university of groningen

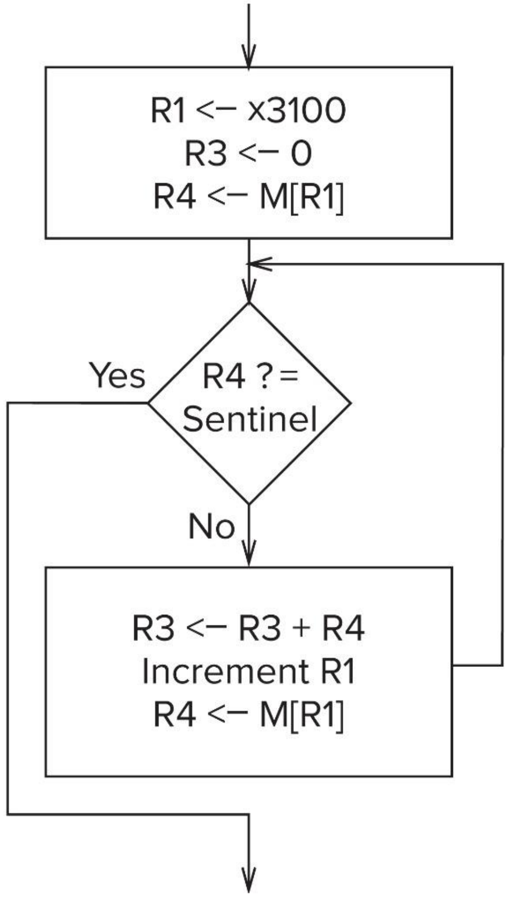
Computer Architecture 2023-24 (WBCS010-05)

Lecture 7: The LC-3 (Chapters 5)

Reza Hassanpour
r.zare.hassanpour@rug.nl

Example: Loop with Sentinel

› Compute the sum of integers starting at x3100, until a negative integer is found. Instructions start at x3000



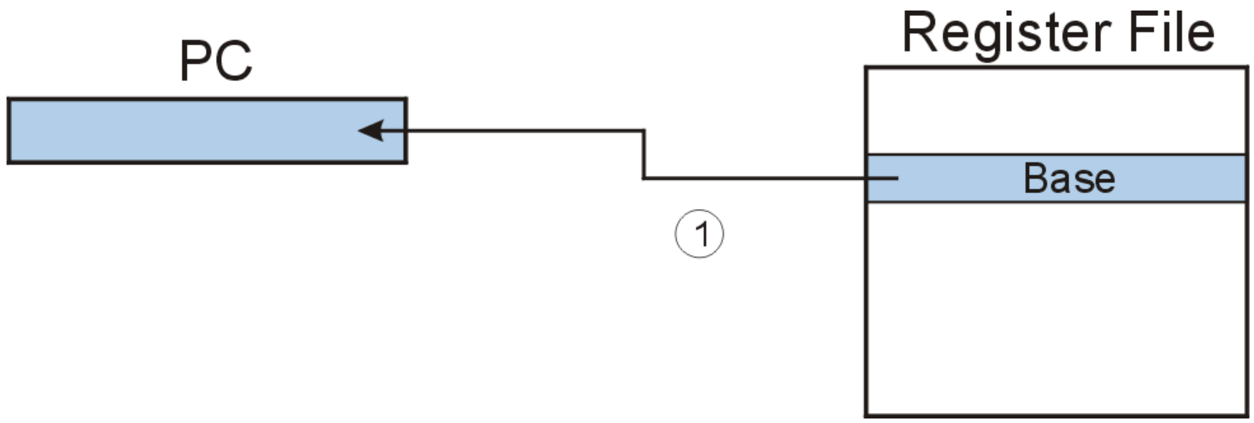
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	R1<- x3100
x3001	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	R3 <- 0
x3002	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	R4 <- M[R1]
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	BRn x3008
x3004	0	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0	R3 <- R3+R4
x3005	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1 <- R1+1
x3006	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	R4 <- M[R1]
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	1	BRnzp x3003

› In this case, it's the data that tells when to exit the loop. When the value loaded into R4 is negative, take branch to x3008

Unconditional Branch (JMP)

- We can create a branch that is always taken by setting n, z, and p to 1
- However, the target address of BR is limited by the 9-bit PC offset
- The JMP instruction provides an unconditional branch to any target location by using the contents of a register as the target address. It simply copies the register into the PC

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	1	1	0	0	0	0	0	Base			0	0	0	0	0	0



TRAP: Invoke a System Service Routine

› The TRAP instruction is used to give control to the operating system to perform a task that user code is not allowed to do. The details will be explained in Chapter 9

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							

› For now, you just need to know that bits [7:0] hold a "trap vector" -- a unique code that specifies the service routine. The service routines used in this part of the course are:

trapvector	service routine
x23	Input a character from the keyboard
x21	Output a character to the monitor
x25	Halt the processor

Input / Output Service Routines

› Getting character input from the keyboard

- TRAP x23 is used to invoke the keyboard input service routine.
- When the OS returns control to our program, the **ASCII code** for the key pressed by the user will be **in R0**.

› Sending character output to the monitor

- TRAP x21 is used to invoke the monitor output service routine.
- **Before** invoking the routine, put the **ASCII character** to be output **into R0**.

Example Program: Count Occurrences of a Character

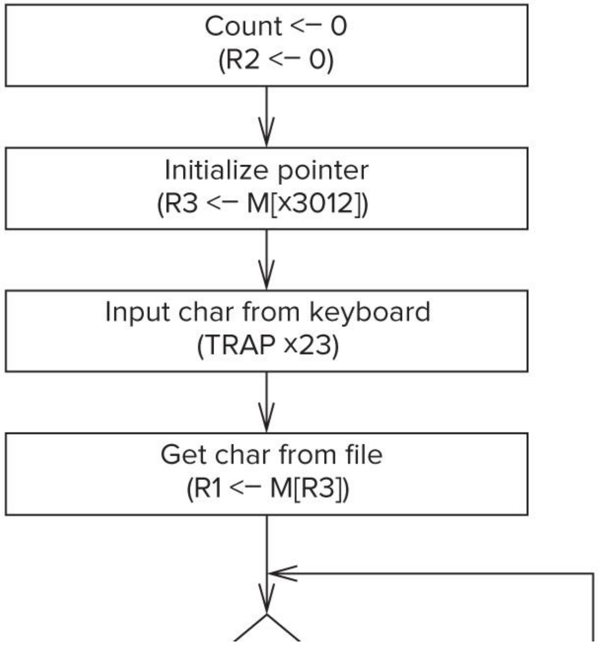
› We want to count the number of times a user-specified character appears in a text file, and then print the count to the monitor

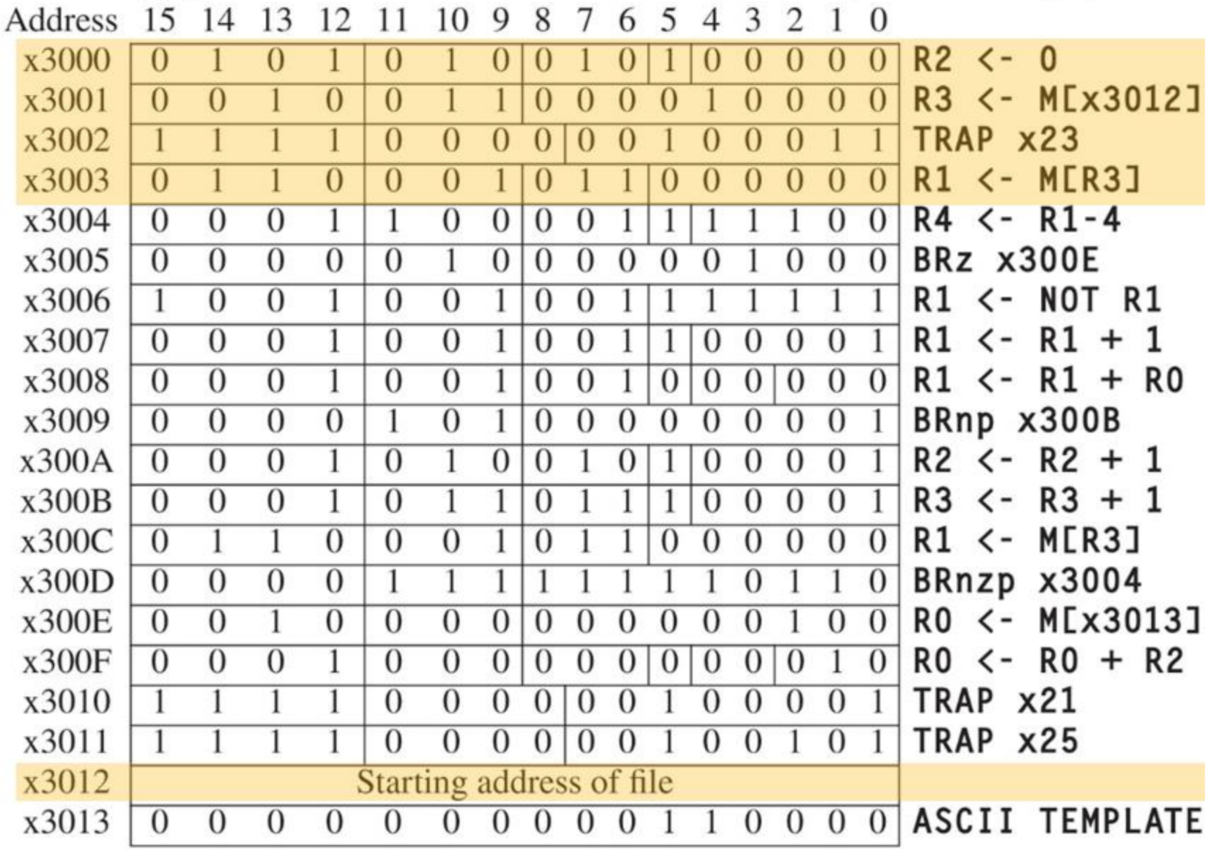
- The text file is stored in memory as a sequence of ASCII characters
- The end of the file is denoted with the **EOT**[*] character (x04). NOTE: We do not know how many characters are in the file; EOT is the sentinel value that signals when we are done
- A pointer to the file will be stored at the end of the program. (A "pointer" is a memory address; it will be the address of the first character in the file)
- We will assume that the character will appear no more than 9 times
- Program instructions will start at x3000. The file data can be anywhere in memory

