Computer Architecture

Conversions

- **base x → 10**
  - $132_7 \rightarrow A_{10}$

$$1 \cdot 7^2 + 3 \cdot 7^1 + 2 \cdot 7^0 = 72_{10}$$

- **base 10 → x**
  - $32_{10} \rightarrow A_3$ $\quad r = 3 \quad [0, 1, 2] \quad w = [\ldots 3^3, 3^2, 3^1, 3^0]$

	div	mod	
32_{10}	3	10	2	lsd
10	3	3	1	
3	3	1	0	
1	3	0	1	msd

$A_3 = 1012_3$

- **base x → y**

$A_x \rightarrow A_{10} \rightarrow A_y$

- **base r → r^n**
  - $r = 3 \rightarrow r = 9 \quad n = 2$

$1\,2\,0\,2\,2\,1\,1\,2\,1\,0\,0\,2\,1\,2\,1 \quad r = 3$

$1\,6\,8\,4\,7\,0\,7\,7 \quad r = 9$

⊙ base 10 → Ex 128

• $A_{10} \rightarrow A_{10} + bias \rightarrow x_{NBC} \rightarrow A_{EX}$

$-102_{10} = -102 + 128 = 26 \rightarrow 0001.1010_{NBC} \rightarrow 0001.1010$

16+10

⊙ Ex 128 → base 10

• $A_{EX} \rightarrow x_{NBC} \rightarrow x_{10} \rightarrow x_{10} - bias \rightarrow A_{10}$

$0001.1010_{Ex} \rightarrow 0001.1010_{NBC} \rightarrow 26_{10} \rightarrow 26 - 128 = -102$

16 8 2

⊙ base 10 → 2C (from def) $\langle -2^{n-1}; 2^{n-1}-1 \rangle$ $M = 2^n$

(only for negative)

$A_{10} \rightarrow M - |A_{10}| \rightarrow x_{10} \rightarrow x_{NBC} \rightarrow A_{2C}$

• n = 8 - bit $\langle -128; 127 \rangle$ $M = 256$

$-80_{10} \rightarrow 256 - |80| \Rightarrow 176 \rightarrow 1010.0000_{NBC} \rightarrow 1010.0000_{2C}$

⊙ 2C → base 10

$A_{2C} \rightarrow x_{NBC} \rightarrow x_{10} \rightarrow x_{10} - M \rightarrow A_{10}$

• $1011.0000_{2C} \rightarrow 1011.0000_{NBC} \rightarrow 176 \rightarrow 176 - 256 \rightarrow -80_{10}$

128 16

⊙ base 10 → 2C (bit negation +1)

$A_{10} \Rightarrow |A_{10}| \rightarrow x_{NBC} \rightarrow \bar{x} + 1 \rightarrow A_{2C}$

• $-80 \rightarrow 80 \rightarrow 0101.0000 \rightarrow 1010.1111 + 1 \Rightarrow 1011.0000_{2C}$

80|2 0
40|2 0
20|2 0
10|2 0
5|2 1
2|2 0
1|2 1
0

⊙ 2C → base 10

$A_{2C} \rightarrow \bar{A} + 1 \rightarrow x_{NBC} \rightarrow x_{10} \rightarrow -x_{10} \rightarrow A_{10}$

$1011.0000_{2C} \rightarrow 0100.1111 + 1 \rightarrow 0101.0000_{NBC} \rightarrow 80 \rightarrow -80_{10}$

⊙ base 10 -> 2C (mixed-radix)

$A_{10} \rightarrow -128 + x_{10} \rightarrow 1\_\_^{NBC}\_\_\_2C$

- $-80_{10} \rightarrow -128 + 48 \rightarrow 1011.0000_{2C}$

⊙ 2C -> base 10

$A_{2C} \rightarrow 1\_\_^{x}NBC\_ \rightarrow -128 + x_{10} \rightarrow A_{10}$

$1011.0000_{2C} \rightarrow 1\_0\_1\_1\_.\_0\_0\_0\_0 \rightarrow -128 + 48 \rightarrow -80_{10}$

⊙ $31_x = 23_y$

$3x + 1 = 2y + 3$

$2y = 3x - 2$

$y = \frac{3x-2}{2}$

$x = 2 \quad y = 2 \quad \times$

$x = 5 \quad y = 5 \quad \checkmark$

System Base - Radix r

- fixed-radix - const. value for all digit positions
  eg. dec, hex, octal, bin
- mixed-radix - may have diff values for dig.it pos.
  eg. time (24, 60, 60), 1's's complement
- other than natural num. (negative, rational, complex, ...)

Systems using more digits than radix r are redundant.

Representation in redundant sys. is not unique.

r-radix system using standard digit set $[0 \dots r-1]$ is non-redundant.

Digits may have the sign - signed digits
eg $[-1, 0, 1]$

Positional fixed-radix sys with $[-\alpha, \beta]$ digit

↓

redundancy $\rho = \alpha + \beta + 1 - r$

Capacity

In non-redundant r-radix sys, with n-digit numb
- the range of repr. is $0 \dots r^n - 1$
- num of unique repr. is r^n

eg. 8-bits bin -> range $0 \dots 255$, 256 unique repr

Number of digits needed to accommodate numbers from arbitrary range $0 \dots max$:

$$n = floor[\log_r max] + 1 = ceil[\log_r (max+1)]$$

eg. for 50000 nums in bin

$$\lfloor \log_2 49999 \rfloor + 1 = \lfloor 16,61 \rfloor = 16 \text{ dig (bits)}$$

$$\lceil \log_2 50000 \rceil = \lceil 15,61 \rceil = 16 \text{ dig (bits)}$$

Non-Positional numerical codes

- **Gray Code** - non-positional bin code
  - codes of every 2 succesive vals differ only in 1 bit
  - codes for 1 and last code also differ in 1 bit

- **BCD - Binary Coded Decimal**
  - each dec digit coded with 4 bits

- **NBC - Natural Binary Code**
  - fixed radix-2 with $[0,1]$ dig set
  - n-bit repr of non-negative vals $[0 \dots 2^n-1]$
  - NBC cannot repr. negative values

Negative Numbers Coding

- **SM - Signed - Magnitude**
  - most significant bit repr. the sign of the number
    (1 - negative, 0 - positive)
  - symetrical range of repr. $[-2^{n-1}+1, 2^{n-1}-1]$
  - disadvantages: complex arith. operations (add / sub), double repr. of 0

- **Excess - N - bias coding**
  - range of $[-N, +P]$ is maped onto positive $[0, N+P]$
  - conversion req. addition of bias value
    eg $[-4, +11]$, bias = 4 $\rightarrow$ $[0, 15]$
    $-1 \rightarrow +3$
  - range of repr. $[-2^{n-1}, 2^{n-1}-1]$
  - bias N amounts to 2^{n-1}

- ~~Biased Coding~~
- **Complement Coding**
  - range $[-N, +P]$ is mapped onto positive $[0, N+P]$
  - positive numbers are identical as in NBC
  - Repr. of negative numbers is calculated as complement to constant $M = N+P+1$
    - $x \Rightarrow M - x$ eg. $[-4, +11]$, $M = 16 \rightarrow [0, 15]$
      $-1 \Rightarrow 15$

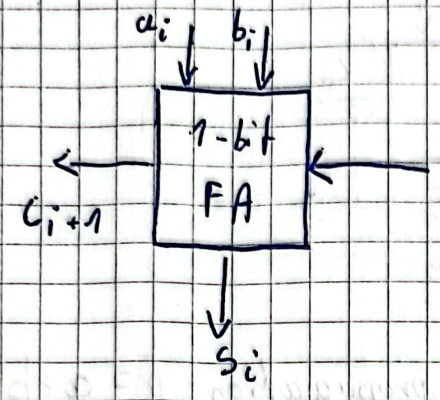
- **Binary 2's complement Coding (2C)**
  - range of n-bit num repr. $[-2^{n-1}, 2^{n-1}-1]$
  - complement constant $M = 2^n$ (radix complement)
  - most signif. bit corresponds to sign (1-neg, 0-po
  - negation: bit_negation(x) + 1
  - ignore last carry bit

- **Binary 1's complement Coding (1C)**
  - range of repr. $[-2^{n-1}+1, 2^{n-1}-1]$
  - Compl. constant: $M = 2^n - 1$ (digit-complement)
  - most signific. bit corresponds to sign (1-neg, 0-po
  - double repr of 0
  - negation: bit_negation(x)
  - adding an carry bit from last pos to total

1-bit Adder

Fundamental building block for arithmetic hardware

- 3 inputs → 2 outputs (Full Adder - FA)
- 3/2 bit-counter (output in NBC is the num of inputs 1's)



c_i	a_i	b_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_{i+1} = a_i \cdot b_i + (a_i + b_i) \cdot c_i$$

$$s_i = a_i \oplus b_i \oplus c_i$$

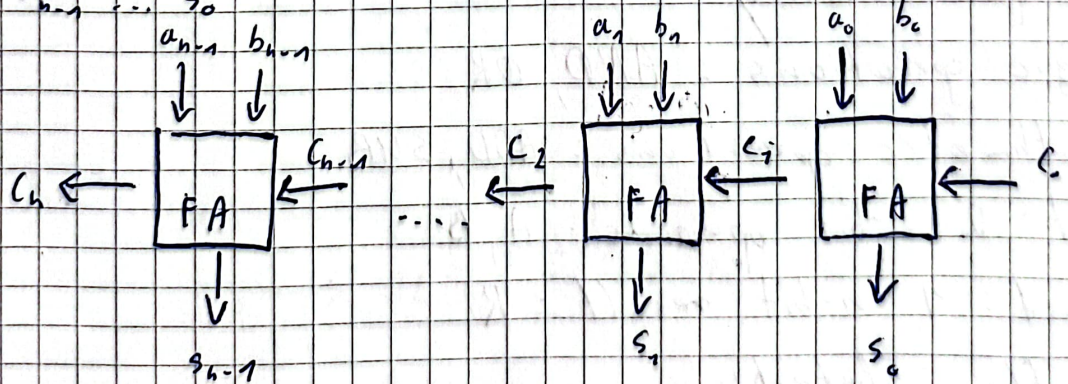
n-bit Adder

- Chained connection of FA → propagation of carry bit
- n-bits at pos from 0 to n-1
- result (sum) is n-bit long, last carry is handled separately

$A = a_{n-1} \dots a_0$

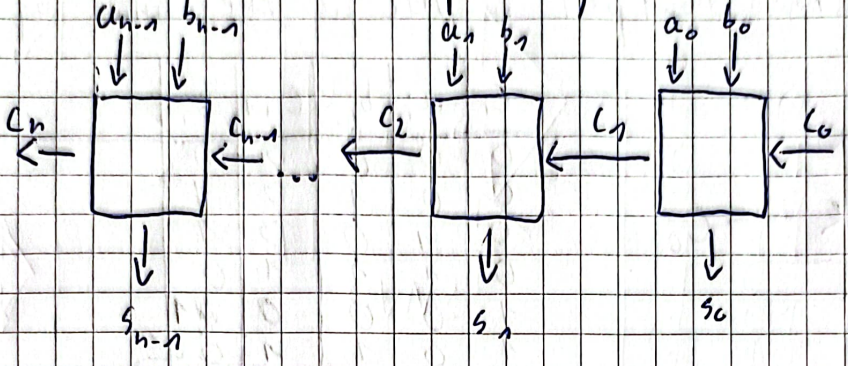
$B = b_{n-1} \dots b_0$

$S = s_{n-1} \dots s_0$



Ripple-Carry Adder

- simplest n-bit adder
- linear time complexity $O(n)$

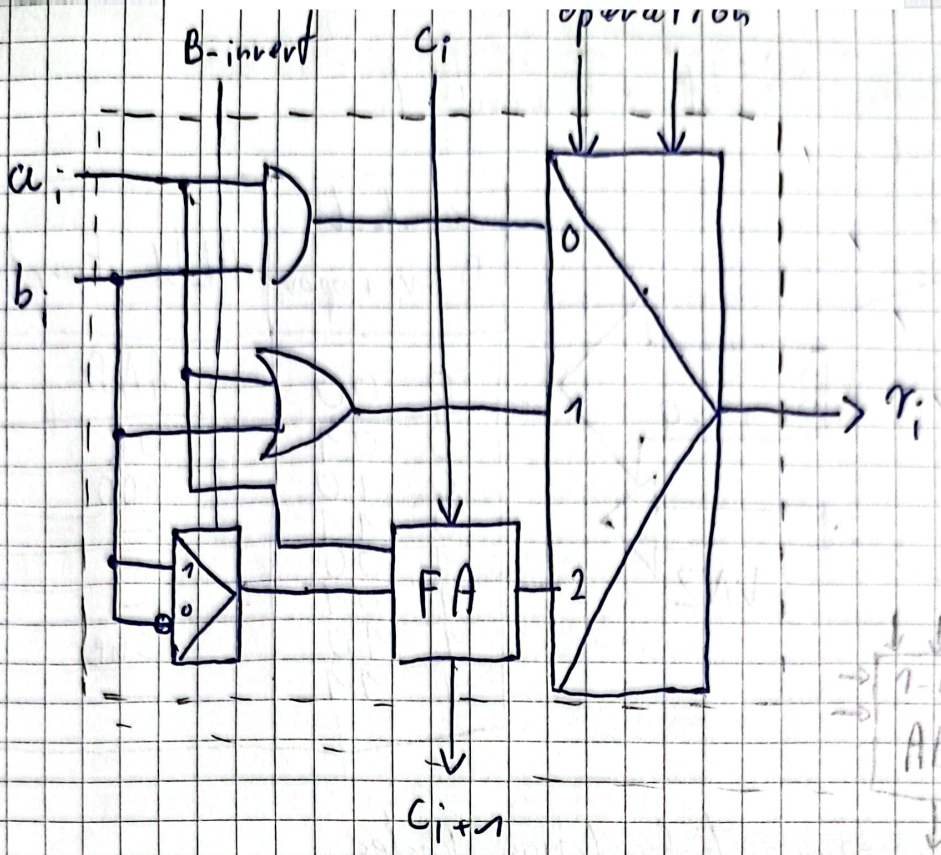


A	0	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	
B	1	0	1	1	1	0	0	1	1	1	0	0	1	1	1	1	
cond	p	p	p	g	p	p	a	p	p	g	a	p	g	p	p	g	
carry				1	0	0	0			1	0		1			1	①
S				S	S	S				S						S	
C			1	1	0	0	0		1	1	0	1	1		1	1	②
S			S	S	S	S			S	S		S			S	S	
C		1	1	1	0	0	0	1	1	1	0	1	1	1	1	1	③
S		S	S	S	S	S	S	S	S	S	S	S		S	S	S	
C	1	1	1	1	0	0	0	1	1	1	0	1	1	1	1	1	④
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	

