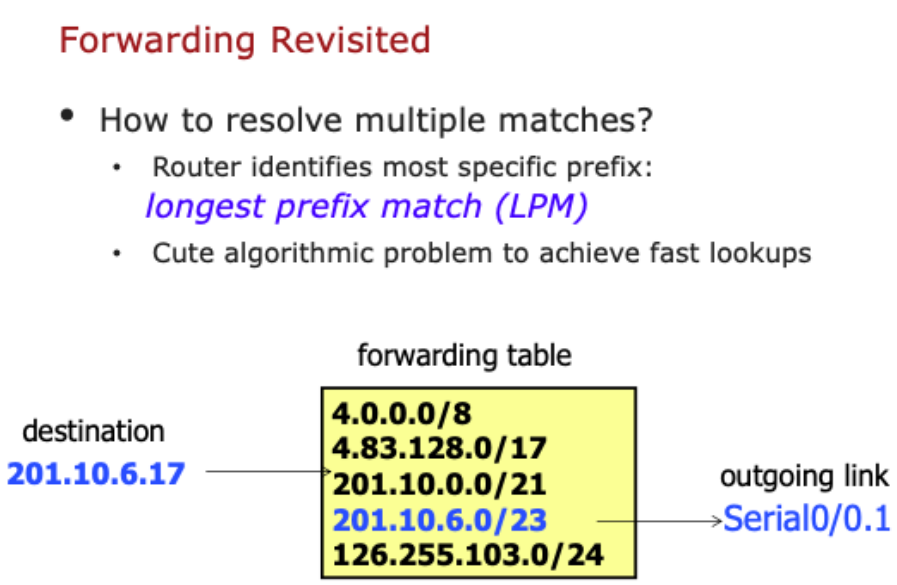
Forwarding refers to the router-local action of transferring a packet from an input link interface to the appropriate output link interface. Forwarding takes place at very short timescales (typically a few nanoseconds), and thus is typically implemented in hardware.

Routing refers to the network-wide process that determines the end-to-end paths that packets take from source to destination. Routing takes place on much longer timescales (typically seconds), and as we will see is often implemented in software. Using our driving analogy, consider the trip from Pennsylvania to Florida undertaken by our traveler back in Section 1.3.1. During this trip, our driver passes through many interchanges en route to Florida. We can think of forwarding as the process of getting through a single interchange: A car enters the interchange from one road and deter- mines which road it should take to leave the interchange. We can think of routing as the process of planning the trip from Pennsylvania to Florida: Before embarking on the trip, the driver has consulted a map and chosen one of many paths possible, with each path consisting of a series of road segments connected at interchanges.

Forwarding Table: is a key element in every network router. A router forwards a packet by examining the value of one or more fields in the arriving packet's header, and then using these header values to index into its forwarding table. The value stored in the forwarding table entry for those values indicates the outgoing link interface at that router to which that packet is to be forwarded.



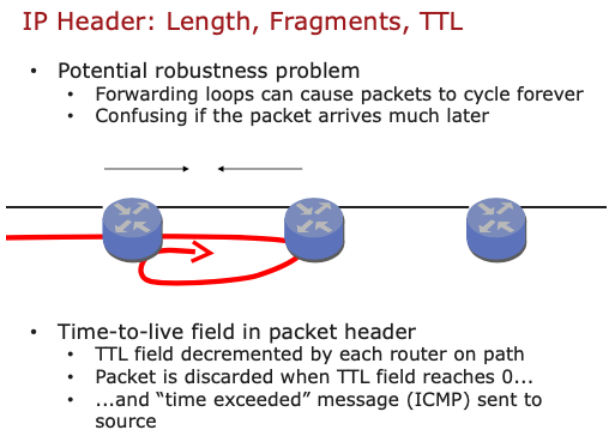
It can happen that a IP address from the destination is identical with two addresses in the forwarding table, here it would use the longest prefix match (LPM) and choose the IP address which is the longest. In the picture two IP addresses in the forwarding table are almost the same, but one is /21 and one is /23, then we chooses the /23 because its the longest prefix.



IP Header

IP packet structure (IPv4)

The packets may not be received in the right order if any of them is stuck in the forwarding loop, we want to prevent this from happening. Because it would be confusing.