* *End Of Transmission*

university of groningen

Part 1: Initializing Registers



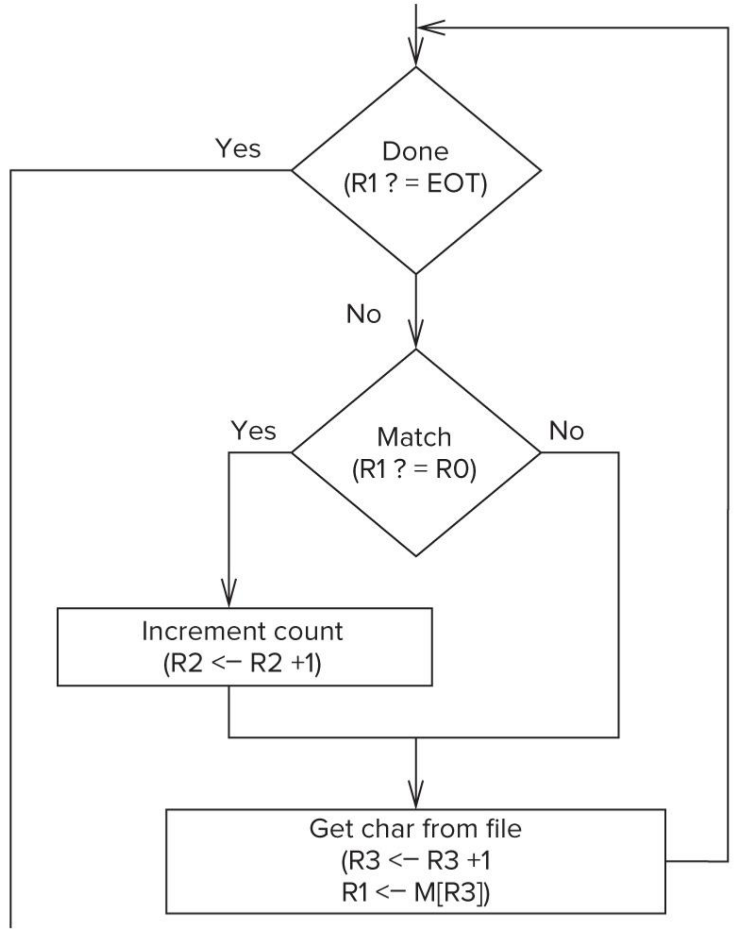


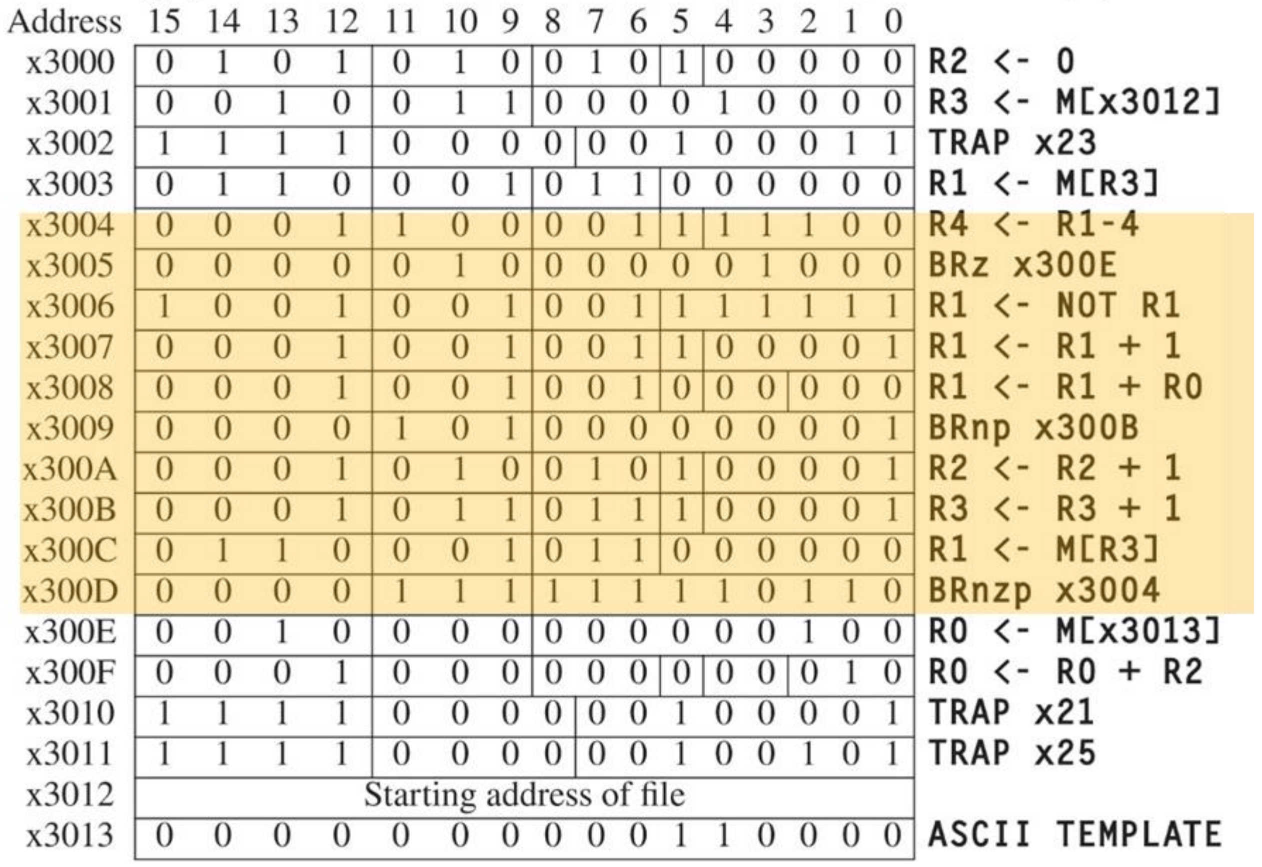
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 <- 0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	R3 <- M[x3012]
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	TRAP x23
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 <- M[R3]
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0	R4 <- R1-4
x3005	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	BRz x300E
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 <- NOT R1
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1 <- R1 + 1
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	R1 <- R1 + R0
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	BRnp x300B
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2 <- R2 + 1
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	R3 <- R3 + 1
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 <- M[R3]
x300D	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	0	BRnzp x3004
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	R0 <- M[x3013]
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	R0 <- R0 + R2
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	TRAP x21
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	TRAP x25
x3012	Starting address of file																
x3013	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	ASCII TEMPLATE

R2 is counter. R3 is address of first character to read from file.
R0 is character from keyboard. R1 is first character from file.

university of groningen

Part 2: Read Characters and Count

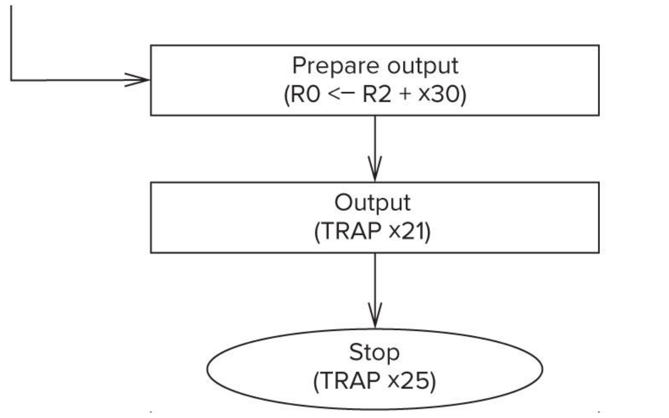


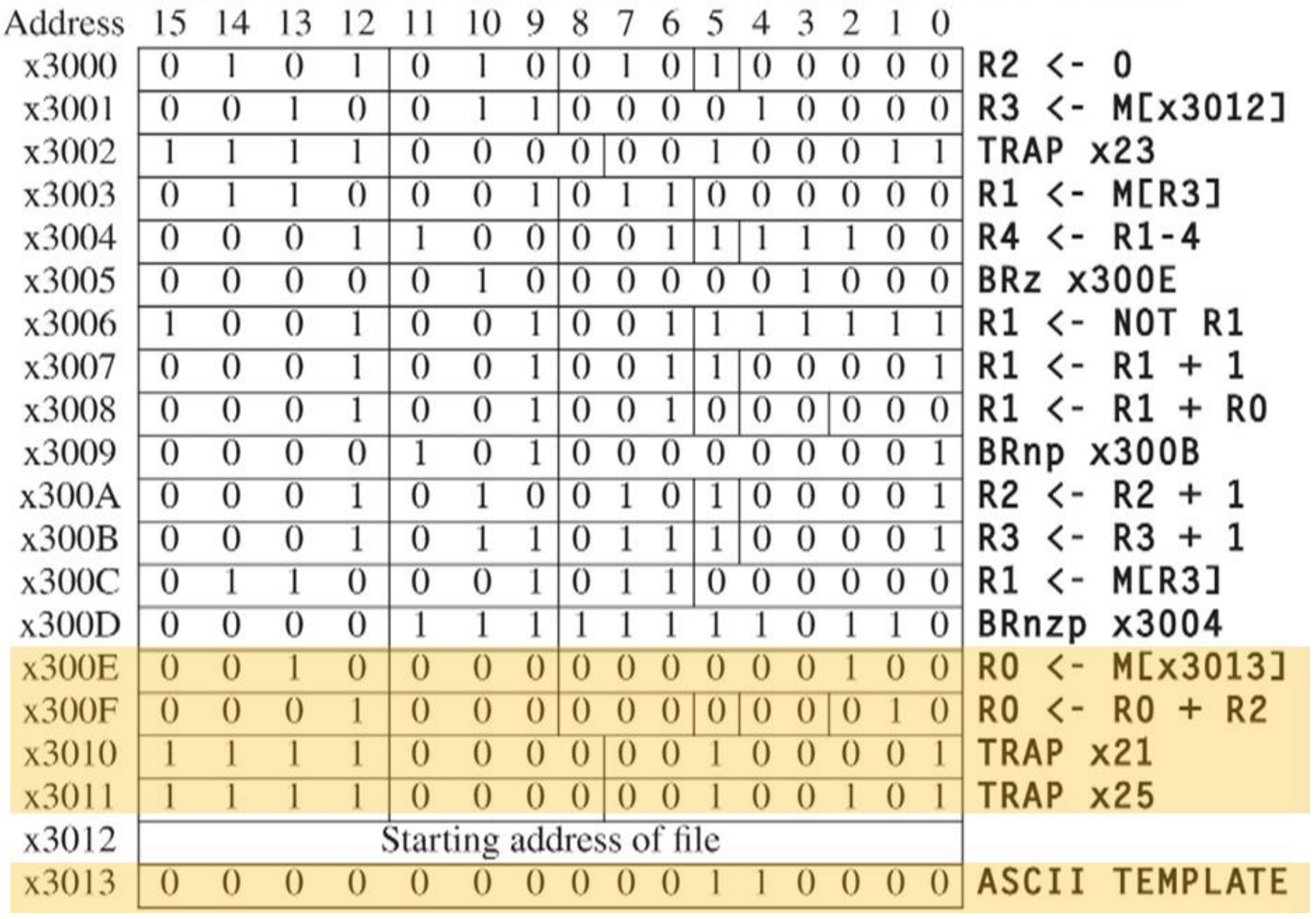


Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 <- 0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	R3 <- M[x3012]
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	TRAP x23
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 <- M[R3]
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0	R4 <- R1-4
x3005	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	BRz x300E
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 <- NOT R1
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1 <- R1 + 1
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	R1 <- R1 + R0
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	BRnp x300B
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2 <- R2 + 1
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	R3 <- R3 + 1
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 <- M[R3]
x300D	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	0	BRnzp x3004
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	R0 <- M[x3013]
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	R0 <- R0 + R2
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	TRAP x21
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	TRAP x25
x3012	Starting address of file																
x3013	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	ASCII TEMPLATE

Compare character to immediate x04 (EOT in ASCII). If equal, exit loop. Otherwise, count if matches user input and read the next character.