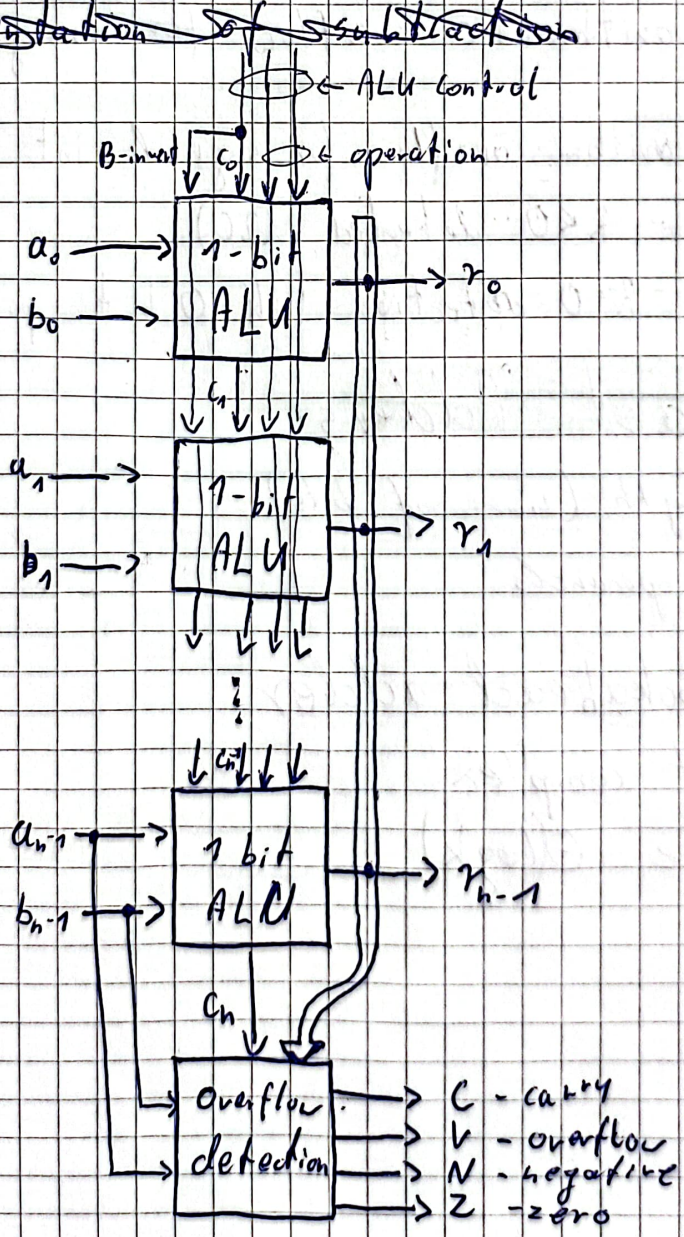
propagation - $p = a \oplus b$
generation - $g = a \cdot b$
absorption - a

ALU - Arithmetic Logic Unit

- basic computation unit for RCPU
- min functionality:
  - logic operations: AND, OR
  - arithmetic operations: ADD, SUB
- Input: 2 n-bit operands: A, B
- Output: 1 n-bit result: R
- overflow detection
- status flags / cond. codes: C, V, N, Z
- control bus

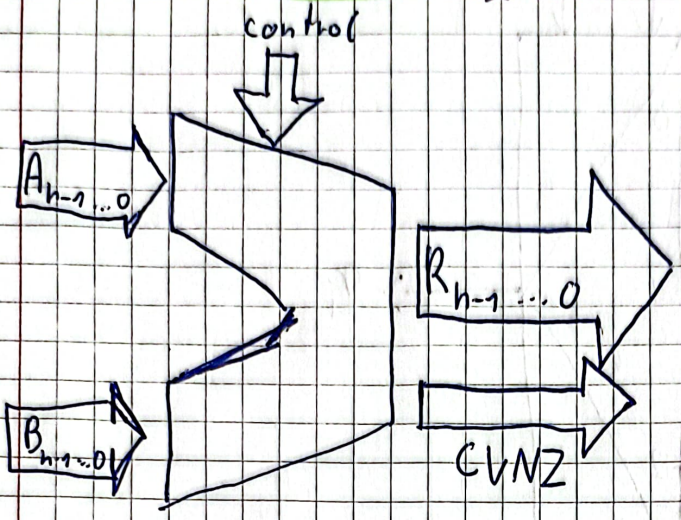


~~implementation of subtraction~~



$$A - B = A + inv(B) + (c_0 = 1)$$

n-bit ALU + Control



control B_inv + oper.	ALU Function
0 00	AND
0 01	OR
0 10	ADD
0 11	—
1 00	—
1 01	—
1 10	SUB
1 11	—

Status flags / Condition Codes

C - Carry - arithmetic overflow for unsigned integer arithmetic (NBC)

V - Overflow - arithm. overflow for signed int arithm. (2C)

N - Negative - R<0 detected (2C)

Z - Zero - R=0 detection (all-0 bit pattern)

Fast Adders

k - operand length (num of bits)

n - num of operands

CLA - carry lookahead adder

- fast but complex
- worst-case $\Theta(\log k)$

CSLA - Carry Select Adder

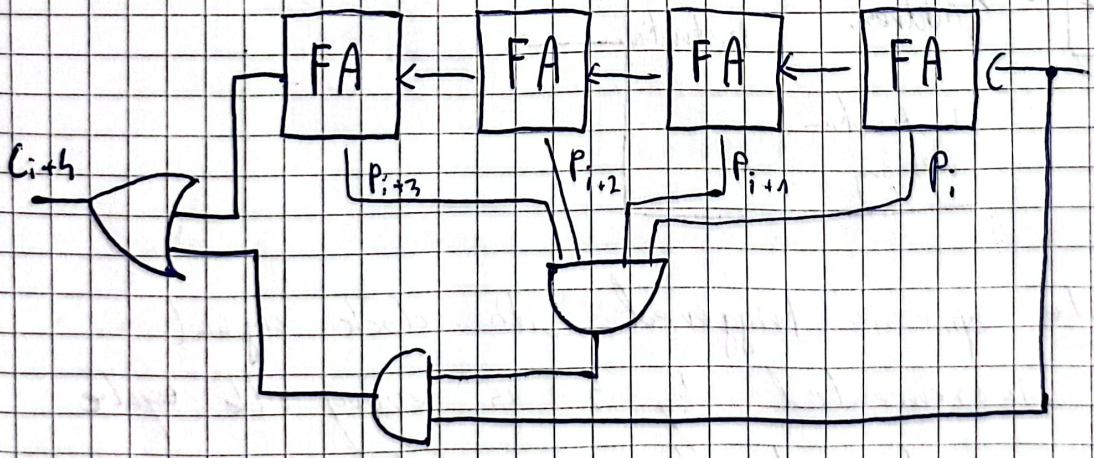
- worst-case $\Theta(\log k)$
- additions are performed in parallel according to alternative scenarios: carry = 0 or 1
- final selectio of results is made wit computed val of carry

A 0101.0100.0001.1000
B ~~0001.1110~~
B 1011.1001.1100.1111

C	0	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0
C	0	0	0	1	1	1	1	0	0	0	1	1				
	0	0	0	0	1	1	0	1	0	0	1	0	1	0	0	0
	0	0	0	0	1	1	1	0								
	0	0	0	0	1	1	1	0	0	0	1	0	1	0	0	0

CSKA - Carry Skip Adder

- carries can be generated, propagated or absorbed
- propagation chains are evaluated in parallel
- worst case $\Theta(\sqrt{k})$



$p_i = a_i + b_i$

Redundant Number Systems

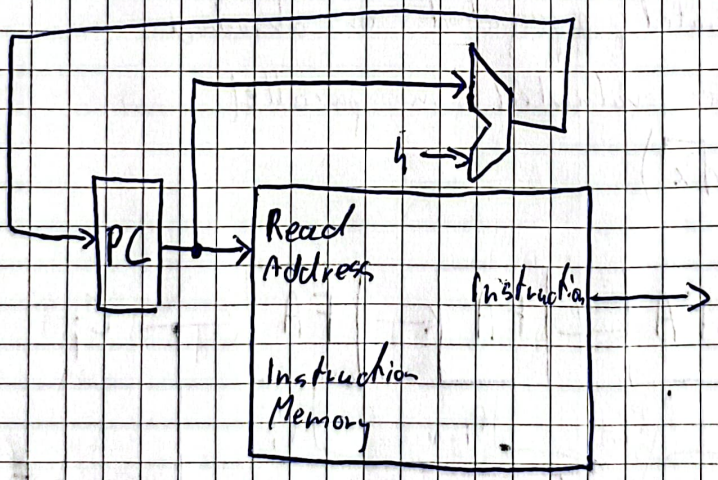
If the digit set contains more than r digits, the system is redundant

- Conversions bet redundant num sys is a simple digit-serial process
- redundancy allows for absorption of carry at each digit position

Single Cycle Architecture

Register PC contains address to instruction

Instruction Fetch block

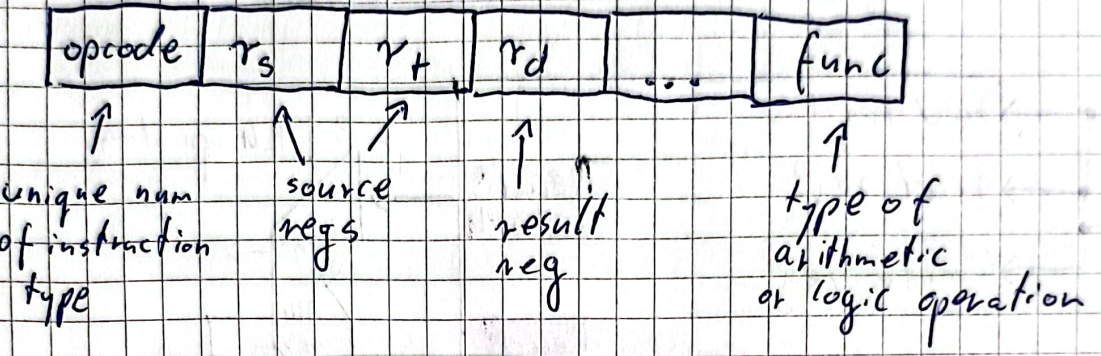


- PC Write op. is triggered with clock signal
- PC is incremented by 5 is every clk cycle
- A sequence of instructions is fetched from memor

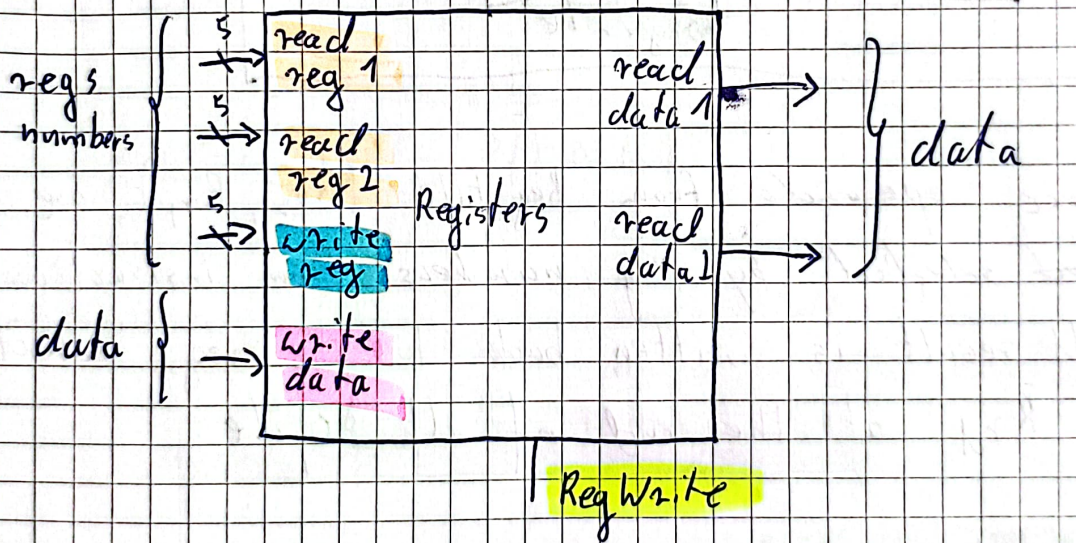
Register-type instructions (R-type)