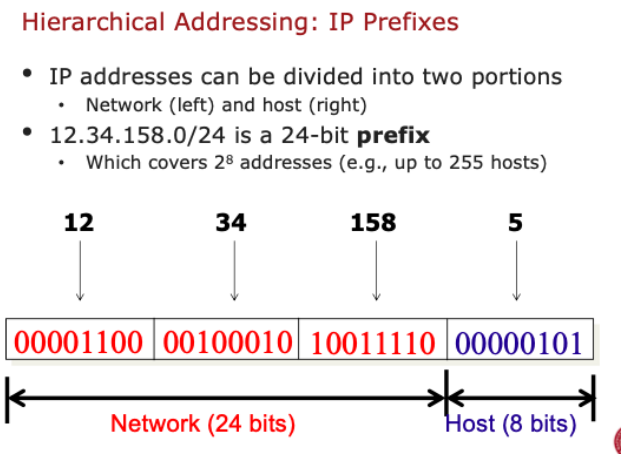


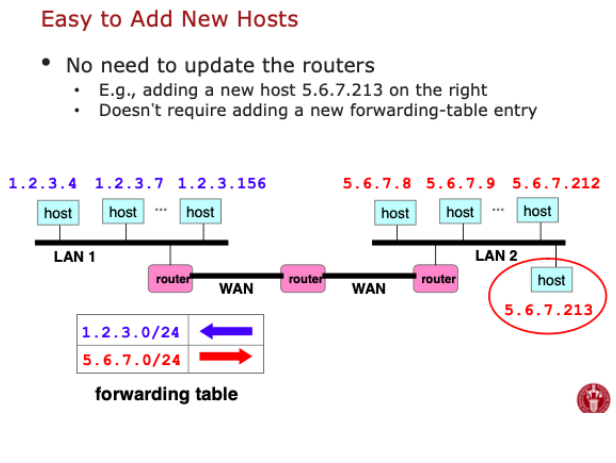
Time to live: Checks if there are any packets stuck in the forwarding loop, if it cycles too many times the TTL field reaches 0 and the packet is discarded.

Standard CS trick

Hierarchical addressing

If we have a scalability issue, we can split our network up into bits. This makes it easier for new hosts to connect to the network.

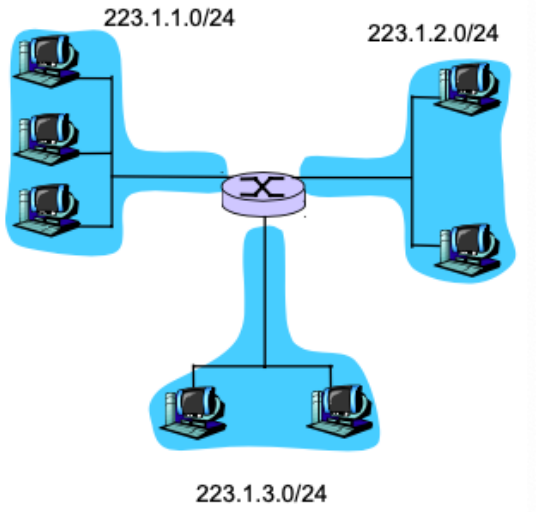




Subnets

Recipe: to determine the subnets, detach each interface from its host or router, creating islands of isolated networks

Each isolated network is called a subnet



Protocols

Dynamic Host Configuration Protocol (DHCP)

- End host learns how to send packets
- Learn IP address, DNS servers, and gateway

Address Resolution Protocol (ARP)

- Others learn how to send packets to the end host
- Learn mapping between IP address & interface address

Key ideas in both protocols

Broadcasting: When in doubt, shout!

- broadcast query to all hosts in the local-area-network
- … when you don't know how to identify the right one

Caching: Remember the past for a while

- store the information you learn to reduce overhead
- remember your own address and other host's addresses

Soft state: but eventually forget the past

- associate a time-to-live field with the information
- .. and either refresh or discard the information
- key for robustness in the face of unpredictable change

Control plane: The traditional approach

How is a routers forwarding table configured in the first place. This is a crucial issue, one that exposes the important interplay between forwarding (in data plane) and routing (in control plane).

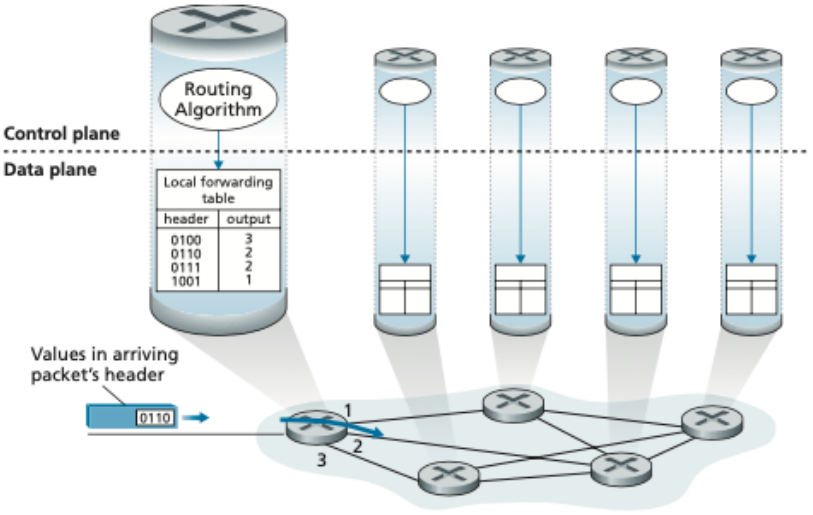


Figure 4.2 ♦ Routing algorithms determine values in forward tables

In Figure 4.2, the routing algorithm determines the contents of the routers' forwarding tables. In this example, a routing algorithm runs in each and every router and both forwarding and routing functions are contained within a router. As we'll see in Sections 5.3 and 5.4, the routing algorithm function in one router communicates with the routing algorithm function in other routers to compute the values for its forwarding table. How is this communication performed? By exchanging routing messages containing routing information according to a routing protocol!