Part 3: Output Count and HALT





Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 <- 0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	R3 <- M[x3012]
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	TRAP x23
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 <- M[R3]
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0	R4 <- R1-4
x3005	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	BRz x300E
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 <- NOT R1
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1 <- R1 + 1
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	R1 <- R1 + R0
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	BRnp x300B
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2 <- R2 + 1
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	R3 <- R3 + 1
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 <- M[R3]
x300D	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	0	BRnzp x3004
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	R0 <- M[x3013]
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	R0 <- R0 + R2
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	TRAP x21
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	TRAP x25
x3012	Starting address of file																
x3013	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	ASCII TEMPLATE

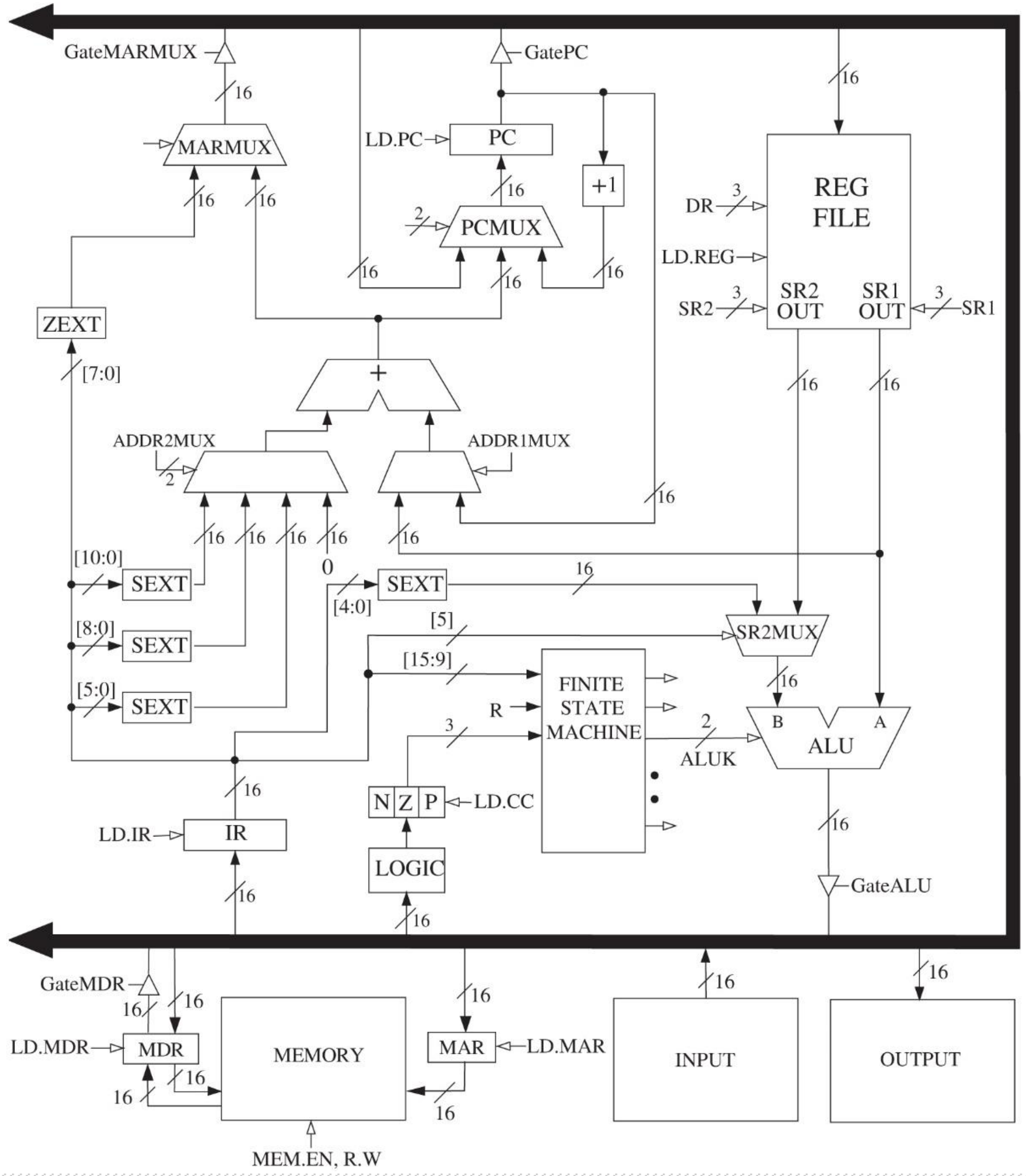
When loop is finished, convert count (R2) to the corresponding ASCII character by adding '0' (x30). Output the character and halt the program

university of groningen

ASCII Codes

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

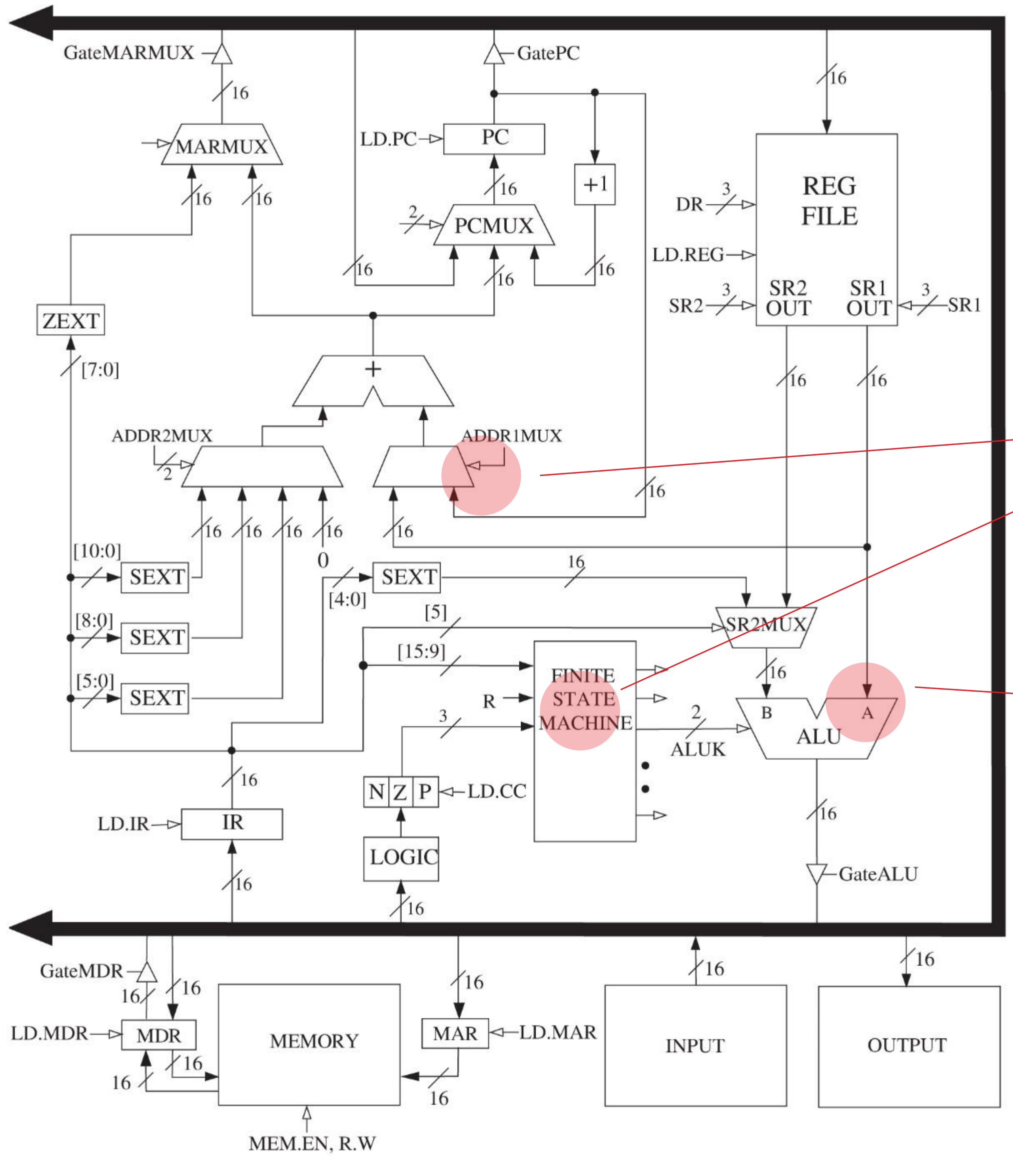
LC-3 Data Path 1



- › Data path is used to execute LC-3 programs
- › PC is initialized to point to the first instruction. Clock is enabled, and the control unit takes over
- › Next slides will give a little more detail on various components