R-type instruction perform operation only on internal registers of processor

- 2 source operands (Rr_s, Rr_t) are in internal regs.
- result is written to internal reg (Rr_d)

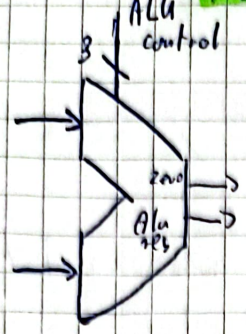


Register File



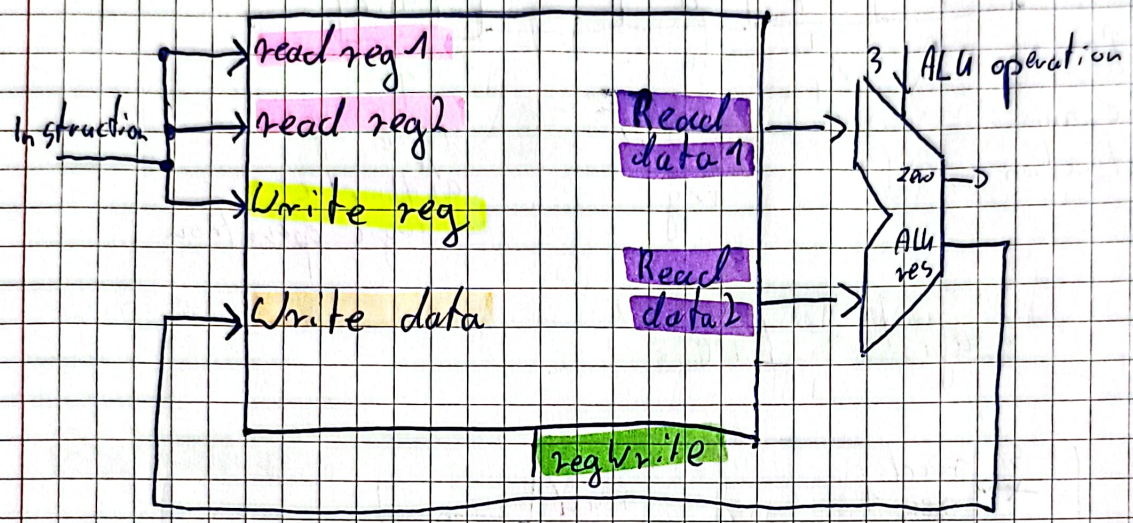
- contains 32 regs, each 32-bit wide
- output of reg file are contents of 2 regs addressed by ReadReg 1 & 2 input nums
- regs numbers are 5-bit wide ($2^5 = 32$)
- writing to a selected internal reg requires:
  - register number (Write Register)
  - data to be written (Write data)
  - operation enable ~~reg~~ signal (RegWrite)
- Write op is synchronized with clk signal

ALU



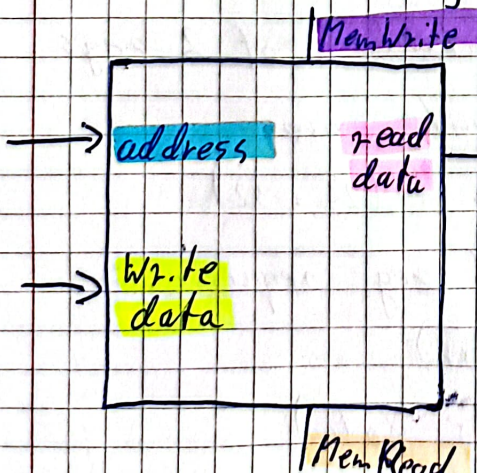
- 32-bit input & output
- operations: add, sub, logical: AND, OR
- 3-bit ALU-control bus
- only 1 output control signal: Zero (Z)

R-type Instruction Execution



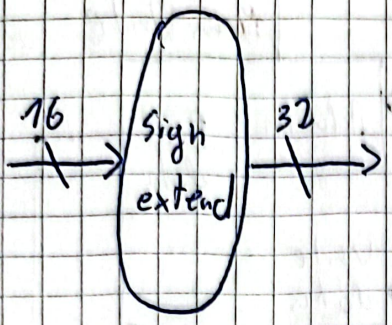
- Source operands from Reg File (Rr_s, Rr_t) are ~~selected~~ selected by reg numbers from instruction code
- ALU result is written back to the reg selected by Rr_d at the end of clk cycle

Data Memory



- Contains data of the program in 32-b
- data is present at readData after
address at Address bus is provided and op. enable signal (MemRead) is enabled
- Memory modification req. the data and address to be present at Write data & Address bus and op enable signal active (MemWri
- Write operation is sync with clk signal

Sign Extension Unit

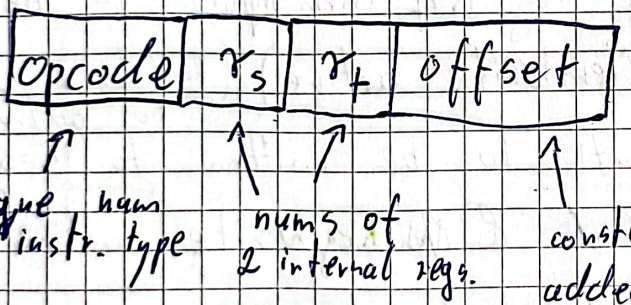


- performs conversion:
  16 bit bin num → 32 bit repr. with proper sign handling (2C)
- combinatorial logic, no clk signal required

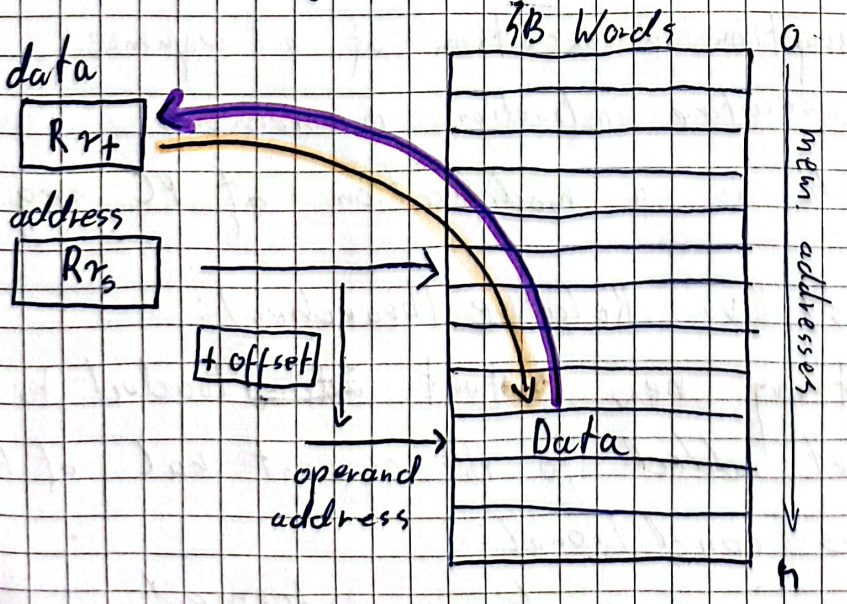
Transfer Instructions (Load/Store)

Instructions perform data transfer:

- data memory → internal reg. (Load)
- internal reg. → data mem. (Store)
- Rr_s - internal reg, Rr_t - memory address
- const val (offset) extends addressing range



Register Indirect Addressing



STORE

SW R7, (R5)

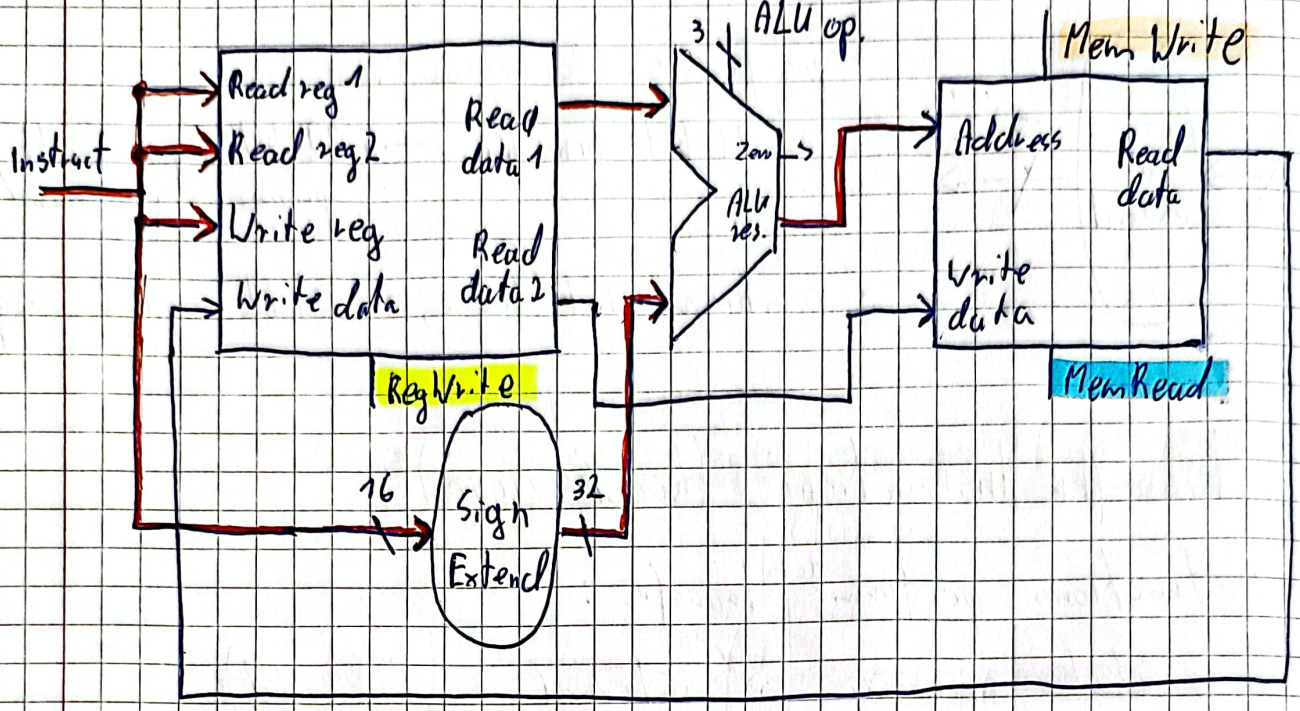
SW R1, 0x200(R2)

LOAD

LW R7, (R5)

LW R1, 0x200(R2)

Load/Store Execution



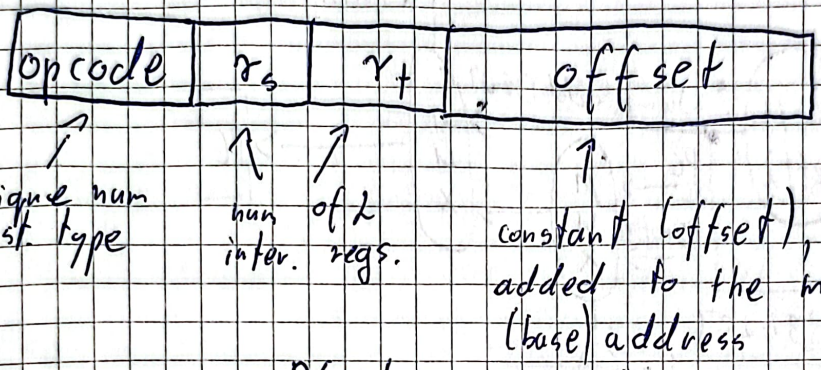
- Register R_{x_s} + offset points towards mem. data (Load/Store)
  (offset + read data 1 add in ALU goes to address)
- Store: contents of reg R_{x_t} (mem input) to be writt
  (signal MemWrite active) (Read data 2 written to "address")
- Load: mem. output to be written to reg R_{x_t}
  (signals MemRead & RegWrite active)

