Introduction to Computer

ES 2016/2017
Adam Belloum
a.s.z.belloum@uva.nl

Slides Sebastian Altmeyer (altmeyer@uva.nl)

Content

- How a processor works
- How instructions are processed
- What the following items mean:
  - Register and Gates
  - ALU
  - Program Counter
  - Instructions, Machine Code, Assembler
  - Pipelining
  - Caches

What is computer architecture?

(a) How to build a processor

- common understanding by computer architects

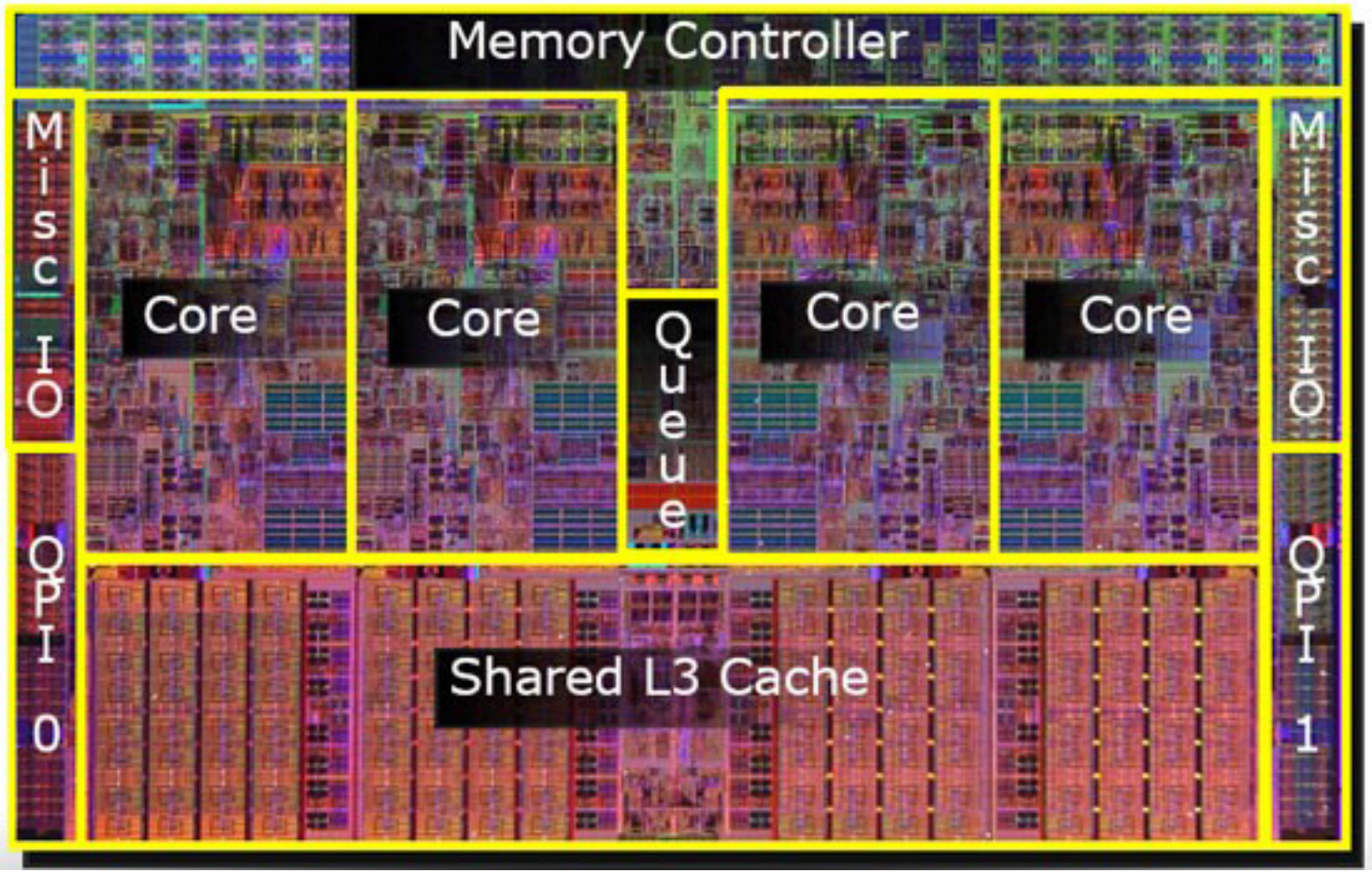
(b) How to build a computer

- common understanding by many others …

(c) How a computer system works

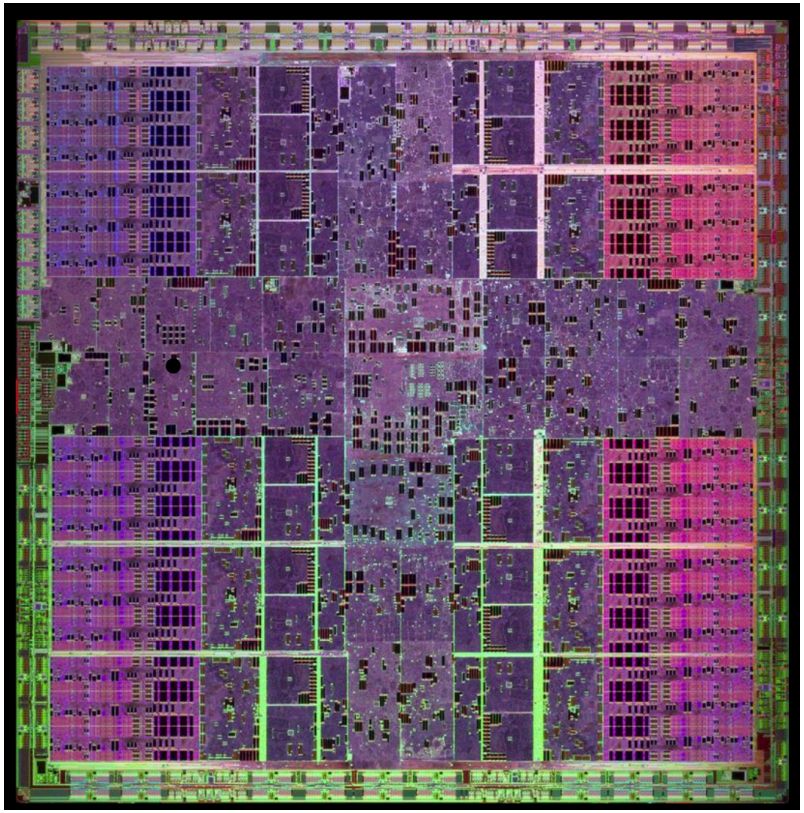
- Including all system layers.

Intel core i7



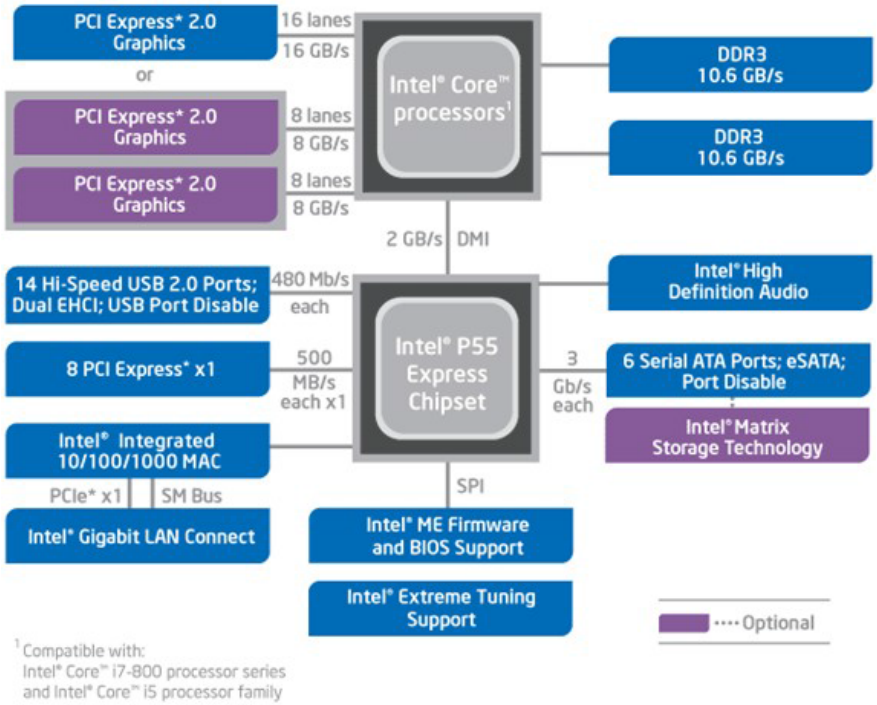
Quadcore with 731,000,000 transistors

GeForce GTX 280 (GT200)