university of
groningen

LC-3 Data Path 2

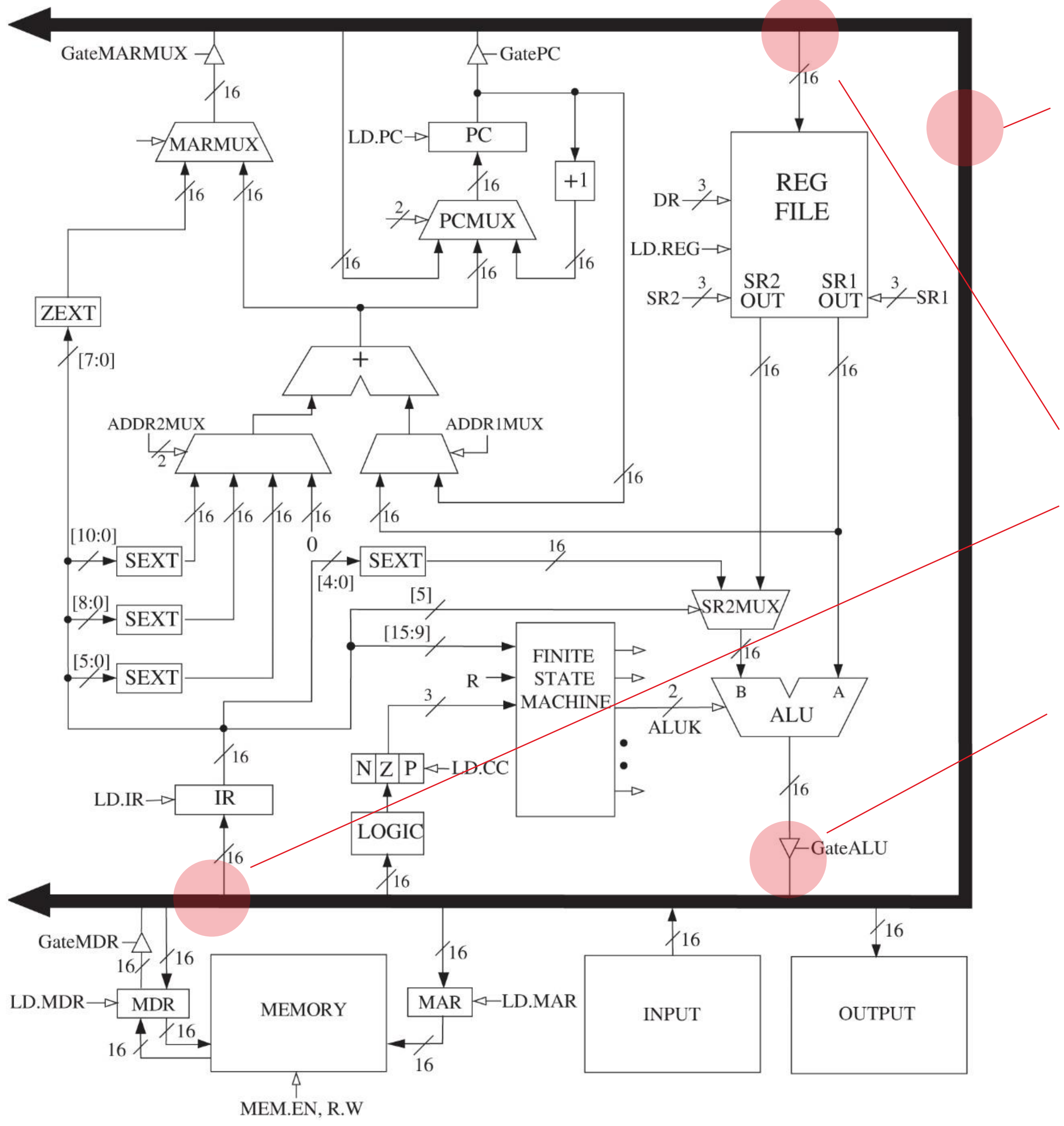


Arrows with open heads represent control signals from FSM

Arrows with filled heads represent data that is processed

university of groningen

LC-3 Data Path 3



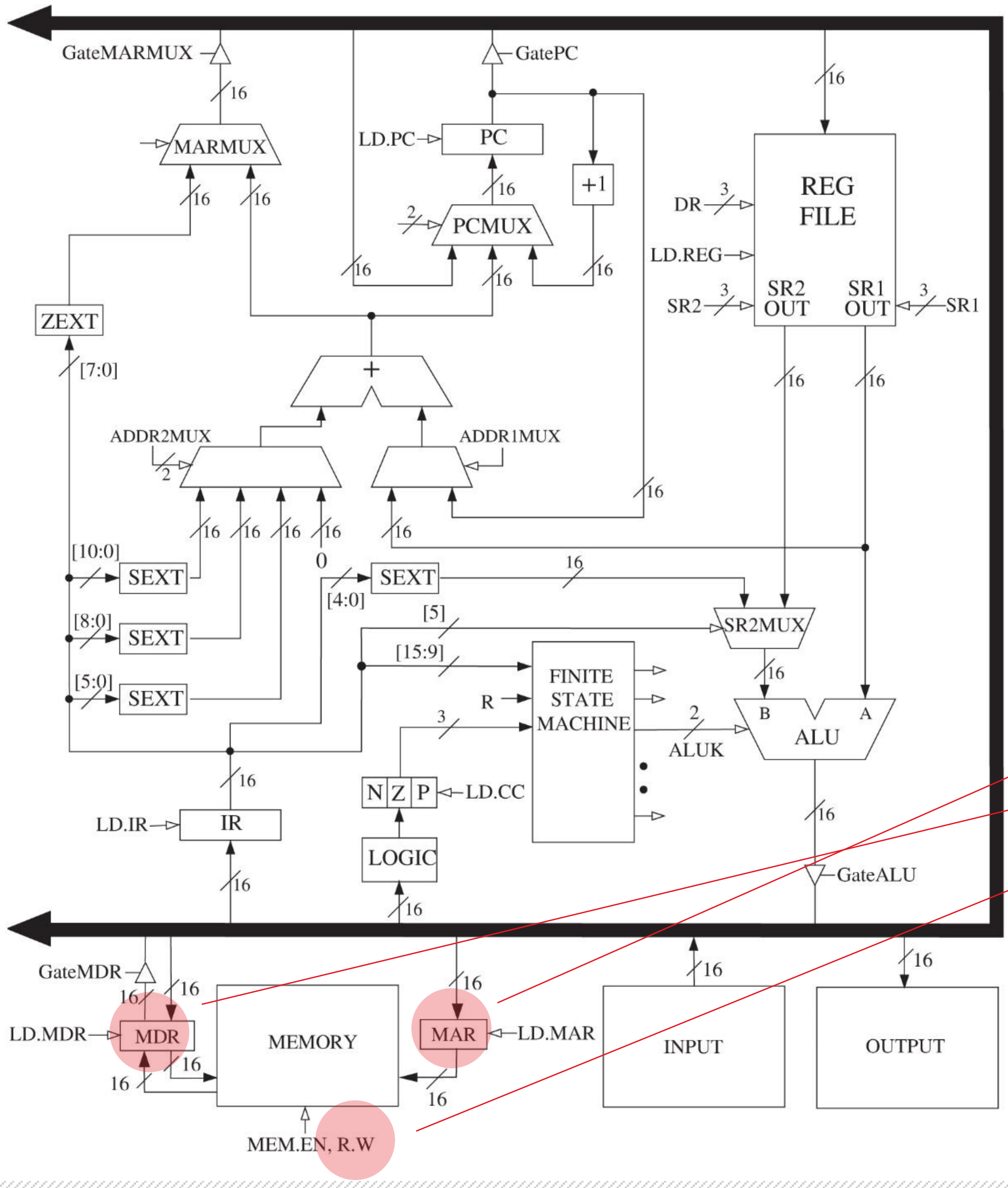
Global bus is a set of wires that allow various components to transfer 16-bit data to other components.

One or more components may read data from the bus on any cycle.

Tri-state device determines which component puts data on the bus. <u>Only one source</u> of data at any time.

university of groningen

LC-3 Data Path 4



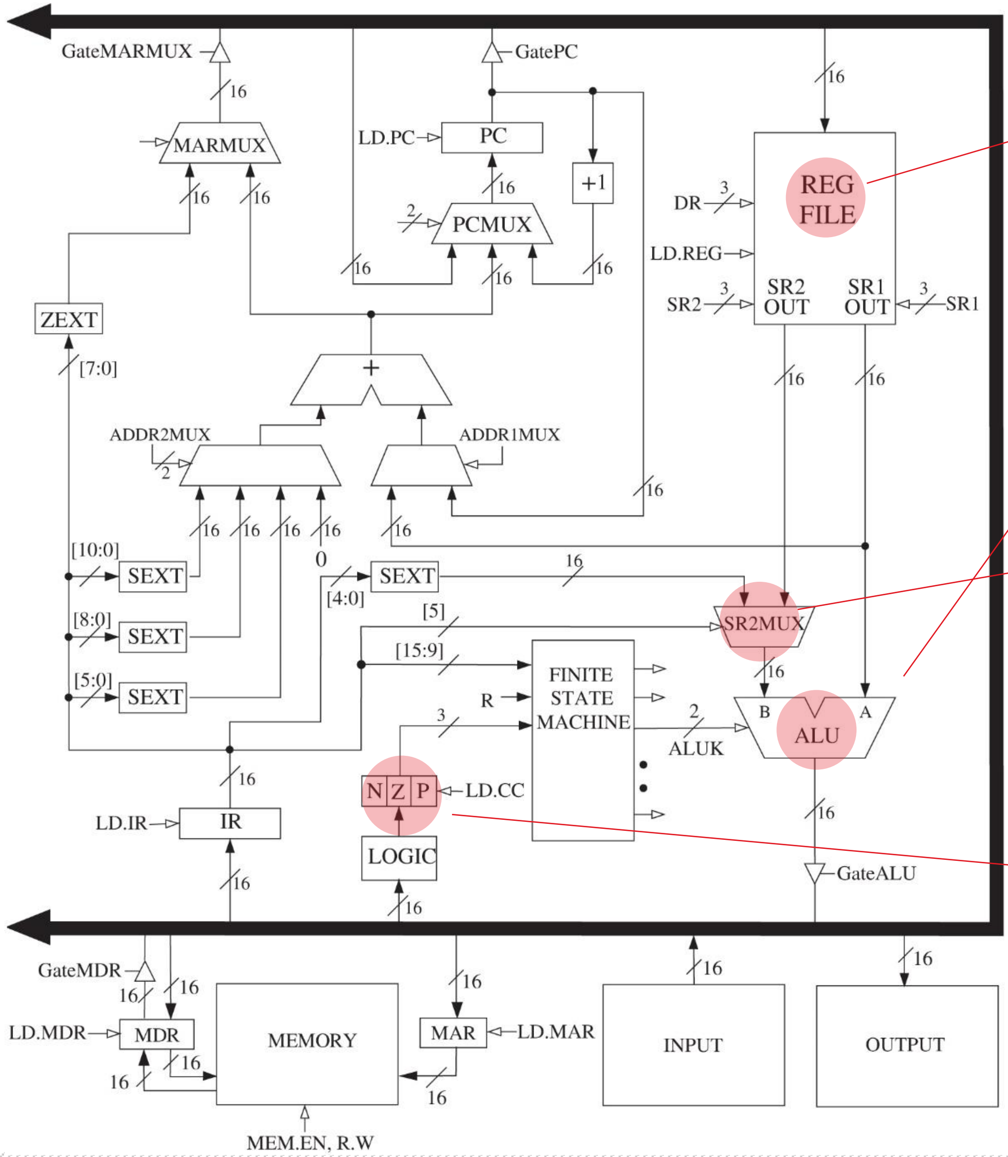
Memory interface:

MAR

MDR

Read/Write control

LC-3 Data Path 5

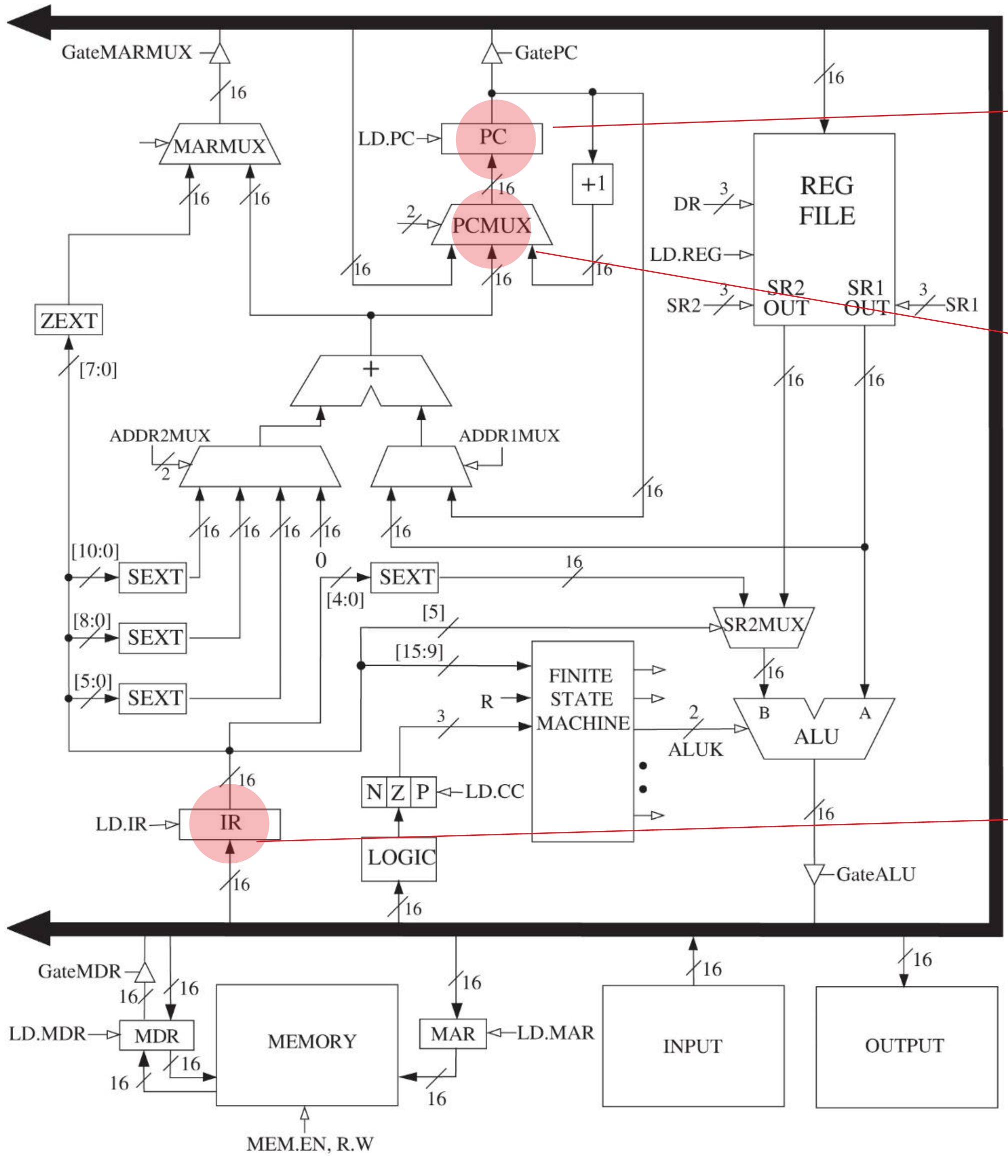


Register File (R0-R7)
Control signals specify two source register (SR1, SR2) and one destination (DR).

ALU performs ADD, AND, NOT.
Operand A always comes from register file. Operand B is from register file or IR. Output goes to bus, to be written into register file.

Condition codes are set by looking at data placed on the bus by ALU or memory (MDR).

LC-3 Data Path 6

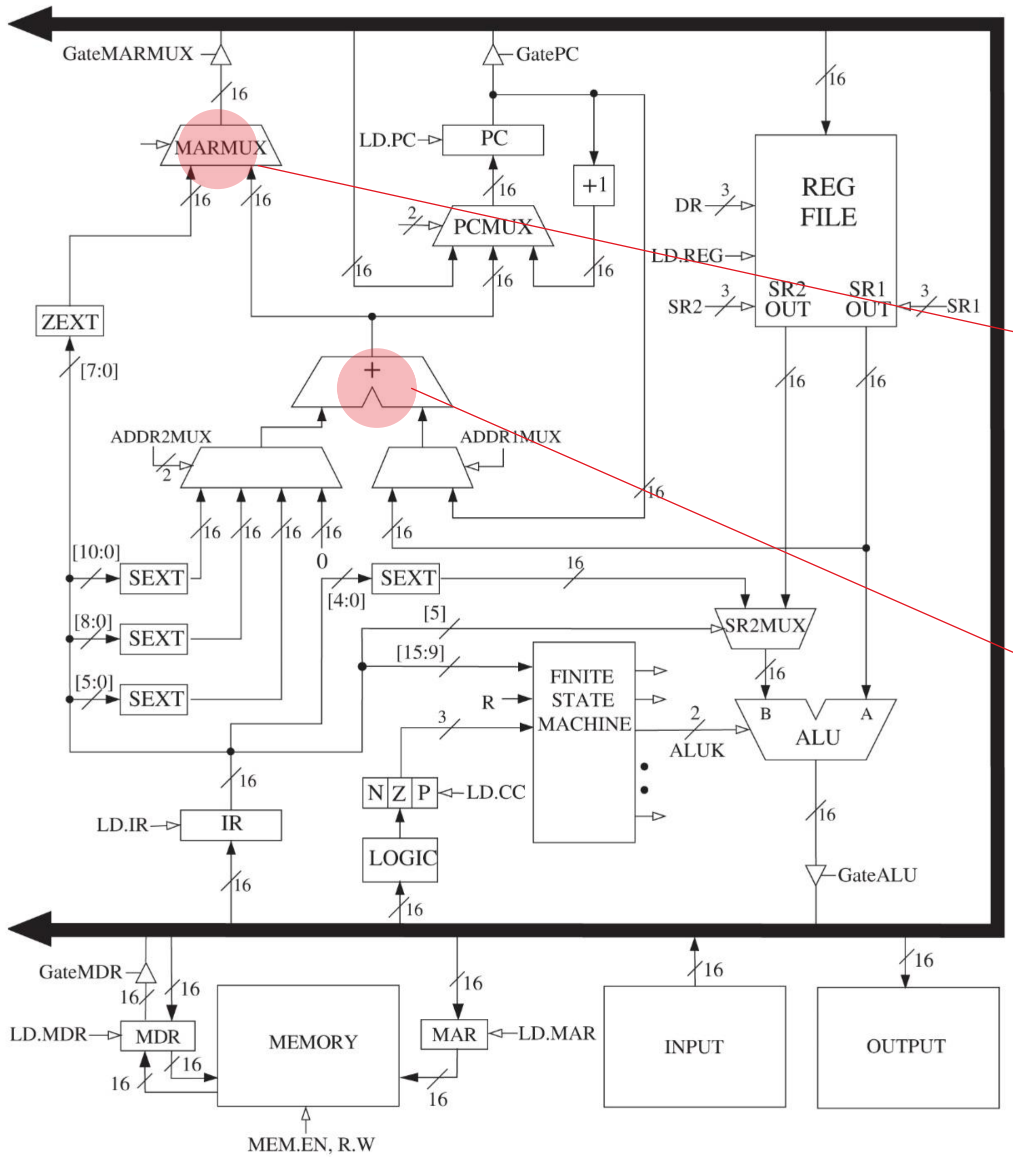


PC puts address on bus. Placed in MAR during Fetch

PCMUX allows various values to be written to PC: incremented PC (Fetch), computed address (BR), or register data from bus (JMP)

IR gets data from bus (MDR) during Fetch

LC-3 Data Path 7



MARMUX chooses value to be written to MAR during load, store, or TRAP

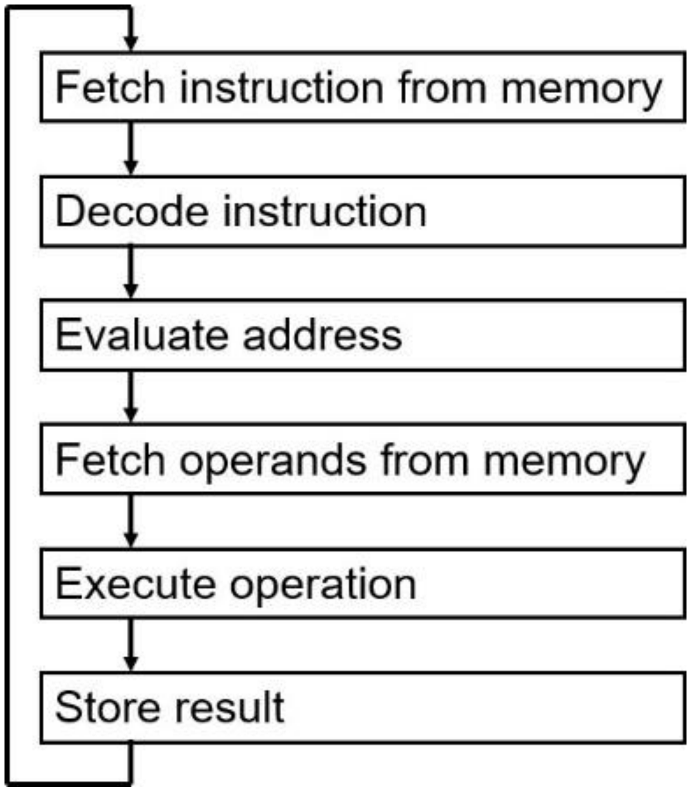
Evaluate Address phase adds offset to PC or register for load, store, BR

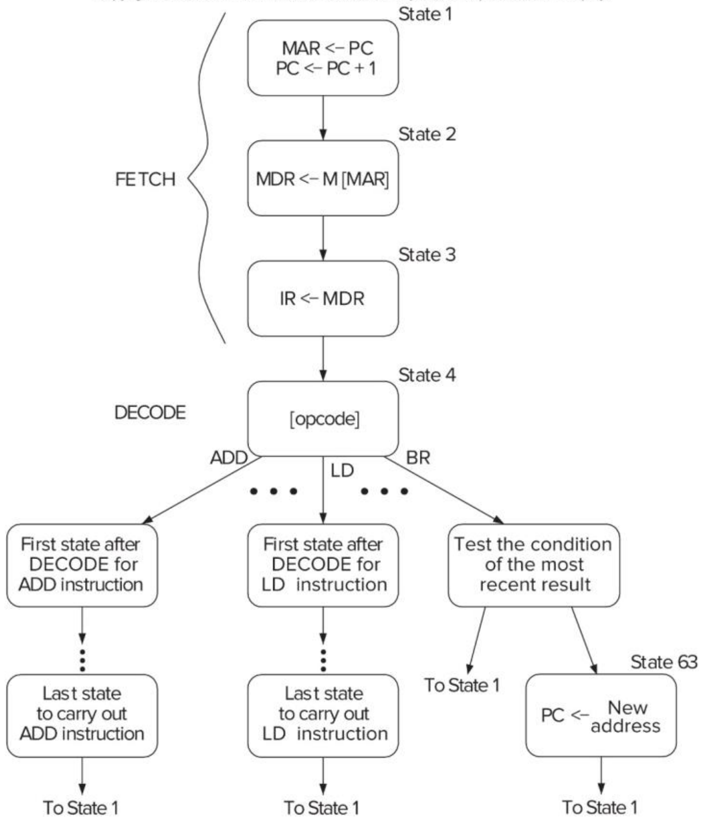
Optimizing the Performance

- Memory access (Load/Store) operations are slow compared to the execution time in the CPU.
- For better performance,
  - Access memory in advance
  - Use faster memories

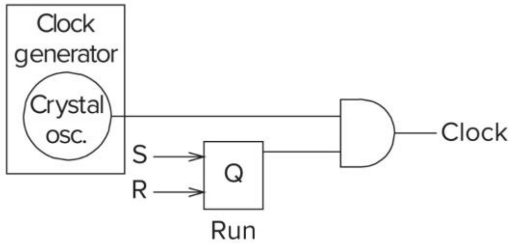
The remaining slides of this lecture are not from the test books