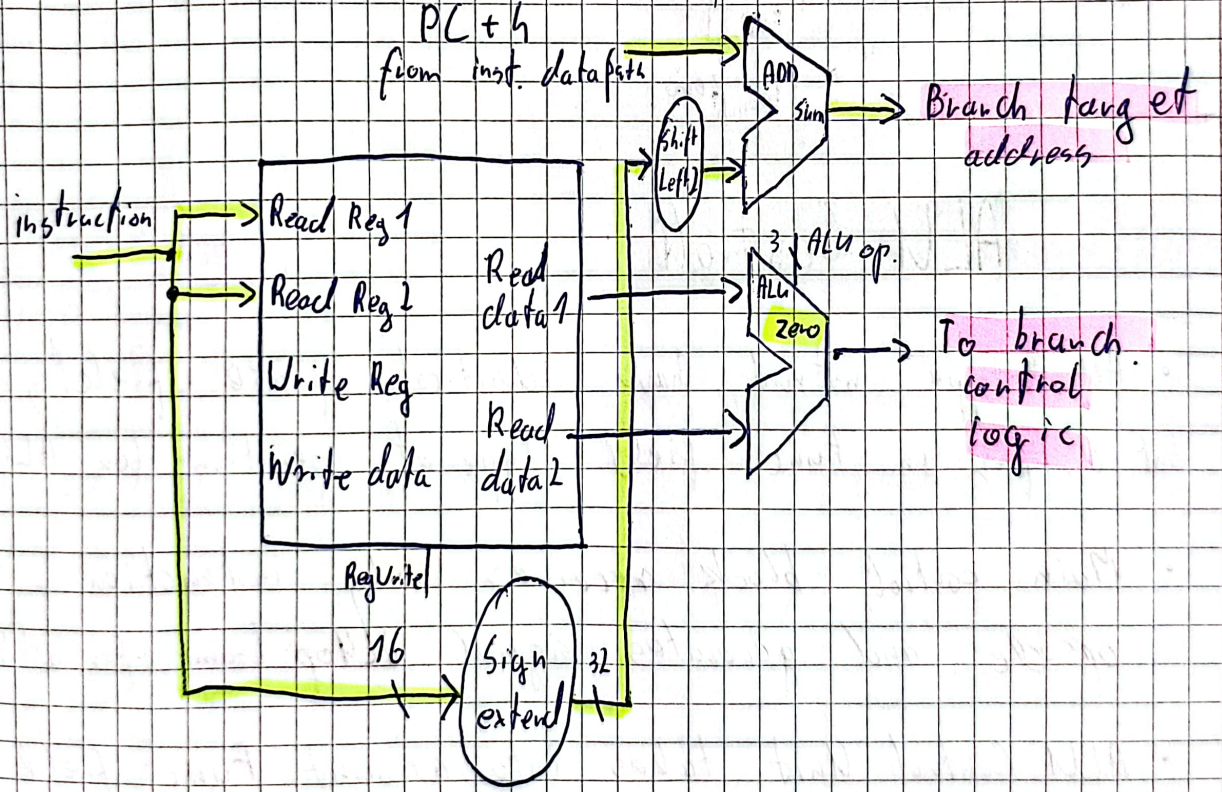
Jump/Branch Instructions

- Jump/Branch - interruption in execution of a sequence of consecutive instructions in memory
- Every Jump/Branch is a modification of PC reg.
- Absolute (jumps) vs Relative (branches):
  - absolute: arbitrary new content ~~added~~ loaded to
  - relative: offset added to the current val of P
- Unconditional vs conditional:
  - unconditional - jump is always performed
  - conditional - final modif. of PC depends on conditions

Relative Conditional Branch (BEQ)

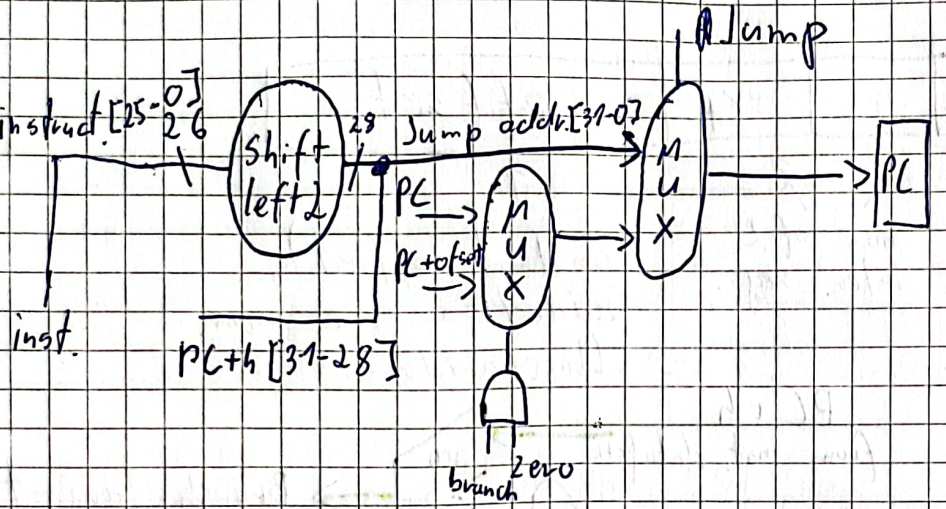
- BEQ Rx, Ry, offset (branch if equal)
  If Rx == Ry (2 = 1) than branch to address PC + offset * 4
- Instructions are 4B, so to widen the branch range we point to every 4th byte (offset * 4)
- BEQ base address refers to inst. mem.





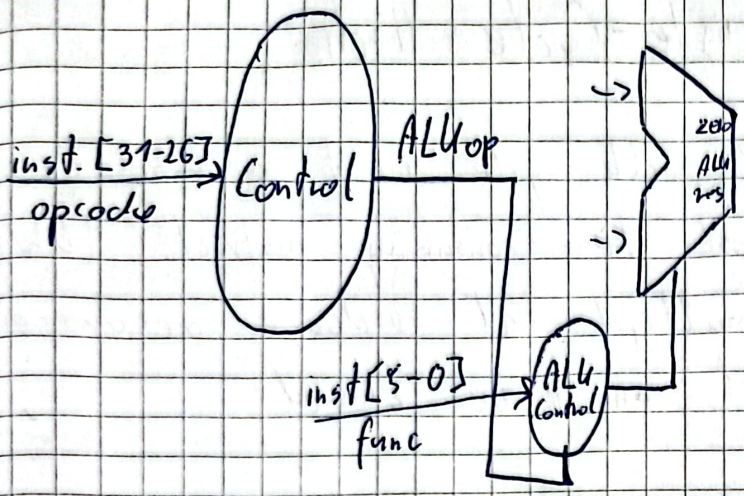
Absolute Unconditional Jump (JMP)

- new, arbitrary val loaded into PC
- no conditions
  - jump addr. mul. by 4 to extend jump range
  - Missing 4 MSB bits complemented from curr PC val (jump within "memory segment")
- multiplexer jump - selects source of next instr. addr



ALU Control

- All R-type instruct. have identical opcode field, but differ in Func field (type of operation for ALU
- Main control block receives only instruction opcode and generates signal ALUop (same for all R-t
- ALU Control Unit takes into account Func field (only for R-types) and provides direct control for AL
- ALUop signal indicates instruction family, but not the actual oper.



Multiplexers controls:

- **ALUSrc** - selects the 2nd ALU operand
- **MemtoReg** - selects the data to be written to reg
- **PCSrc** - selects the source for new PC value
- **RegDst** - selects correct reg. num. to be modified ($R\text{-type} \rightarrow r_d$, $Load \rightarrow r_t$)

Control Unit

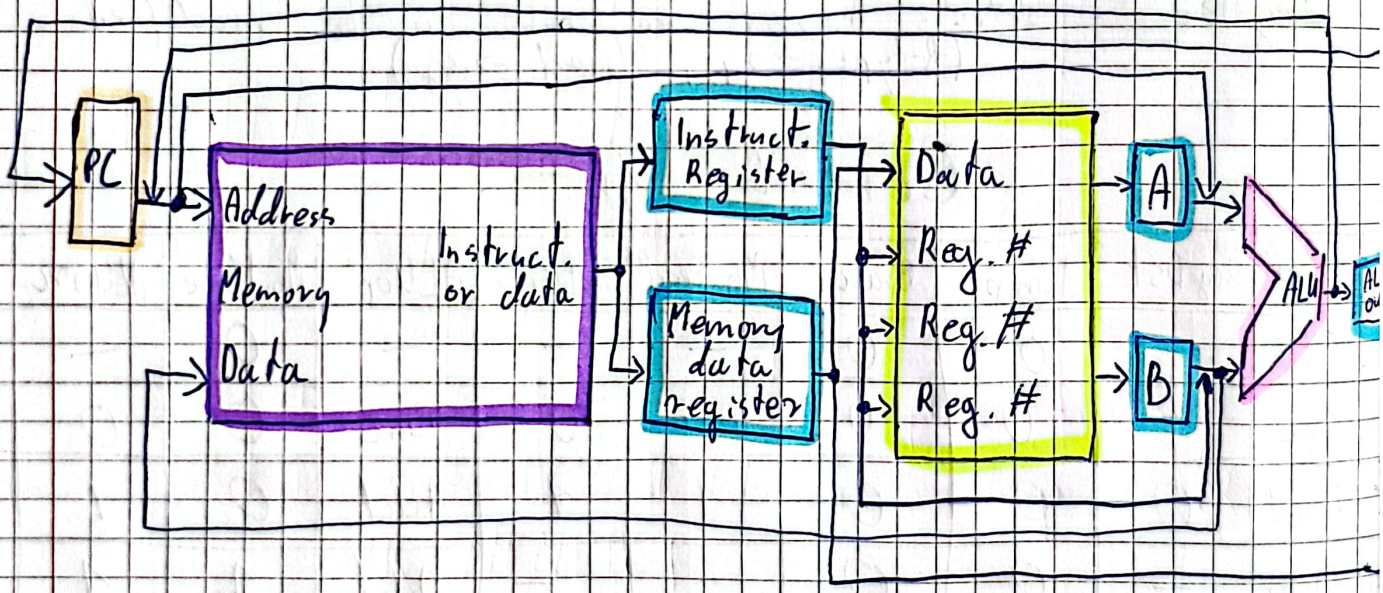
opcode	RegDst	Jump	Branch	MemRead	MemtoReg	ALUop	MemWrite	ALUSrc	RegWrite
NOP	-	0	0	-	-	-	0	-	0
R-type	1	0	0	-	0	Func	0	0	1
LW Load	0	0	0	1	1	Add	0	1	1
SW Store	-	0	0	-	-	Add	1	1	0
BEQ	-	0	1	-	-	Sub	0	0	0
JMP	-	1	-	-	-	-	0	-	0

Limitations of Single-Cycle Arch.

- **Slow** - duration of single clk cycle must accomodate exec. of the most time-consuming instruct. (LW)
- **Hardware-inefficient** - multiplicated hardware resources and cannot be shared

Multi-Cycle Architecture

- each instr. is exec. in several clk cycles
- instr. have various exec. time
- efficient use of hardware
- impl. of complex instruct.



Essential hardware:

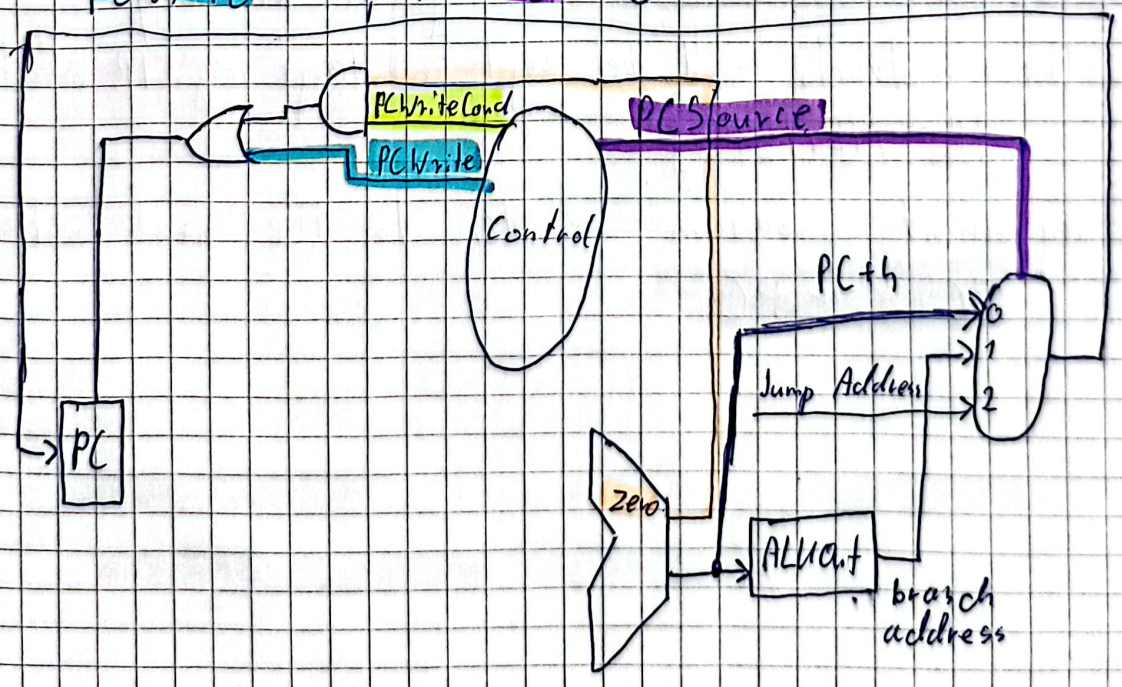
- **program counter** (PC)
- **single mem. block** (for both instruct. and data)
- **register file**
- **single general-purpose ALU**
- **intermediate registers**: IR, MDR / A, B / ALUOut

- Instruction Reg (IR) - holds the instruction code
- Memory Data Reg. (MDR) - data interface reg
- A, B - the operands for ALU
- ALUOut - result of ALU compulation

Each instruct. modifies the processor state by writing the result to a final destination:
- register file (R-type, Load)
- data memory (Store)
- PC (branch, jump)

PC Control

- Set PC to next instruct. (PC+4)
  - PCWrite = 1, PCSrc = 00
- Conditional branch (BEQ)
  PCWriteCond = 1, PCWrite = 0, PCSrc = 01, ALU - Zero
- Absolute jump (JMP)
  PCWrite = 1, PCSrc = 10



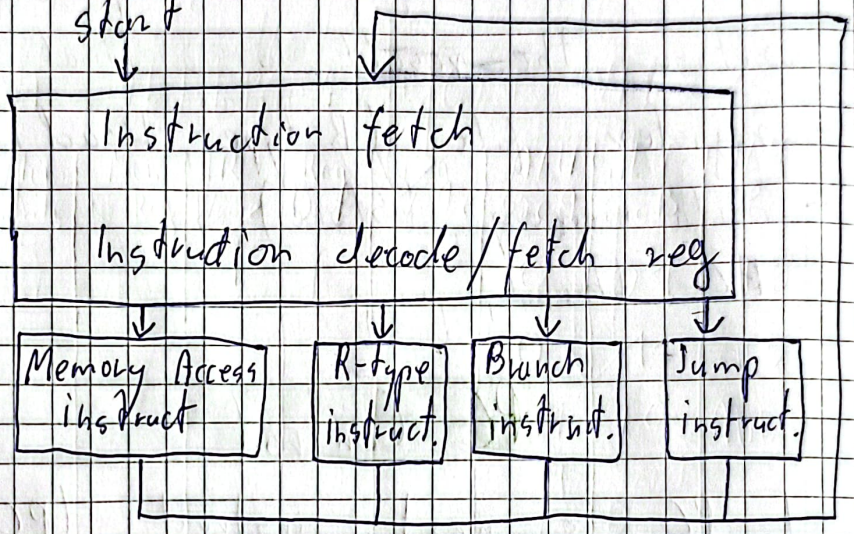
Instruction machine cycle

Common stages: 1, 2

- instruction fetch
- instruction decode & register fetch

Specific stages:

- Load / Store
- R-type
- BEQ
- JMP



1. Instruction Fetch

A. PC contents (address of code) are transferred to the instruction memory

B. The instruct. is read from memory

C. The instruct. will be kept in temp reg. IR (instruct. r

IR = MEM[PC]

D. Increment program counter do the next instruct.

PC + 4 → PC

2. Instruction decode / register fetch

A. Retrieving the appropriate reg. vals. from the reg. file to the ReadData1 port (contents of Rs) and the ReadData2 (contents of Rt)

```
A = Reg(Rs);
B = Reg(Rt);
```

B. Decoding the instruct. and creating control signals that will accompany the command exec.

C. Sign extending the operand encoded in bits 0-15 (i.e. 16 bits of offset/immediate) of the command itself even though we may not need this, just in case it turns out to be a branch command. Jump address is always calculated for branch inst. even if inst. ≠ branch.