240 stream processors with 1.4 billion transistors

… and it's natural habitat



Highly complex, but simple components

- **registers** (to store information)
- **gates** (to process information
- **buses** (to connect registers and gates)
- a **clock**

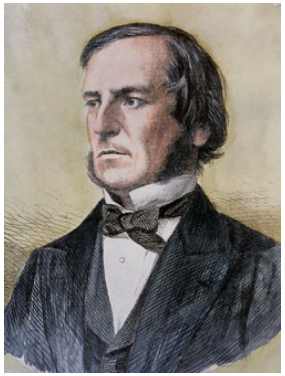
A processor can only flip 0 to 1 and vice versa
… but that is all we need

Gottfried Leibniz

Binary Numbers

George Boole

Boolean Algebra



Claude Shannon

Expressiveness of Relays

John von Neumann

First Processor

Leibniz: Binary Numbers

how to represent any **natural number** just using bits

$$b_{n-1}\, b_{n-2}\, b_{n-3} \ldots b_2\, b_1\, b_0 \equiv \sum_{i=0}^{n-1} b_i 2^i$$

- n denotes register/bus width, i.e. 32-bit architecture means n = 32

- Example (8 bits): 00101010 = **42**
- Range: **$[0 ; 2^{n-1}-1]$**

Two's Complement

how to represent a negative number?

$$b_{n-1}\, b_{n-2}\, b_{n-3} \ldots b_2\, b_1\, b_0 \equiv -\, b_{n-1}\, 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

- Example (8 bits):
  - 11010110 = - 42
  - 00101010 = 42

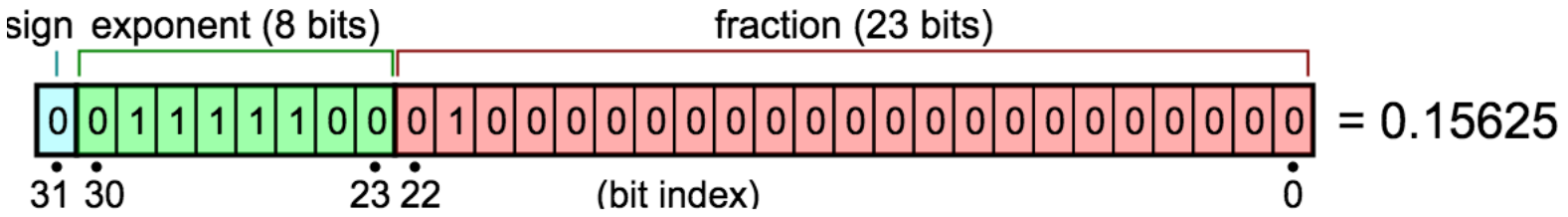
  - Range: $[-2^{n-2} ; 2^{n-2}-1]$

Floating Point (IEEE 754 single-precision 32-bit)

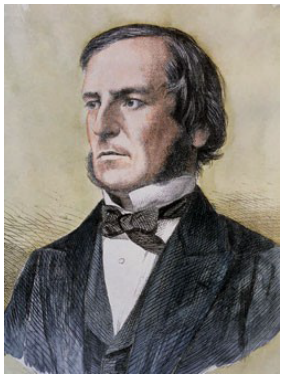
$b_{31}b_{30}b_{29}...b_{23}b_{22}b_{21}...b_0$

Sign Exponent Fraction

$$-1^{Sign}(Fraction)_2\ 2^{(Exponent)_2-127}$$



Boole: Boolean Algebra



- Variables

$$z, v, w \in \{0, 1\}$$

- Operators

$$Z = Not\ v = \overline{v}$$

$$Z = v \wedge w$$

$$Z = v \vee w$$

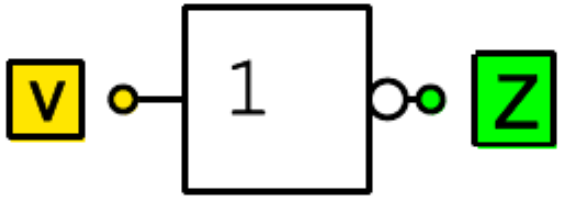
Claude Shannon

Has shown in his Master's Thesis:

"A **Symbolic Analysis** of **Relay** and **Switching** Circuits" how to implement **boolean logic** using electronic circuits