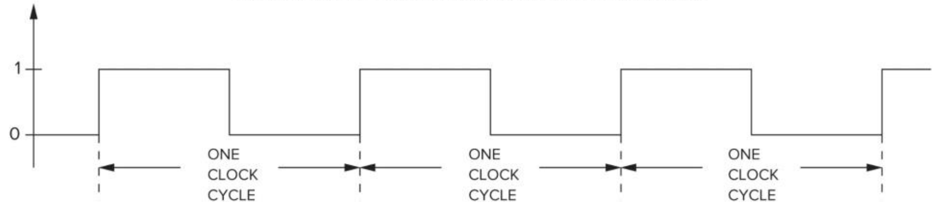
Machine cycle is not instruction cycle




State 1
MAR <- PC
PC <- PC + 1
State 2
FETCH
MDR <- M [MAR]
State 3
IR <- MDR
State 4
DECODE
[opcode]
ADD
LD
BR
First state after DECODE for ADD instruction
First state after DECODE for LD instruction
Test the condition of the most recent result
State 63
To State 1
PC <- New address
Last state to carry out ADD instruction
Last state to carry out LD instruction
To State 1
To State 1
To State 1

T (period) [s]
f(frequency) [Hz]
T=1/f
2 ns ➝ 500MHz





On instruction complexity

- Originally computers had extremely basic instructions
- Later more complex e.g., *floating point* instructions

- Ever bigger sets of hardware-based instructions due to:
  - Instruction compatibility requirements
  - Rising cost of software development

- Interpreted instruction sets on low cost computers
  - Easier to fix, ability to add new instructions, efficient development of complex instructions

RISC vs CISC

- **pre 1980s:** Complex instructions were used in an attempt to bridge the “sematic gap”
- **in 1980s:** RISC (Reduced vs Complex Instruction Set Computer) counter movement
- Emphasis on **faster issuing** of instructions
- **1990s and onward:** hybrid models starting from Intel’s 486 architecture
  - Common instructions are fast in RISC sense
  - Complex (but uncommon) instructions are slow but easier to implement

(RISC) design principles for modern computers

- All instructions must be directly **executed by hardware**
- **Maximize the rate** at which instructions are issued
- Instructions should be **easy to decode**
- Only **load and store** instructions should reference memory
- Provide **plenty of registers**

On parallelism

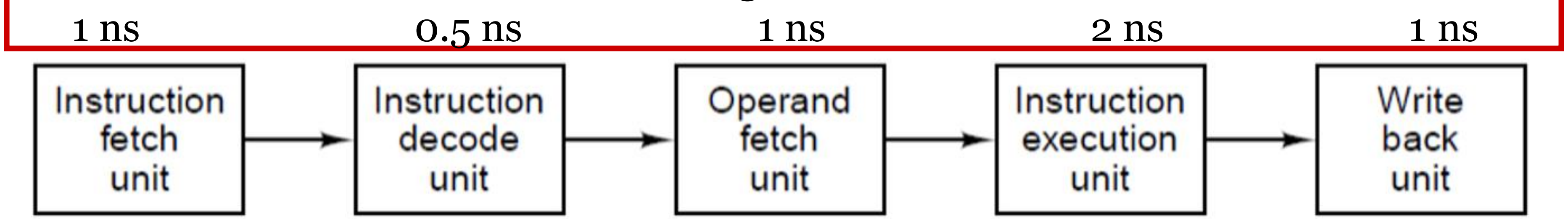
- **Instruction-level**
  - More instructions per second
  - **Pipelining** and **Superscalar architectures**

- **Processor-level**
  - Hard limit of information transfer (~20 cm/nsec)
  - Faster, larger chips ---> more heat
  - The number of transistors per unit area still increases
  - Adding more CPUs on a chip is the only way to improve
  - (*back to*) Data Parallel Computers, Multiprocessors, and Multicomputers (what about Cloud, e.g., AWS and Azure?)

Pipelining

- **Prefetch buffers** already available since 1959 →fetching & execution stages
- A **pipeline** divides instruction execution into many stages running in parallel, synchronised by the CPU clock
- Trade-off between latency and processor bandwidth (see next slide)

What can be observed about these timings:

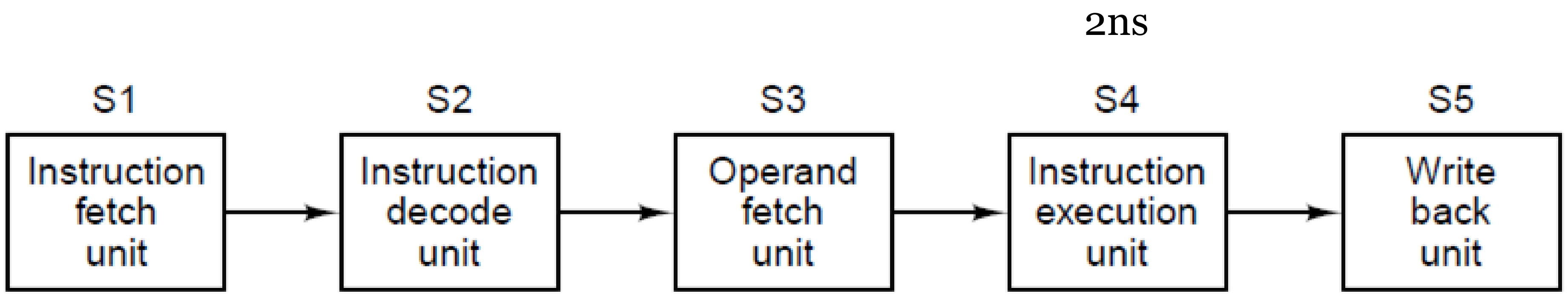


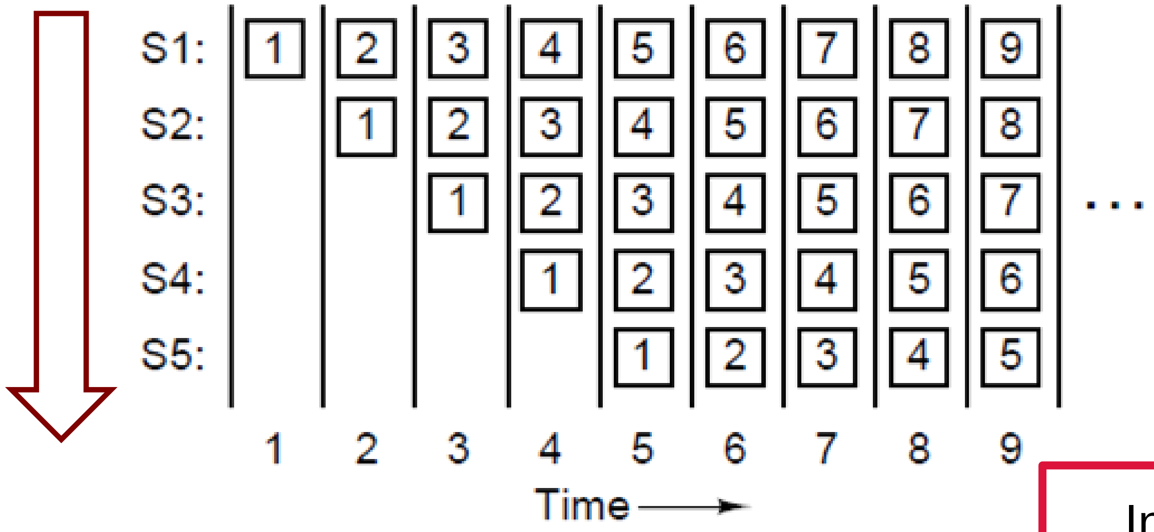
Assume the following implementation of a 5-stage instruction cycle

Pipelining metrics

- Cycle time *T* (in ns), number of stages *n*
- **Latency** = *n*T* (in ns)
  - Execution time for each instruction in nanoseconds
- **Bandwidth** = *1/T* (expressed in Million IPS: Instructions per second)
  - How many MIPS the CPU is capable of delivering

Pipelining example (5 stages)



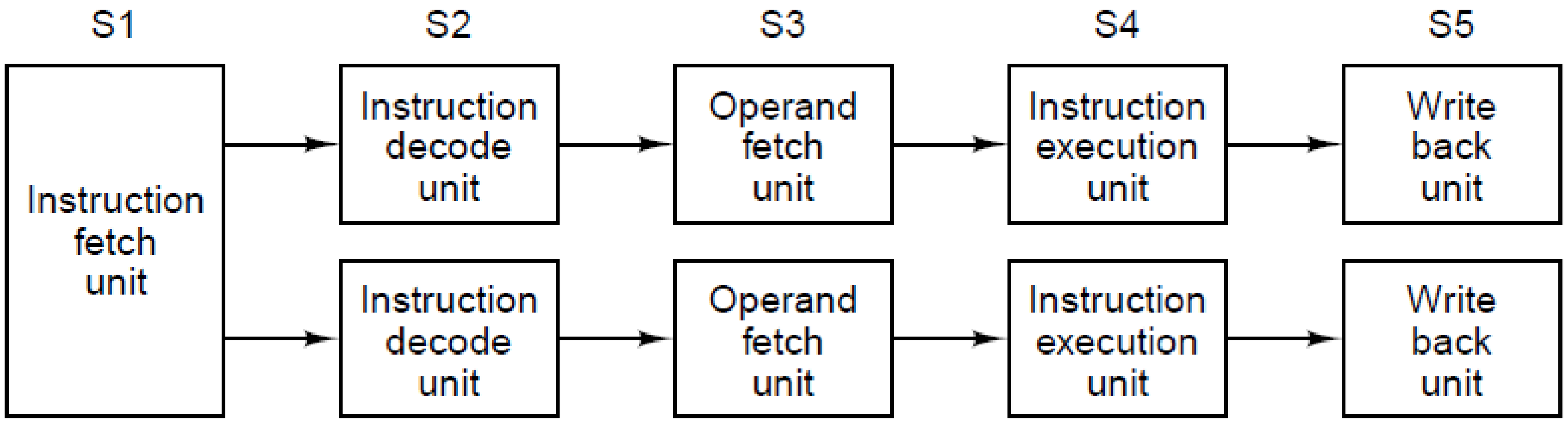


- T = 2 ns, n = 5
- Latency = n*T = 10 ns
- Bandwidth = 1/T = $1 / 2 * 10^{-9} = 500 * 10^{6}$ IPS = 500 MIPS

Intel Pentium Pro:
541 MIPS at 200 MHz
(1996)