ALU will calculate jump address and save it to ALUOutt

```
ALUOut = PC + [(signExtendedOffset) << 2];
```

3. Execute

We know now what command we have, we perform req. actions on the A and B vals that were retrived in prev. step.

R-type instructions:

ALU performs an op action on operands A & B

Result-ALU = A op B

Mem. access (SW/LW):

ALU calculates the effective mem. addr. by adding the OFFSET to Rs

Branch Commands:

(jump address is already calc. in ALUOut reg.)

ALU determines if A = B by sub. A - B. If result=0 then flag Zero is set and address in ALUout will be transfered to PC, else PC will remain PC+4

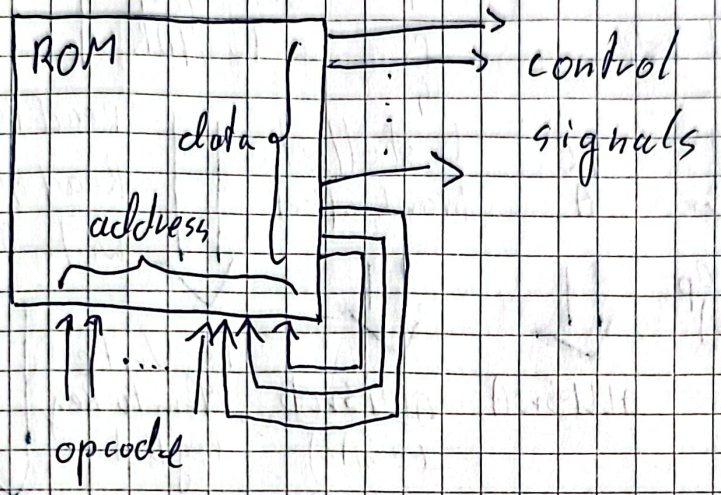
Jump Commands:

PC is set to 4 bits[PC+4] + 26 bits INDEX << 2

Instruct	PCWriteCond	PC Write	IorD	Mem Read	Mem Write	Mem to Reg	IR Write	Reg Dst	Reg Write	ALU SrcA	ALU SrcB	ALU OP	PC Src
Fetch	0	1	0	1	0	–	1	–	0	0	01	ADD	00
Decode	0	0	–	–	0	–	0	–	0	0	11	ADD	
LW/SW	0	0	–	–	0	–	0	–	0	1	10	ADD	–
LW Mem Read	0	0	1	1	0	0	0	–	0	–	–	–	–
LW Comp[	0	0	–	–	0	1	0	01	1	–	–	–	–
SW Mem Write	0	0	1	–	1	–	0	–	0	–	–	–	–
R-type exec.	0	0	–	–	0	–	0	–	0	1	00	FUNC	–
R-type comp.	0	0	–	–	0	0	0	10	1	–	–	–	–
BEQ	1	0	–	–	0	–	0	–	0	1	00	SUB	01
JMP	0	1	–	–	0	–	0	–	0	–	–	–	10

ROM-based Control Unit

- Ctrl. signals - ROM words
- opcode and state - ROM address
- microcode - ROM content



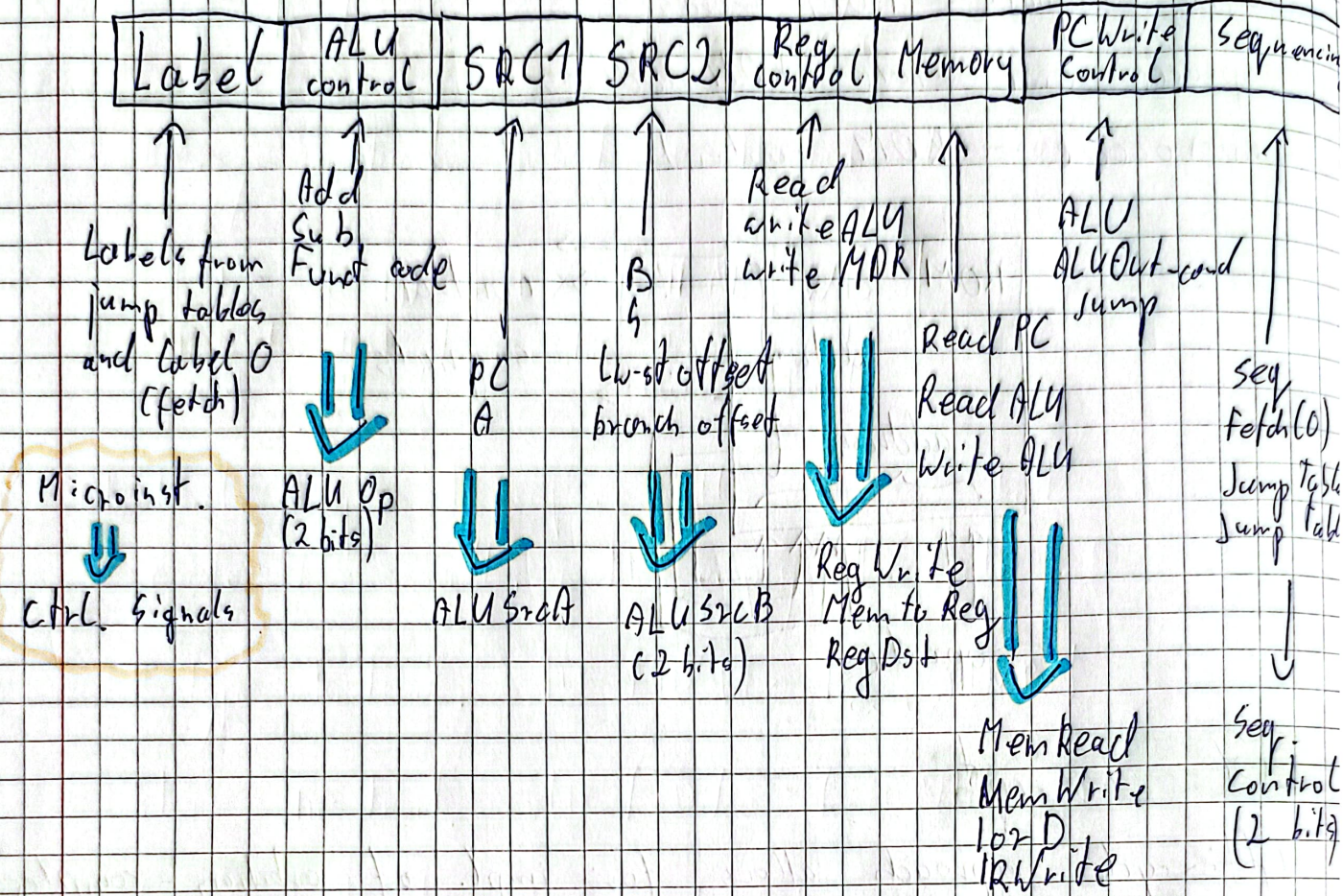
CISC concept

- Multi-cycle approach allows for impl. of arbitrary-complex instruct. without slowing down already impl' ones.
- Mem-based impl. of ctrl. as microcode (in reprogrammable ROM) simplifies the disign and synthesis of new instructs.
- Microcode can be easily inherited & enhanced bet gens. of CPUs, assuring backwards compatibility of inst. lists

Microprogram-based Control

- Ctrl. Unit is a ROM organized in n-bit words (microinstructs) representing ctrl. signals and addressed by microinstr. counter.
- microinstructs read from mem. in consecutive clk. cycles provide proper exec. sequence for every CPU inst.
- All sequ. of microinstructs that impl. CPU instr. are called microprogram

Microinstruction structure



Fetch & Decode in microcode

Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	branch offset	Read			Jump Table 1

Exceptions & Interrupts

An occurrence of predictable event at unpredictable moment during program exec.

- event from within processor
  eg. arith. overflow, div-by-zero, undef inst., sys-call, ect.
- event from outside processor
  eg. request from periph., mem access fault

Handling of interrupts requires immediate change of normal program seq. by jumping to a handling routine
eg. killing a process, outputing error msg, crash-stop computer, e

Handling Exceptions / Interrupts

- **Actions** (minimal)
  - save curr PC (into EPC reg.)
  - save exception reason / type (into cause reg.)
  - disable further interrupts / except. from occuring (status reg.
  - set PC (jump) to hardwired exception handler address
- **Methods**
  - polled handling:
    eg. jump to a common address of general handling routine and examine the cause reg.
  - vector handling:
    eg. jump to an address of invidual handling routine taken from the vector table

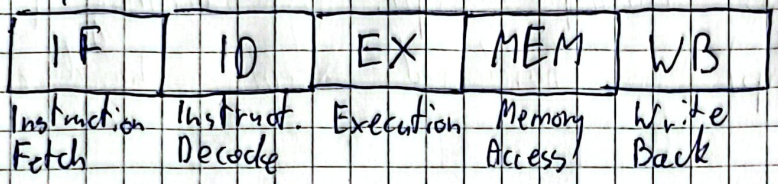
Simplified Impl. of Excepts in CISC

- 2 types of exceptions:
  - illegal instruct.: attempt to exec machine code with opcode not corresponding to any valid inst.
    Cause = 0
  - arithmetic overflow: overflow bit during 2C oper. in ALU
    Cause = 1

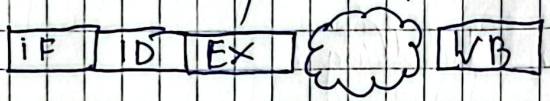
Pipelined Architecture

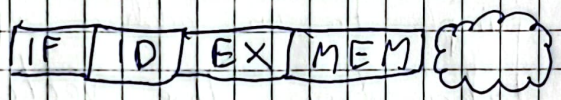
Instruction Machine Cycle

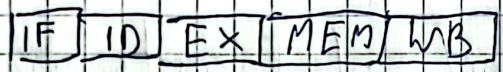
Every instruct. must go through the same stages

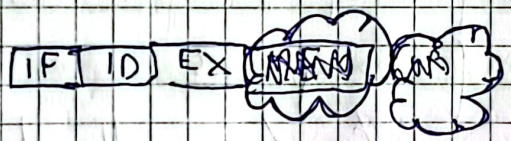


Instruct. may not require all stages but must execute them all, with empty clk cycle if needed

R-type Instruct. 

Store Instruct. 

Load Instruct. 

Branch & jump Inst. 