Network layering: Routing

Assembly - C kode

Partition

This is a *leaf procedure*, meaning you can use caller-saves registers such as t0-t6 without worry.

```
        .text
partition:
        # array in a0
        # p in a1
        # r in a2

            slli t0, a1, 2     # right memory offset because int array
            add t0, t0, a0   # address to array[p]  :  add p offset (t0) to array (a0) address
            lw t0, 0(t0)       # array[p]/value  :  set t0 to the value from the address t0

            addi t1, a1, -1   # i = p - 1  :  i is in t1:
            mv t2, a2          # j = r  :  j is in t2
            j doloop1          # jump to doloop1

doloop1:
            addi t2, t2, -1        # j--
            slli t3, t2, 2          # right memory offset because int array
            add t3, t3, a0         # address to array[j]  :  add j offset (t3) to array (a0) address
            lw t4, 0(t3)           # array[j]/value  :  set t4 to the value from the address t3
            blt t0, t4, doloop1  # if pivot < array[j]
            j doloop2

doloop2:
            addi t1, t1, 1         # i++
            slli t5, t1, 2          # right memory offset because int array
            add t5, t5, a0         # address to array[i]  :  add i offset (t5) to array (a0) address
            lw t6, 0(t5)           # array[i]/value  :  set t6 to the value from the address t5
            blt t6, t0, doloop2  # if array[i] < pivot

            blt t1, t2, loopCode  # if (i < j)
            jal zero, return

loopCode:
            sw t4, 0(t5)    # array[i] adress = array[j] value
            sw t6, 0(t3)    # array[j] address = tmp = array[i] value
            j doloop1

return:
            addi t2, t2, 1    # j + 1
            mv a0, t2         # return j + 1 in a0
            jalr zero, ra, 0  # jump to ra
```

Partition in C

```c
int partition(int *array, int p, int r) {
  int pivot = array[p];
  int i=p-1;
  int j=r;

  while (1) {
    do { j--; } while (pivot < array[j]);
    do { i++; } while (array[i] < pivot);
    if (i < j) {
      int tmp = array[i];
      array[i] = array[j];
      array[j] = tmp;
    } else {
      return j+1;
    }
  }
}
```

Quicksort (its recursive)

```
quicksort:
    # a0 stores array
    # a1 stores start
    # a2 stores end

    addi sp, sp, -20   # makes space for 5 items on the stack
    sw ra, 16(sp)      # stores return address
    sw a2, 8(sp)       # stores "end"
    sw a1, 4(sp)       # stores "start"
    sw a0, 0(sp)       # stores array

    sub t0, a2, a1    # end - start
    li t1, 2          # 2
    blt t0, t1, stop  # (end - start) < 2

    jal ra, partition  # calls on partition

    sw a0, 12(sp)     # stores a0 (return value from partition) in 12(sp). 12(sp) is now q
    lw a0, 0(sp)      # restores argument a0 ("array") to original value
    lw a2, 12(sp)     # changes "end" to value of a0 (q)
```

```
        jal ra, quicksort  # quicksort(array, start, q)

        lw a1, 12(sp)      # changes "start" to value of a0 (q)
        lw a2, 8(sp)       # restores a2 ("end") to original value
        jal ra, quicksort  # quicksort(array, q, end)

stop:
        lw ra, 16(sp)
        addi sp, sp, 20   # adjust stack pointer to pop 5 items
        jalr zero, ra, 0    # jump to return adress
```

Quicksort in C

```c
void quicksort(int *array, int start, int end) {
  if (end-start < 2) {
    return;
  }

  int q = partition(array, start, end);
  quicksort(array, start, q);
  quicksort(array, q, end);
}
```

Insertionsort (in exam 21/22)

```
START:
     addi     sp, sp, -8
     sw       ra, 0(sp)
     mv       t0, 1
     jal      r0, L2
L1:
     slli     t4, t2, 3
     add      ra, a0, t4
     lw       t1, 8(ra)
     addi     t0, t0, 1
L2:
     blt      a1, t0, L4
     slli     t4, t0, 3
     add      ra, a0, t4
     lw       t1, 0(ra)
     addi     t2, t0, -1
L3:
     blt      t2, 0, L1
     slli     t4, t2, 3
     add      ra, a0, t4
     lw       t3, 0(ra)
     bge      t1, t3, L1
     slli     t4, t2, 3
     addi     ra, a0, t4
     sw       t3, 0(ra)
     addi     t2, t2, -1
     jal      r0, L3
L4:
     lw       ra, (sp)
     addi     sp, sp, 8
     ret      ra
```

Insertionsort in C

```
void sort(long num_elem, long array[]) {
  for (long i = 1; i < num_elem; ++i) {
    long x = array[i];
    long j = i - 1;
    while (j >= 0 && array[j] > x) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = x;
  }
}
```