2 ns → 500MHz

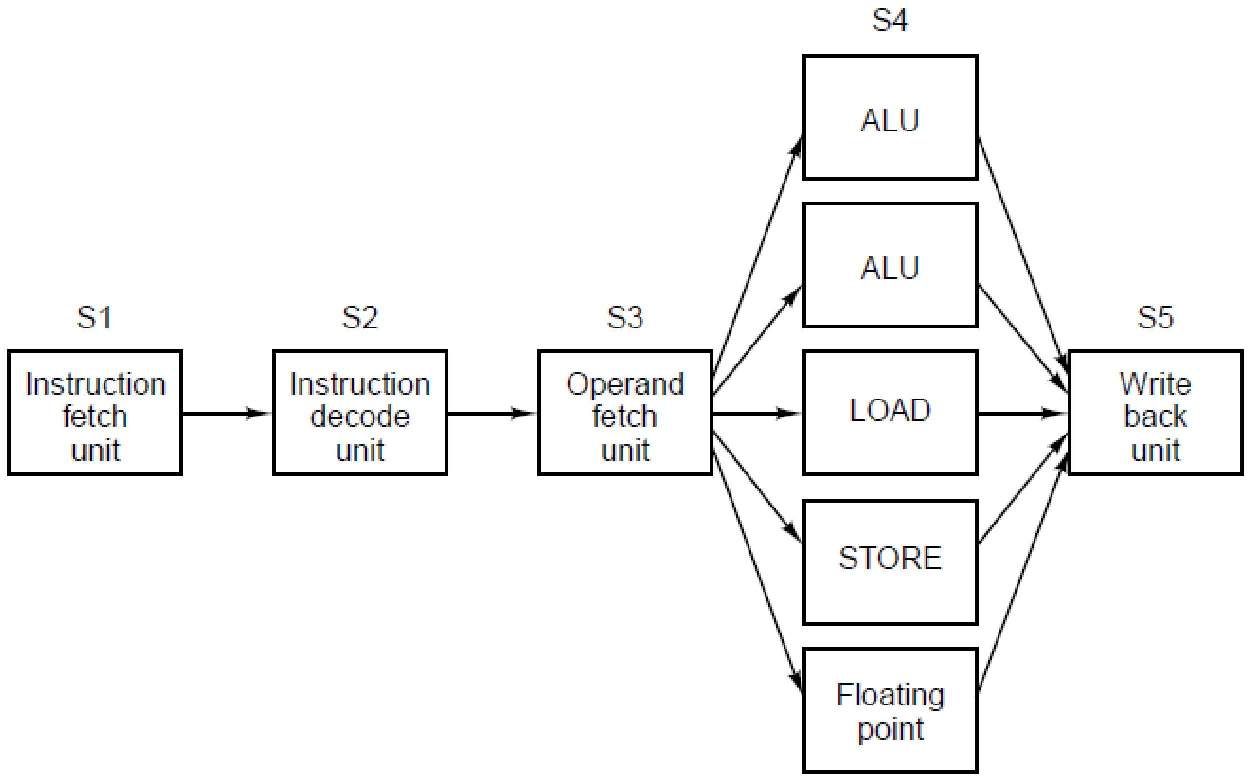
Multiple pipelines



- Conflict prevention happens at the compiler (software) level
- Different purposes of pipelines also possible
  - Intel’s Pentium architecture: u/v (regular/integer arithmetic-specific) pipelines

Superscalar architectures

- **Key observation**: S4 (instruction execution) may take much longer than S3 (operand fetching) → multiple execution units can speed up execution time



What about cache memories

- Main issue with computer Main memories: designed for capacity, **not for speed**
  - CPUs are getting faster at a faster pace
  - More cores on the CPU means more Main Memory refs
  - Moving (more) memory inside the CPU increases construction cost

Cache memory motivation

- Possible solutions

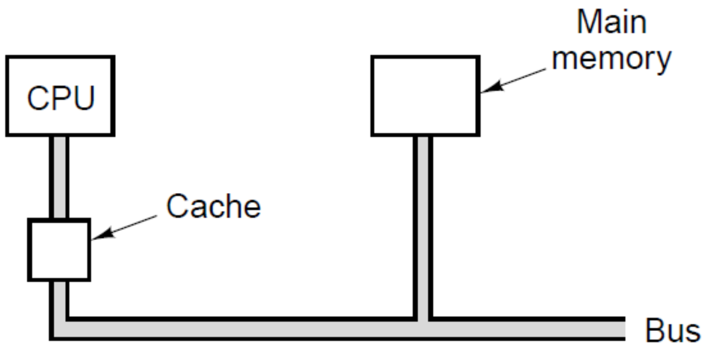
1) Issue read to Main memory and continue execution until word is needed; then stall (too many wasted cycles)

2) Issue reads in advance through compiler optimization (ends up in a software stall)

3) Add intermediate memory level of fast (=expensive) memory as close to CPU as possible – called **cache**

Cache memory basics

- Most heavily used memory words to be kept in cache
- **Principle of locality**
  - If address A is read, then *most probably* A+1 will be read in the next time interval, or
  - Memory reads in a short time interval only use a small part of the (total) memory
- **Fixed size blocks** are transferred between cache and memory (called cache lines)



Cache quality

- Access times: c for the cache, m for main memory
  - Refer k times to address A: 1(st) reference to main memory, $k-1$ references to the cache
  - The cache hit ratio h is the proportion of cache references:
    $$h = (k-1)/k$$
  - Average access time: $(1-h)m + c$
  - As k increases, $h \to 1$ and m "vanishes" from the equation

Cache Access Methods

› How can we decide if a data block is in cache or not?

› Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines.

› There are three methods to access cache data:
- Direct Mapping
- Associative Mapping
- Set Associative Mapping

Direct Mapping

- It is the simplex technique, maps each block of main memory into only one possible cache line i.e. a given main memory block can be placed in one and only one place on cache.
- i = j modulo m,
- Where i = cache line number;
- j = main memory block number;
- m = number of lines in the cache

Example

- Assume memory is 64K Words of 4 bytes,
- 32 bit addressability, 16 address lines
- Cache is 512 words (512x32)
- To show each cache row we need 9 bits
- Remaining 7 bits of the address of the word in memory is used as tag

Direct Mapping Cache

	Data
x0000	1234
x0001	2735
x0002	7428
x0003	9082
x0402	0012

Memory

Tag (7bits)	Data
x00	1234
x02	0012

Cache

Associative Mapping

- Direct mapping has high performance only when the cache size is large. (Many blocks may be mapped to the same cache line)
- In associative mapping, the mapping of the main memory block can be done with any of the cache block.
- The tag field of the cache line includes the address of the block

Set-Associative Mapping

› It is a compromise between direct and associative mappings that exhibits the strength and reduces the disadvantages.

› Each line includes multiple blocks (hence set-associative)

› A block is mapped to a specific set. Inside a set, it can be mapped to any line.

› The most commonly set-associative cache is two-way set associative cache.

› Set associative mapping is more expensive but has better performance than other mappings

Example

- The original Pentium 4 processor had a four-way set associative L1 data cache of 8 KiB in size, with 64-byte cache blocks. Hence, there are 8 KiB / 64 = 128 cache blocks.
- Pentium 4 processor also had an eight-way set associative L2 integrated cache 256 KiB in size, with 128-byte cache blocks.

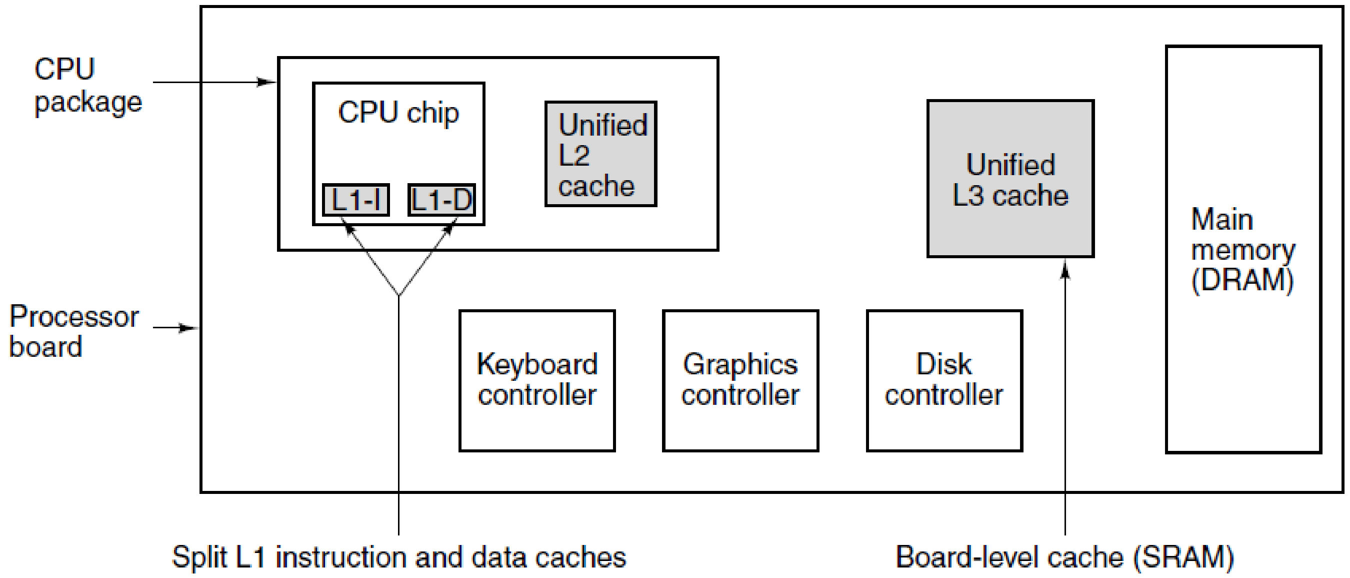
Cache design

- In practice: fixed-sized memory blocks (64, 32, 16 bytes) referred to as cache lines

- Design considerations:

1. Size: bigger = faster, but also more expensive
2. Line size: should we refresh many data or often?
3. Administration: how do we know what is kept inside the cache at any time?
4. Content: unified or split cache for data and instructions?
5. Topology: what is the optimal number of caches? How are they organized?
6. and many more concerning replacement, power, ...