It is divided that way so in all stages of exe there are independent resources being used

IF - writing IR, incrementing PC
ID - register decoding
EX - operation with ALU
MEM - read or write memory
WB - writing final result to register file

- All resources are being used at any moment
- At any moment several inst. are being processed
- Every clk cycle 1 inst. is being effected, finishe

Structural hazard - resource conflict

There can happen a competition for hardware resources bet 2 instr. in pipeline

eg. access do the same memory: IF & MEM

Possible solutions:

- hardware duplication

  eg. independent program (IF) & data (MEM) mem. block (Harvard Architecture)

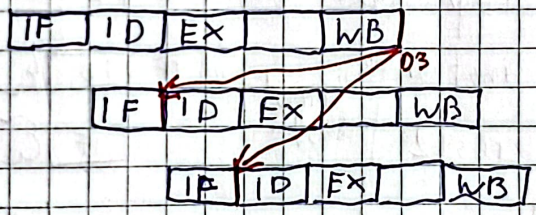
- pipeline stall - simplest and sometimes necessary, but always brings performance drop

Data Hazard

An attempt do use reg. content violating the reg. modification order.

eg.

```
ADD D1, D2, D3
SUB D4, D3, D5
MUL D4, D3, D1
```



Data hazards are very common so stall is wrong solution

Solution:

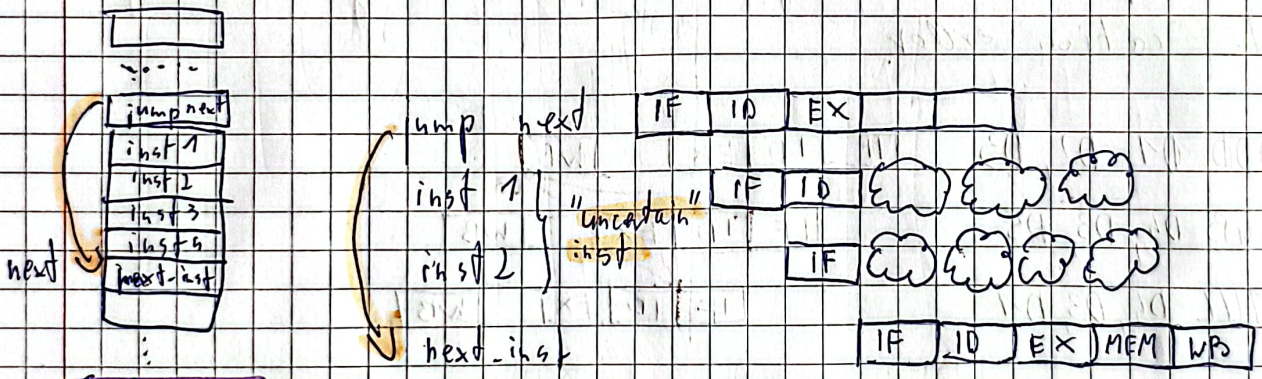
- Forwarding - using up-to-date results directly from earlier execution stages, without waiting for final reg. update

```
ADD ...
SUB ...
MUL ...
```

- **code optimization**; data hazards can be effectively eliminated by code optimization (by optimizing compilers). There is architecture dependend or independent optimizations

Control Hazard

- every branch or jump breaks instruct. sequence
- cond. branch delay the moment of exec. next instruct. until the cond. is evaluated
- pipeline CPU may allow to fetch some instructs. that are "uncertain" and invalidate them later if necessary (cause of the cond.)



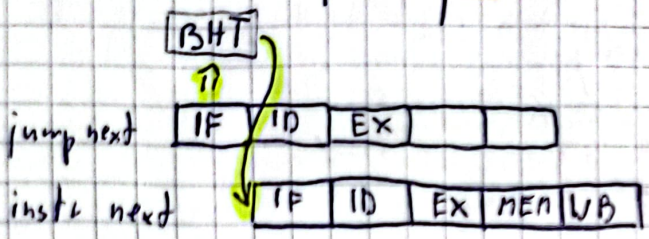
Solutions

Delayed Branch

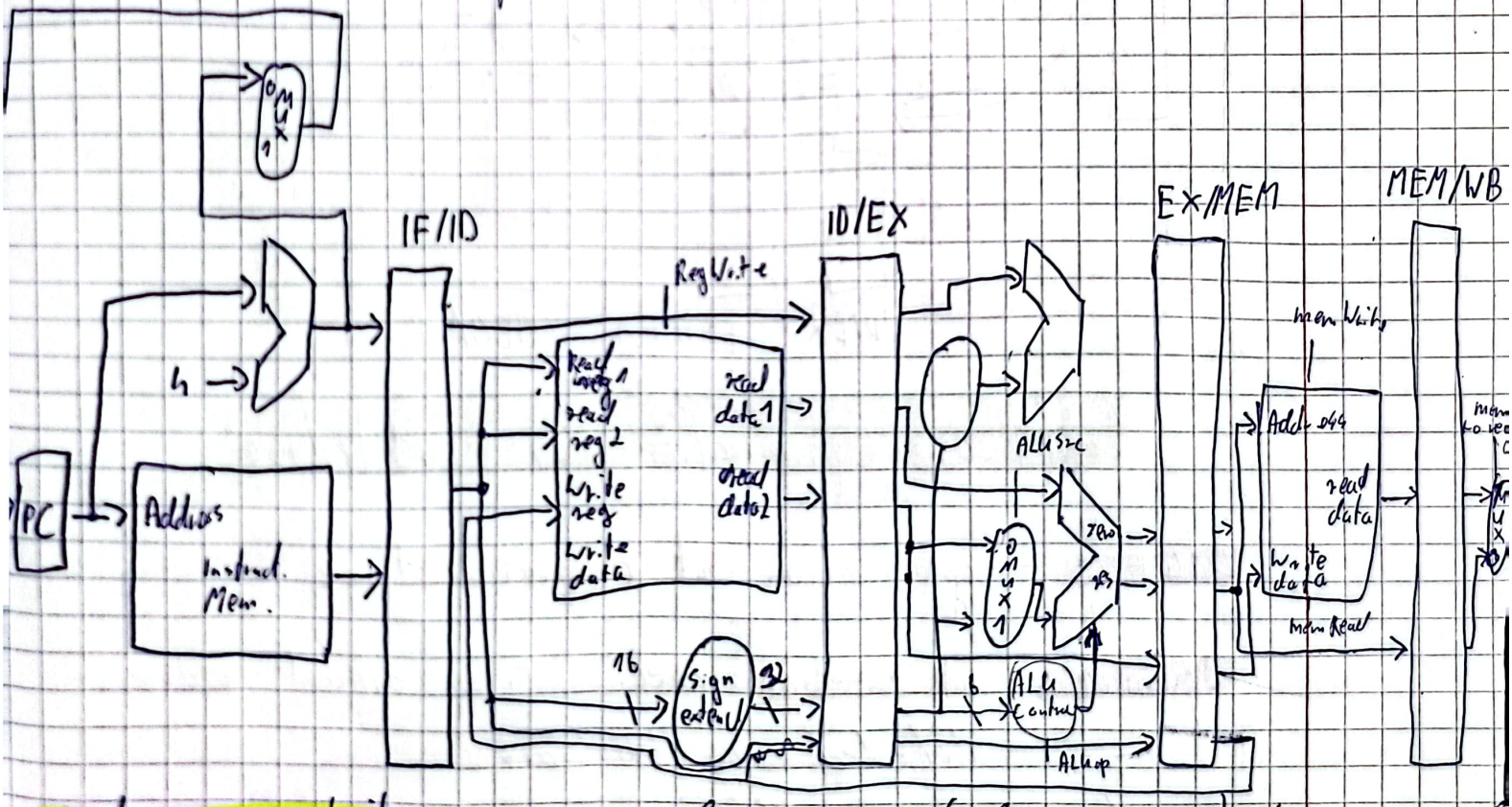
In delayed branch instructs bef branch and target, that entered the pipeline (in delayed slot), will be executed unconditionally. Those instructs must be chosen carefully (useful or at least harmless - NOP)

Branch History Table

BHT contains addresses of recently exec. branches and their last target - during IF stage the branch and target can be identified just in time. Then the conditional branch validates the entry in BHT and the target instruct can be invalidated if necessary. Unsuccesful predictions are limited to starts and ends of loops.



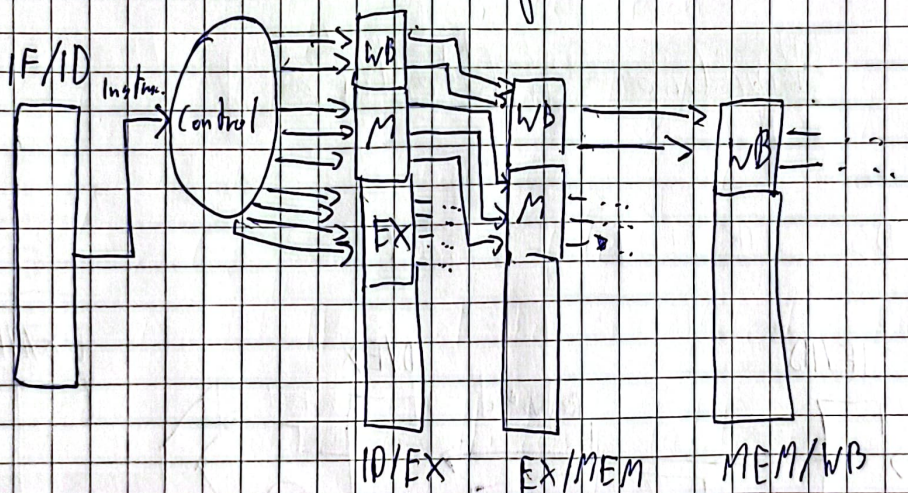
Pipeline Architecture



- Hardware complexity - resource duplication (adders, mem.) to avoid structural hazards
- Introduction of "long" intermediate regs to sync data flow bet stages of instruct. exec.

- instructs can use any resources only 1 time, more complex tasks which req. reuse of resources are impos
- instruct. list and functionality is reduced
- performance increase at cost of having simpler instruct → programs (MC) must be longer
- Ctrl unit is simple - result of reduced inst. list
- all ctrl signals must be developed in decode stage
- instruct. must "carry" controll signals through the pipeline and use them selectively

Control signals



Software Optimization for Architecture

- Static - optimization at compilation time (optimising compi
- Dynamic - at run-time: exec. insts in optimal order detected by hardware:
  - dynamic scheduling
  - out of order execution
  - speculative execution

Pipeline Stall

- Some stages must be repeated - other invalidated
- To achieve hardware pipeline stall we deactivate ctrl signals for stages: EX, MEM, WB and we postpone writing to PC and IF/ID

Forwarding

- Hardware solution to most data hazards (bet EX-MEM, EX-WB stages)
- transfer of newest results from EX/MEM & MEM/WB to ALU input

Data hazard types

- RAR (read-after-read): no conflict
- WAW (write-after-write) - output dependence: no conflict
- WAR (write-after-read) - anti-dependence: no conflict
- RAW (read-after-write) - true data dependence:
  - R-type after R-type → forwarding
  - R-type after LW → pipeline stall + forwarding

"HARD" Data Hazards

Forwarding cannot solve all data hazards:

- RAW → LW & ADD - necessary pipeline stall

- detection of hard data hazards must be done early (in ID)
- additional RAW-hazard detection (combinational comparator) block req. in ID
- RAW-hazard block should be transparent for main ctrl & forwarding unit
- RAW-hazard detects:
  - LW in EX stage (by ID/EX.MemRead)
  - conflicting inst. in ID (by opcode)
  - matching nums of regs (ID.Ex.Rt & IF/ID.Rs/Rt=ID.Rt)

Control Hazard

- Any jump/branch breaks the natural seq. of instructions and spoils the pipeline (CPI > 1)

- Cond. branches (apart from addr. calc.) must also calc. the cond - it may take extra time
- jump/branch exec. may require a few next instr. to be invalidated

- Effective solutions:
  - Early Branch Detection - req. additional hardware
  - Branch History Table - the best, but still based on guess

Early Branch Detection

- Cond. (simple) is calc. in ID stage - only 1 stage of delay
- only simple condition is allowed cause the regs must be read from reg. file
- instruct. in IF must be invalidated - turned into NOP

Branch History Table

- saving earlier jump/branch address
- speculative execution
- complex and late condition calc. is allowed
- no pipeline stall at all (in theory)
- target is a guess and req. validation
- wrong predictions causes invalidation of many instruct

Memory Access Time - describes the speed of data transfer bet memory and microprocessor

Memory Cycle Time - describes how often the memory access can be repeated

SRAM

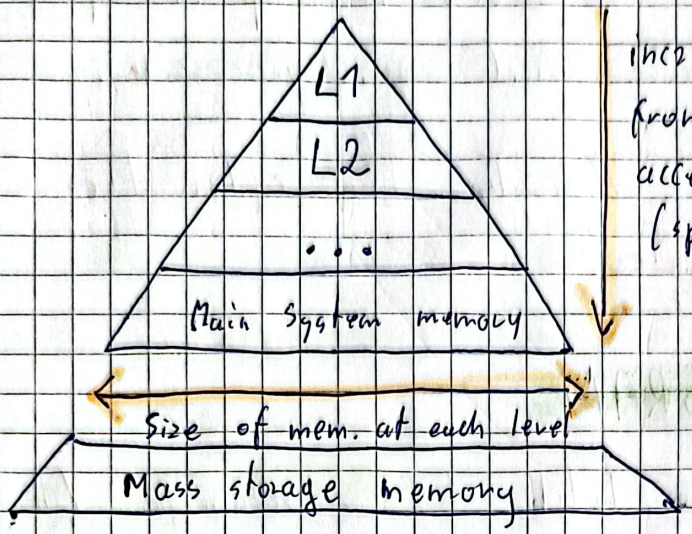
- bistable flip-flop or latch
- no need to refresh
- short access time
- more board space
- low retain power / high write power
- more heat dissipation
- high cost

DRAM

- charge in capacitor
- need to refresh
- long access time
- little board space
- low power heat
- low cost-per-size