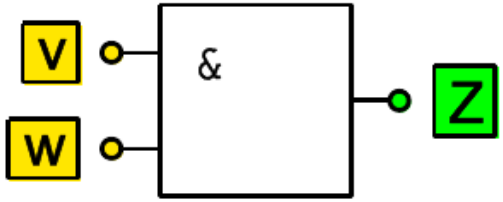
Boole and Shannon: Gates

$Z = Not\ v$



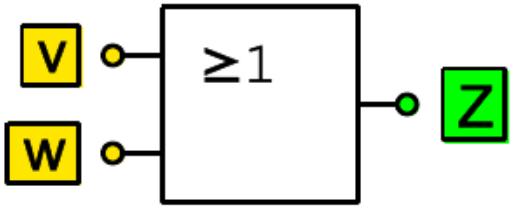
NOT

$Z = v \wedge w$



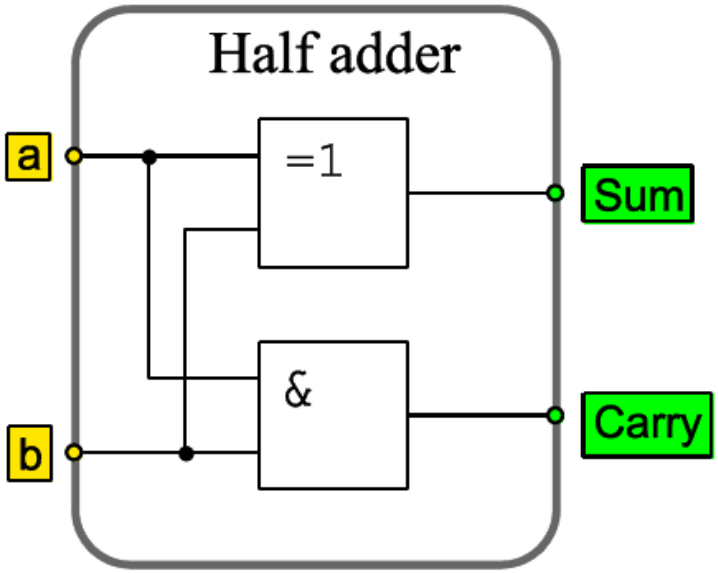
AND

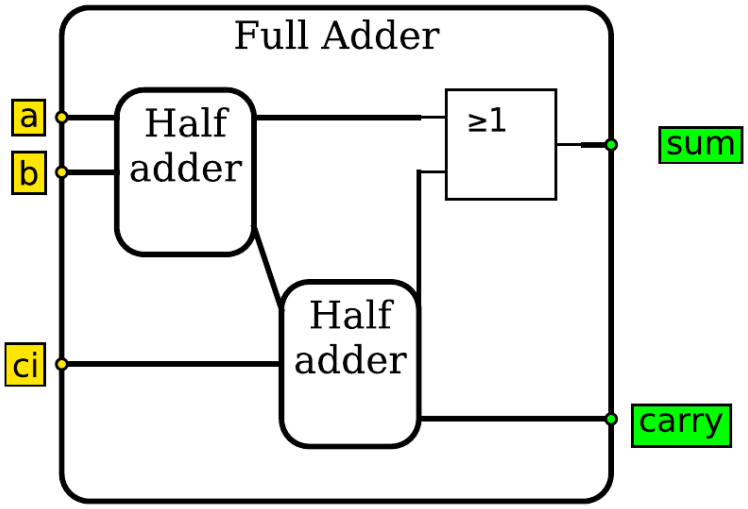
$Z = v \vee w$

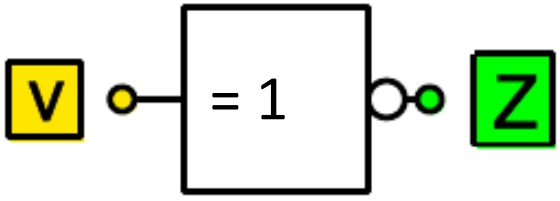