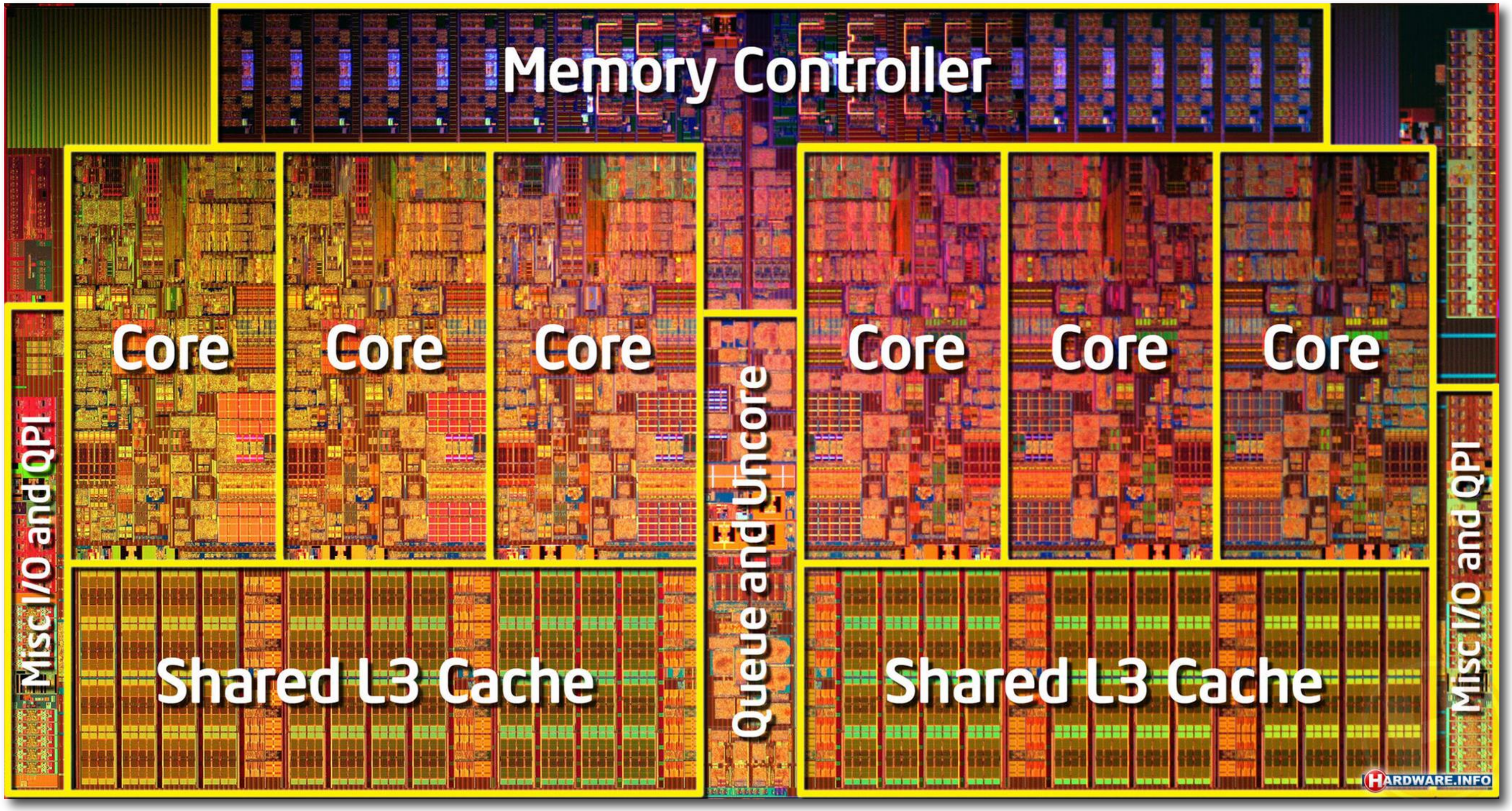
Cache memory – hierarchy



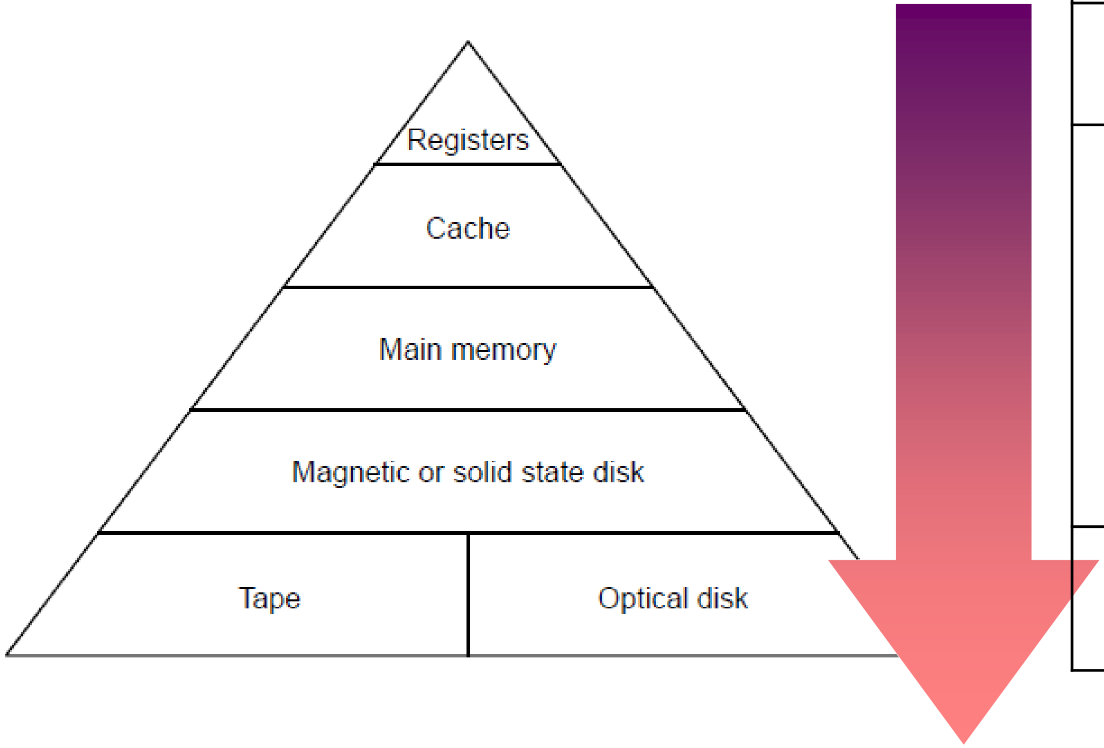
- Every cache level from L1 to L3 is bigger but also slower
- Locality principle: **spatial** (A will be followed by A+1) & **temporal** (A will be accessed again in the future)
- Typical cache line sizes of 4 to 64 Bytes

university of
groningen

Intel i7 Cache Memories



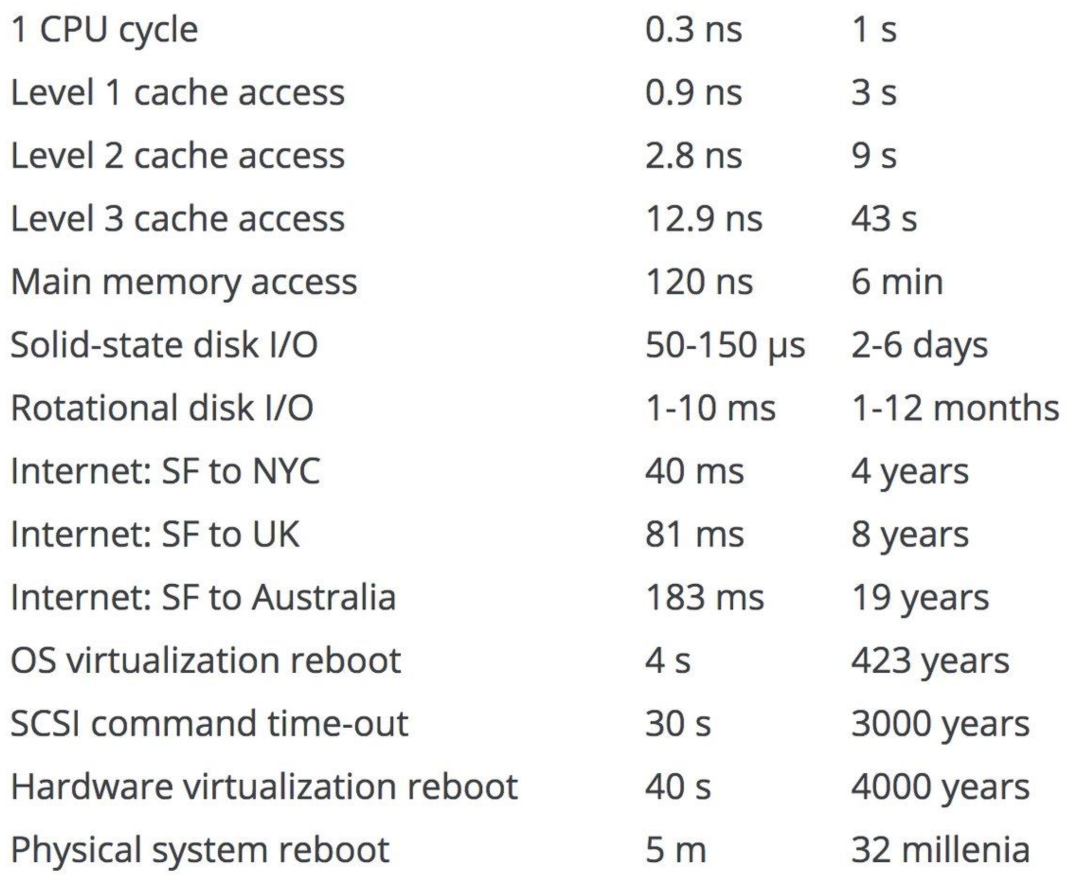
The memory hierarchy



Access time	Storage capacity	US$ per Bit
Less than nanosecs	Bytes	Dollars per MB
Seconds	(Unlimited)	Pennies per Gb

university of groningen

(memory) Latencies in perspective



1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
SCSI command time-out	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 m	32 millenia

https://twitter.com/srigi/status/917998817051541504?lang=en

(longer term note) importance of locality

› Which code has a better locality?

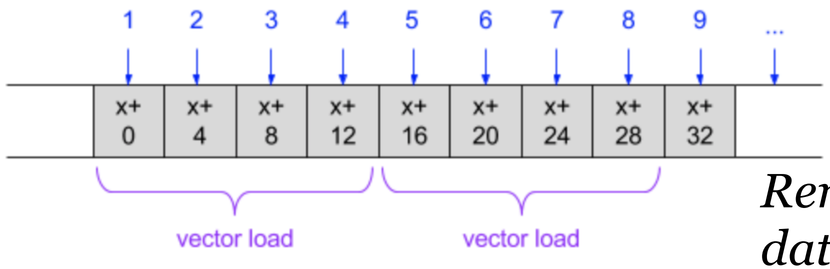
```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

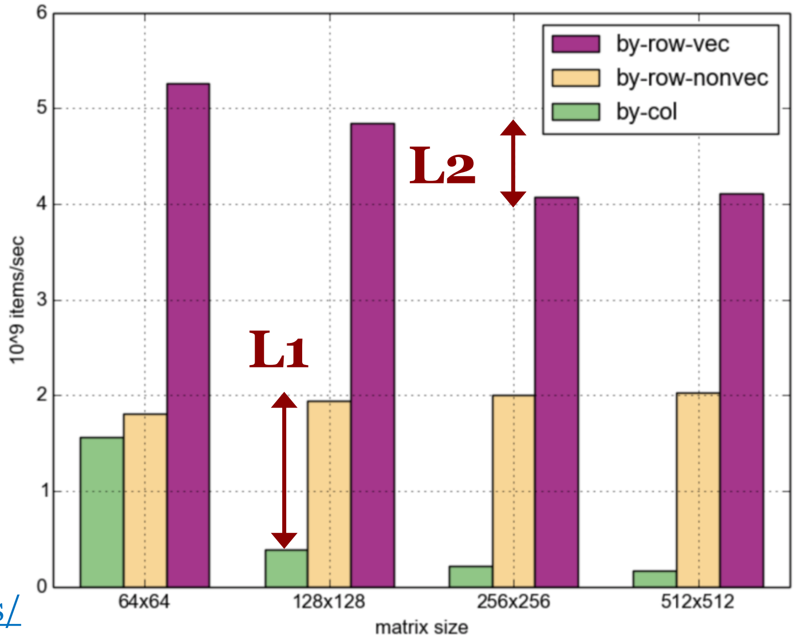
```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

› Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer



Remember to access data in order!



By-row is faster than by-column (6-8x, depends on the size)
64x64 matrix fits in 32-KB L1 cache (small difference by-row/by-col)
Larger matrix sizes make by-column go to L2 more frequently
Vectorized version beats the non-vectorized code by 2-3x
256x256 and up saturate (L2 (256KB) misses go to main memory)
https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays/

university of groningen

Questions?