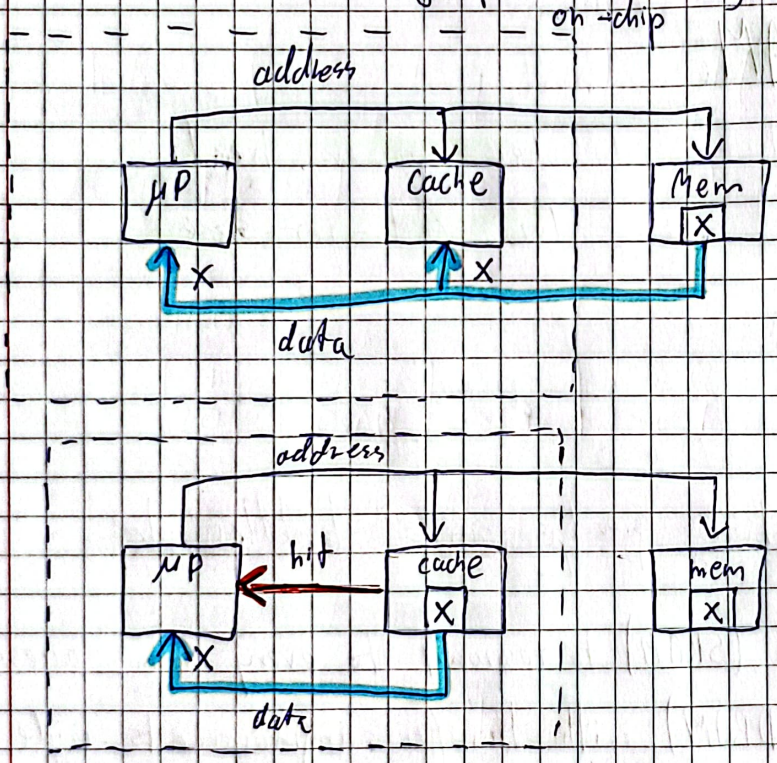
Solutions for mem. being performance bottleneck

1) Mem. fast enough (SRAM) to respond to every mem. access req.
2) Slow mem. sys. (DRAM) with transfer improvements: wide buses and serial accesses
3) Combination of fast and slow mem. sys. arranged so the fast one is used more often



Cache Principle

Reading from memory

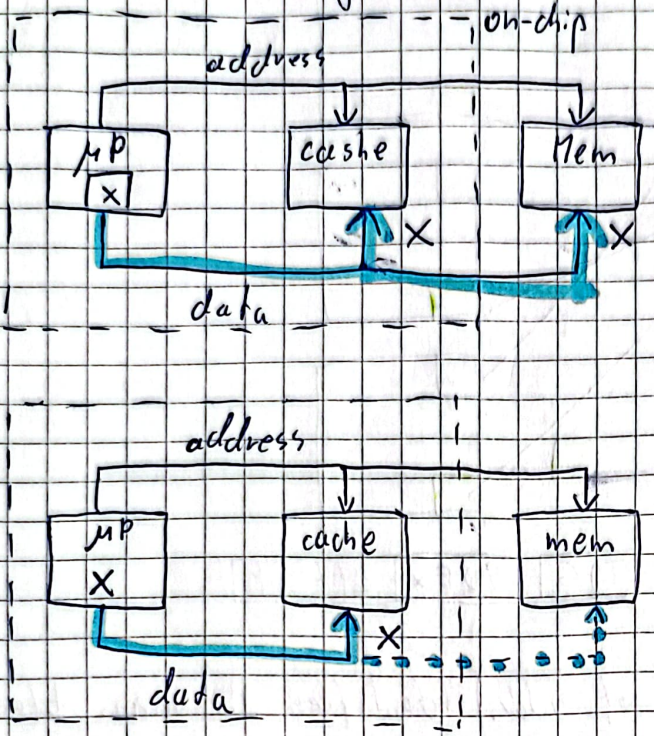


Data not found in Cach
transfer from mem. (slow
to both μP & cache

Data found in cache
transfer from cache (fas
to μP

Hit rate - fraction of accesses to cache memory in the total number of all memory accesses

Writing to memory



Write-through

Transfer to both: memory (slow) & cache

Write-back

Transfer to cach (fast) only

The mem. will be updated when this cache is claimed by other data

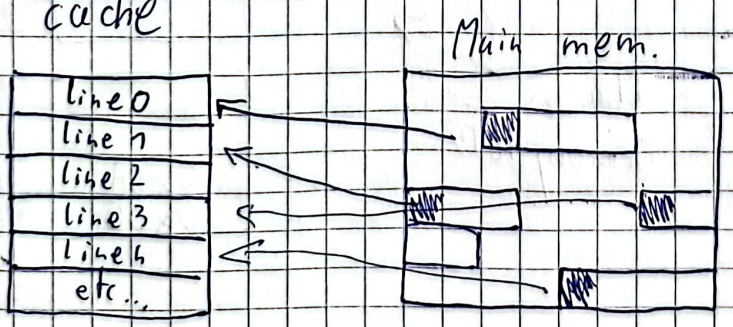
Principles of locality

Temporal locality (locality in time) -

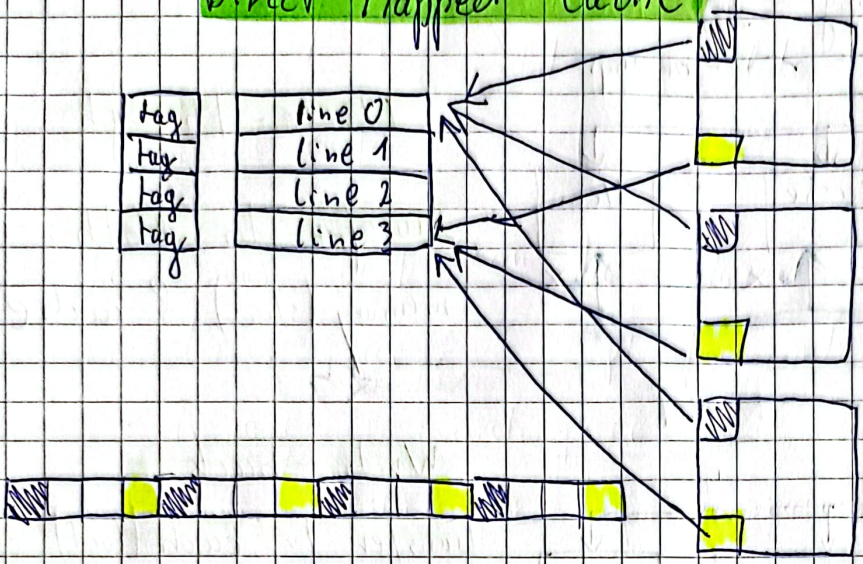
if an item was referenced, it will be referenced again soon

Spatial locality (locality in space) -

if an item was referenced, items close to it will be refed too



Direct-Mapped Cache



memory address → | TAG | INDEX | offset |

Tag - (most sig. part of addr.) identifies the mem. block the data come from

Index - (mid. part of addr.) identifies line numbers within cache (and block)

Offset - (least sig. part of addr.) identifies the byte (word) within a cache line

Eg.

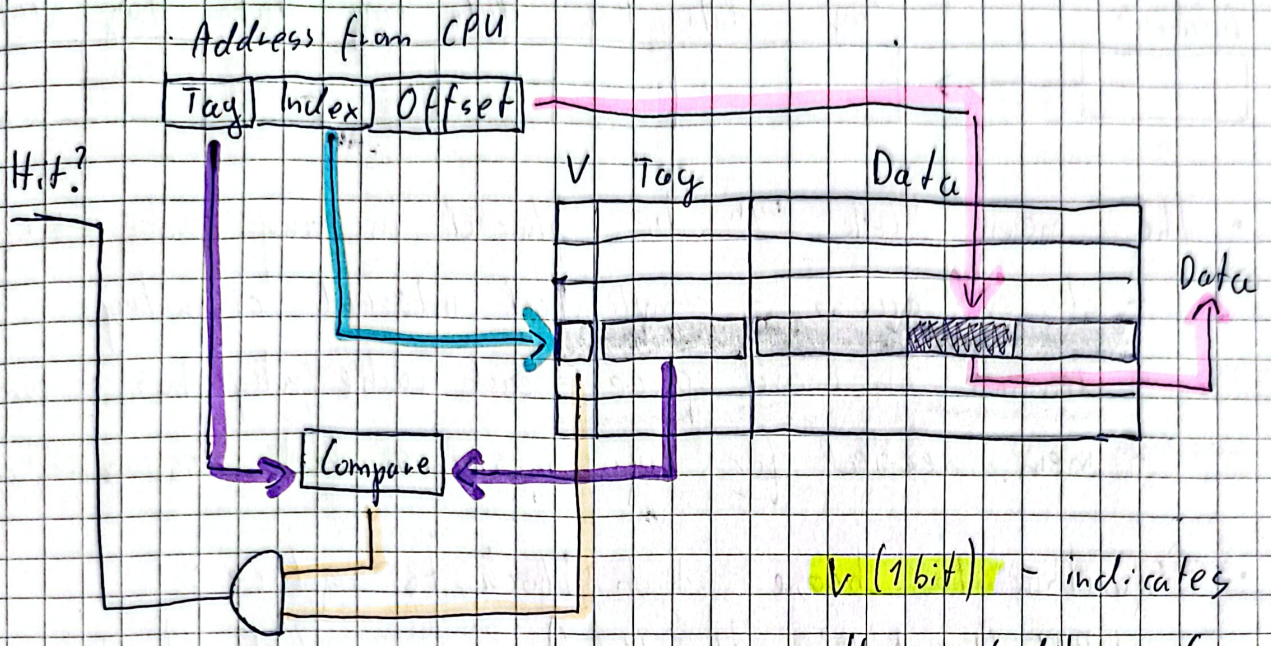
TAG	offset 1	offset 0	
01	0x05	0x04	Line 0
10	0x0B	0x0A	Line 1

cache 4B

Tag	Index	Offset
00	0	0
00	0	1
00	1	0
00	1	1
01	0	0
01	0	1
01	1	0
01	1	1
10	0	0
10	0	1
10	1	0
10	1	1

Main Memory 16B

hit signal



V (1 bit) - indicates the validity of the cache line

Cache Thrashing

When alternating mem. refer. point to the same cache line, the cache line is frequently replaced, lowering the performance.

Direct-Mapped caching offers no benefit in case of cache thrashing

Set-Associative Cache

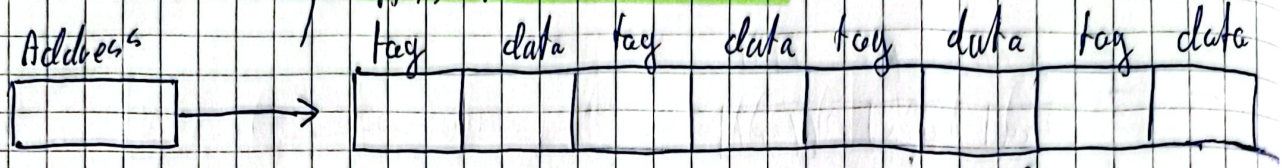
- The key to performance increase & thrashing reduction is the more flexible placement of mem. blocks by combining several direct-mapped caches.

Two-Way set-associative

Block	Tag	Data	Tag	Data
0				
1				
2				
3				
4				

The degree of associativity reduces the miss rate, at the cost of increase in the hit time and hardware complexity

Fully Associative Cache



- The mem. block can be placed in any cache line
  - slower access - complicated internal circuitry
  - demand on board space - each cache entry has a compare
  - mem. needed for tags increases with associativity

- Algorithm to choose which block to replace:
  - LRU (Least Recently Used) - req. additional bits for each cache line, updated each access
  - Random - candidates are selected randomly

Software Managed Caches

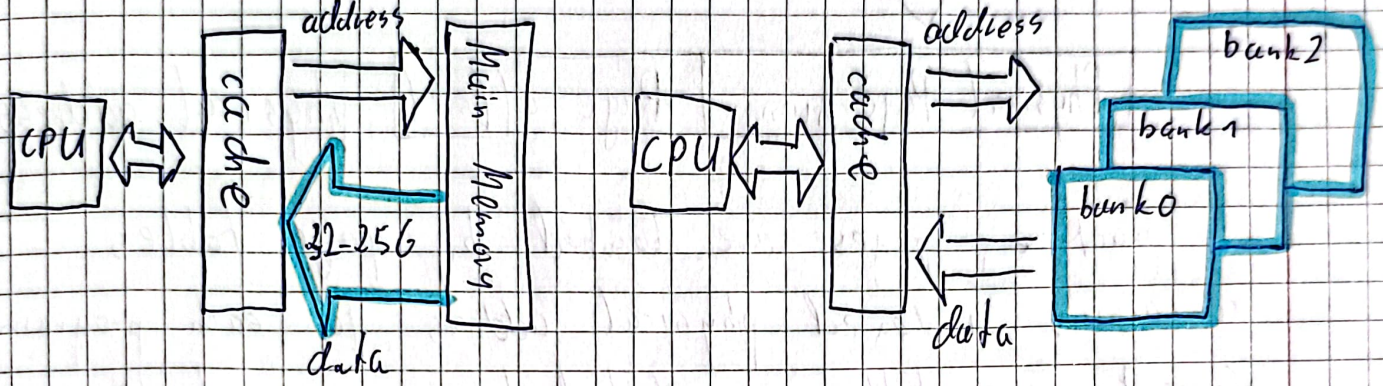
Idea: transfer the data to cache before CPU needs it so that the cache-fill time will be hidden and hopefully all mem. refs will operate at full cache speed.

Prefetching - method of loading cache mem. supported by some CPUs by impl. new instruct.
Prefetch instruct. operates as normal instruct., except that CPU doesn't have to wait for result
Compilers can generate prefetch instruct. when they detects data access using fixed stride

Memory Performance

2 main obstacles:

- Bandwith - best possible steady-state transfer rate (usually when running long unit-stride loop)
- Latency - the worst-case delay during single mem. access



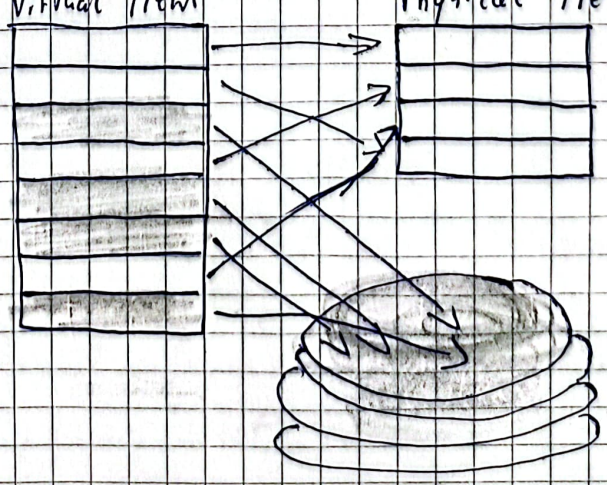
Wide mem. sys. -
- High bandwidth

Interleaved mem. sys. -
- Lower Latency

Virtual Memory

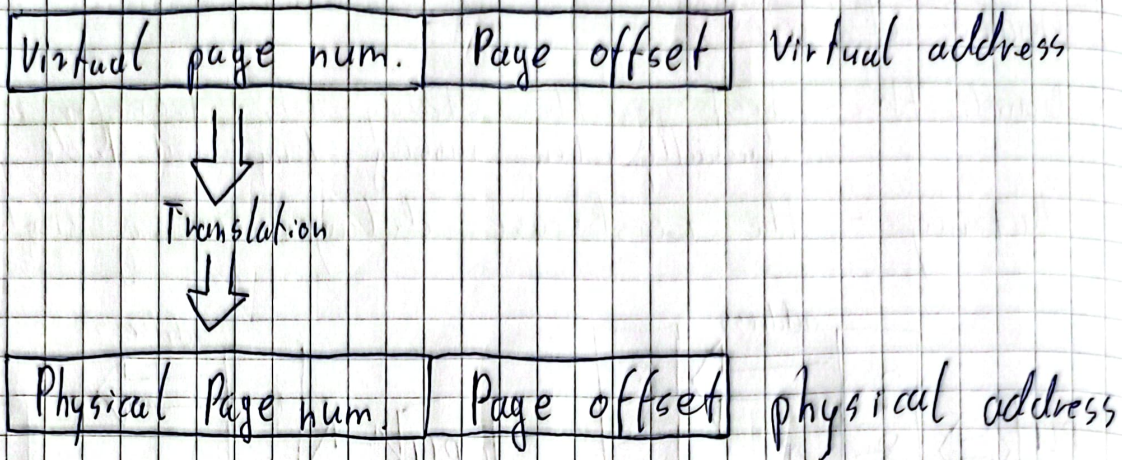
Idea: physical mem. sys. (DRAM) acts as "cache" for large data storage (magnetic disks)

- efficient physical mem. sharing in multitasking sys.
- programs can use mem. larger than physical
- virt. mem. takes away from programmers a burden of alloc. of mem. for programs
- using fixed-size block simplify the mem alloc. (no need for contiguous blocks)



cache	virt. mem.
line	→ page
cache miss	→ page fault

Addres Translation



In practice, pages are located by page tables containing indexed physical address for each program. Current page table is selected by a page table register associated with the given program.

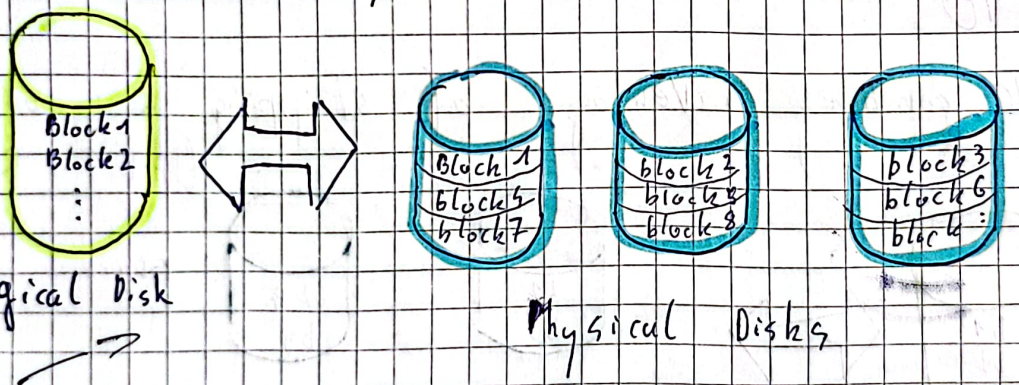
Massive Data Storage

- needed for:
  - virtual machines
  - multimedia processing
  - server systems (databases, etc.)

Disk Array - RAID

RAID - Redundant Array of Independent/Inexpensive Disks

- 1 coherent logical disk off out of many physical
- logical disk data are divided into to stripes (blocks)
- Distribution of stripes over physical disk may enhance speed and/or safety and is called RAID-mode



logical Disk

Physical Disks

RAID 0

Raid 0 creates logical disk by summing up the space of stripes distributed among physical disks

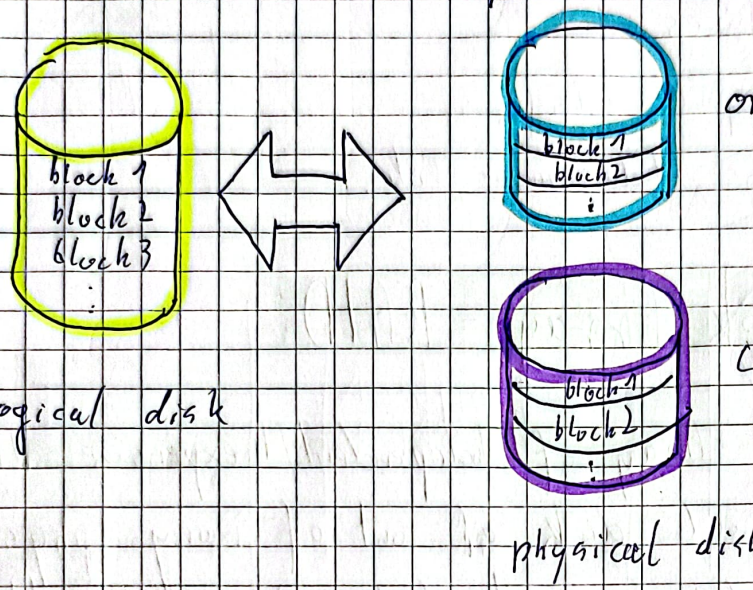
\+ highest speed, max disk space
\- not-fault-tolerant, lower MTBF

MTBF - Mean Time Between Failures:
avg time bef failures that can be repaired

MTTF - Mean Time to Failure:
mean time to failure that cannot be repaired

RAID 1

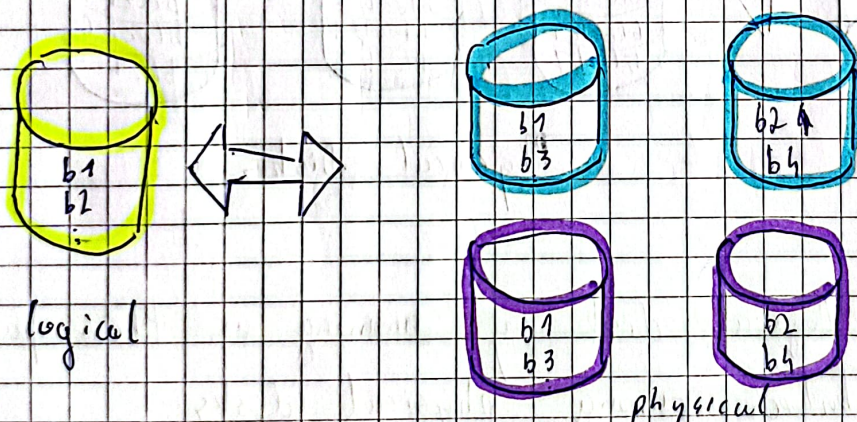
RAID 1 creates mirror copy of physical discs



\+ fault-tolerant, high speed, fastest data recovery
\- low disk space

RAID 10

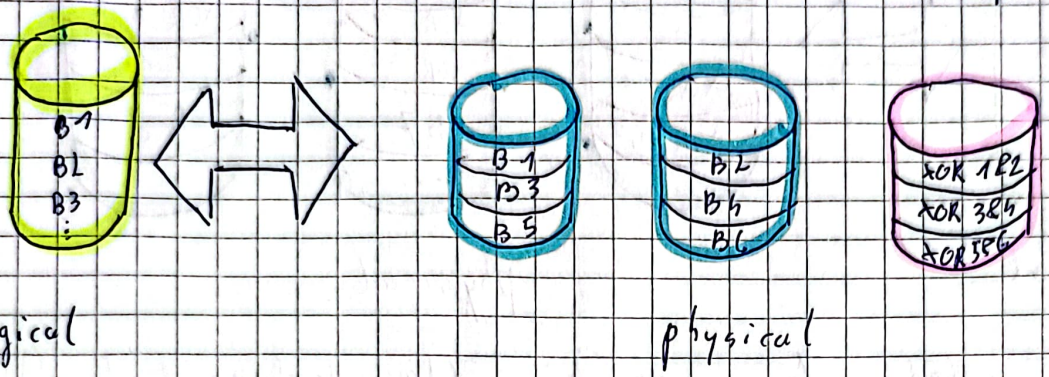
RAID 10 combines advantages of stripping & mirroring (RAID 1 + RAID 0)



\+ fault-tolerant, high speed, fast data recovery
\- low disk space

RAID 3&4

Creates logical disk by summing up the stripes on physical disks with additional dedicated parity disk



logical physical

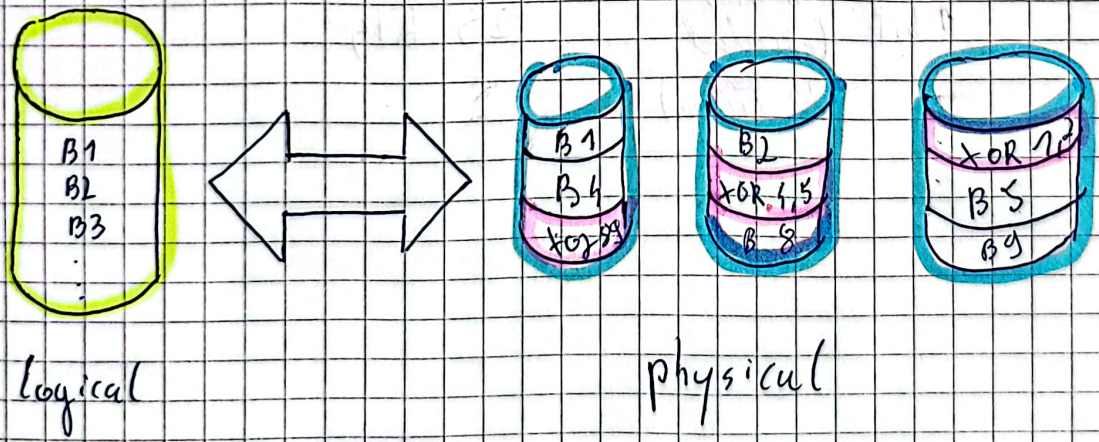
\+ fault-tolerant, high speed, disk space usage

\- unequal load balance, long data recovery

RAID 5

Remedy 4 RAID 3&4 problem, which is intensive usage of parity disk, causing higher probability of failure

Raid5 distributes parity stripes over all physical disks, which enhances speed & equalizes disk load



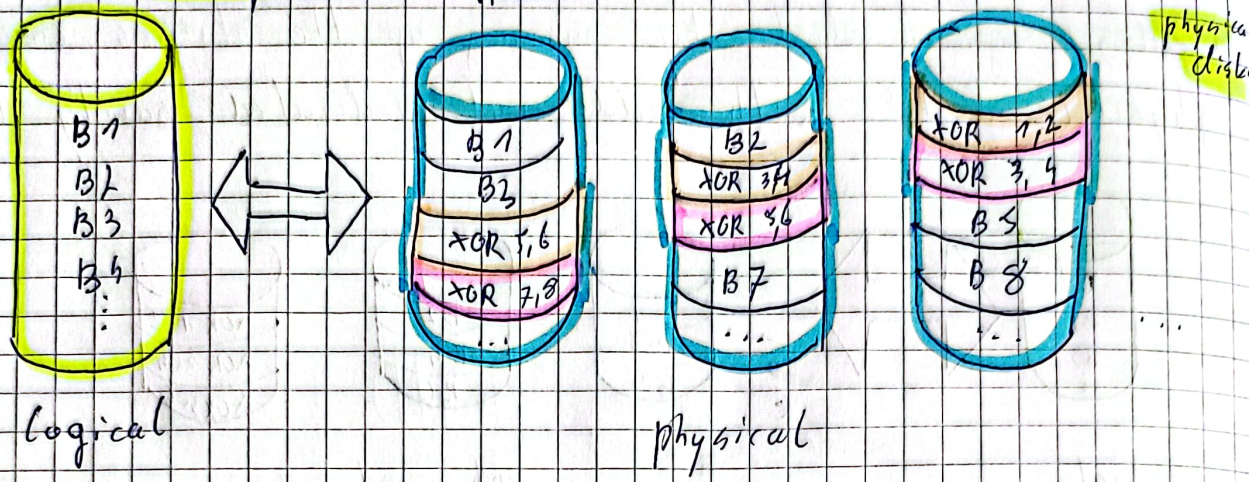
logical physical

\+ speed, disk space usage, load balance

\- long & unsafe data recovery after 1-drive failure

RAID 6

uses 2 parity bits & offers data recovery in case of sim. fail. of 2 physical disks



\+ data safety during recovery after 1-drive failure

\- disk space usage, min. n+2 disks needed

IEEE Floating Point Standard

$$(-1)^s * 1.f * 2^{e-127}$$