OR

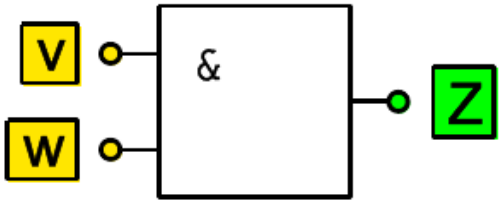
Half Adder/Full Adder



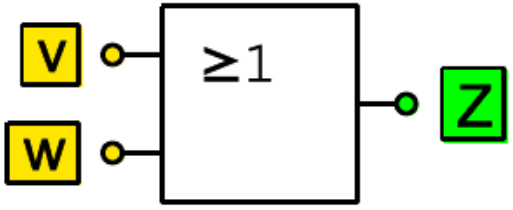




XOR

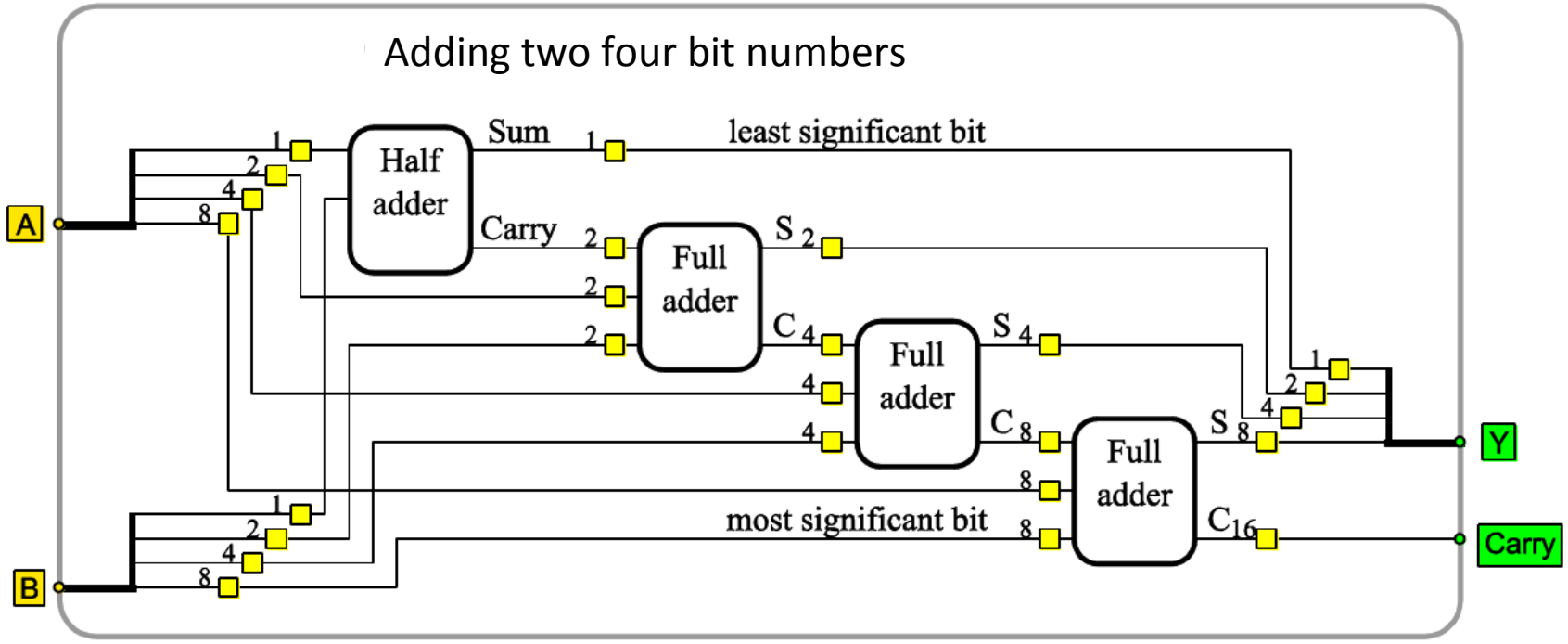


AND



OR

Adder

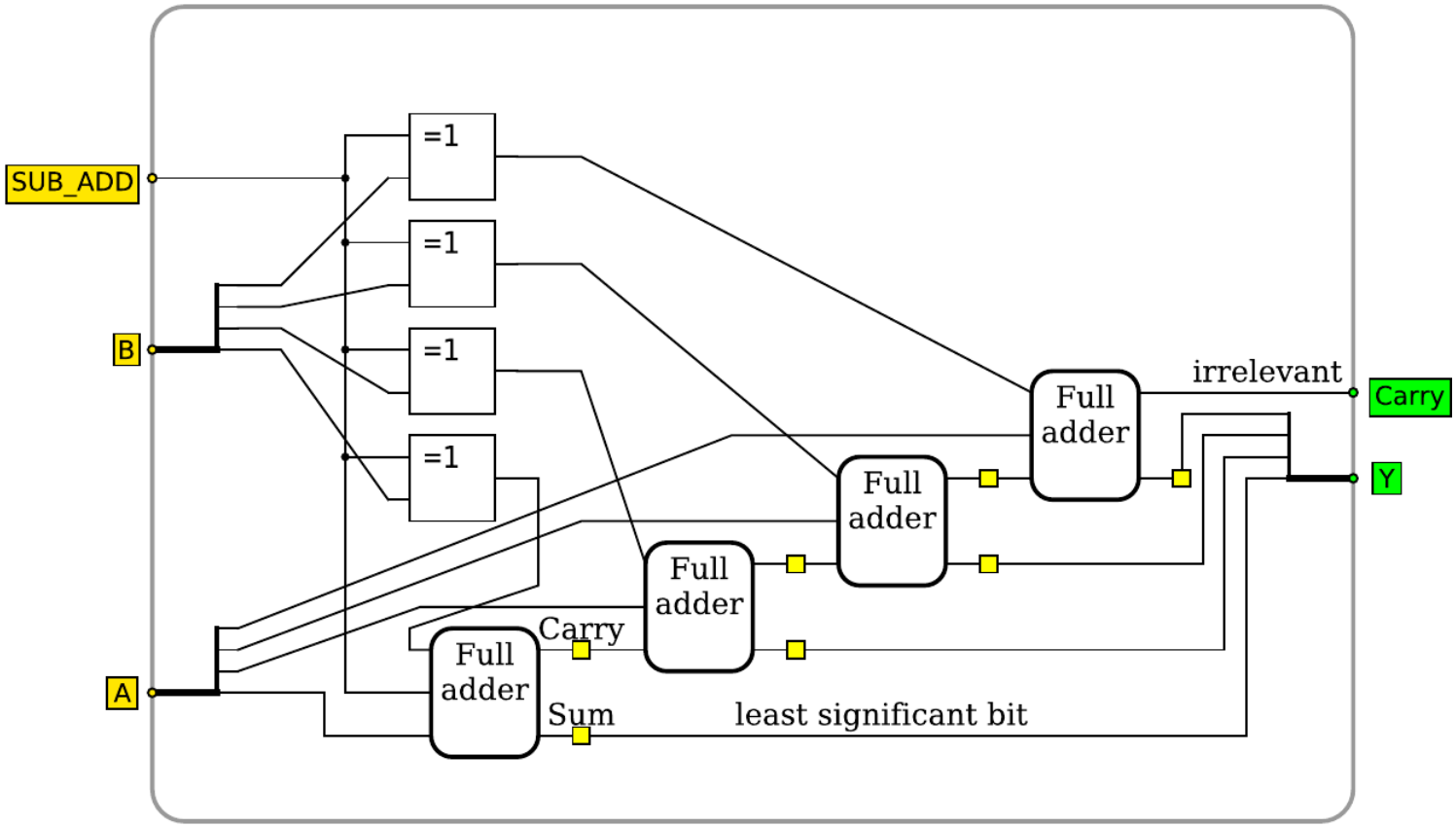

Adding two four bit numbers
A
B
Half adder
Sum
Carry
Full adder
Full adder
Full adder
least significant bit
most significant bit
S 2
C 4
S 4
C 8
S 8
C16
Y
Carry

How to subtrat numbers?

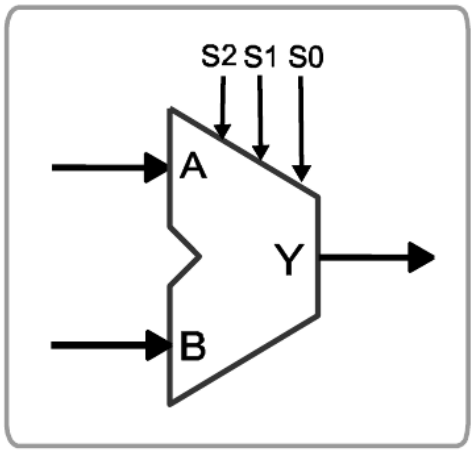
$$a - b = a + \overline{b} + 1$$

How to subtrat numbers?

$$a - b = a + \overline{b} + 1$$

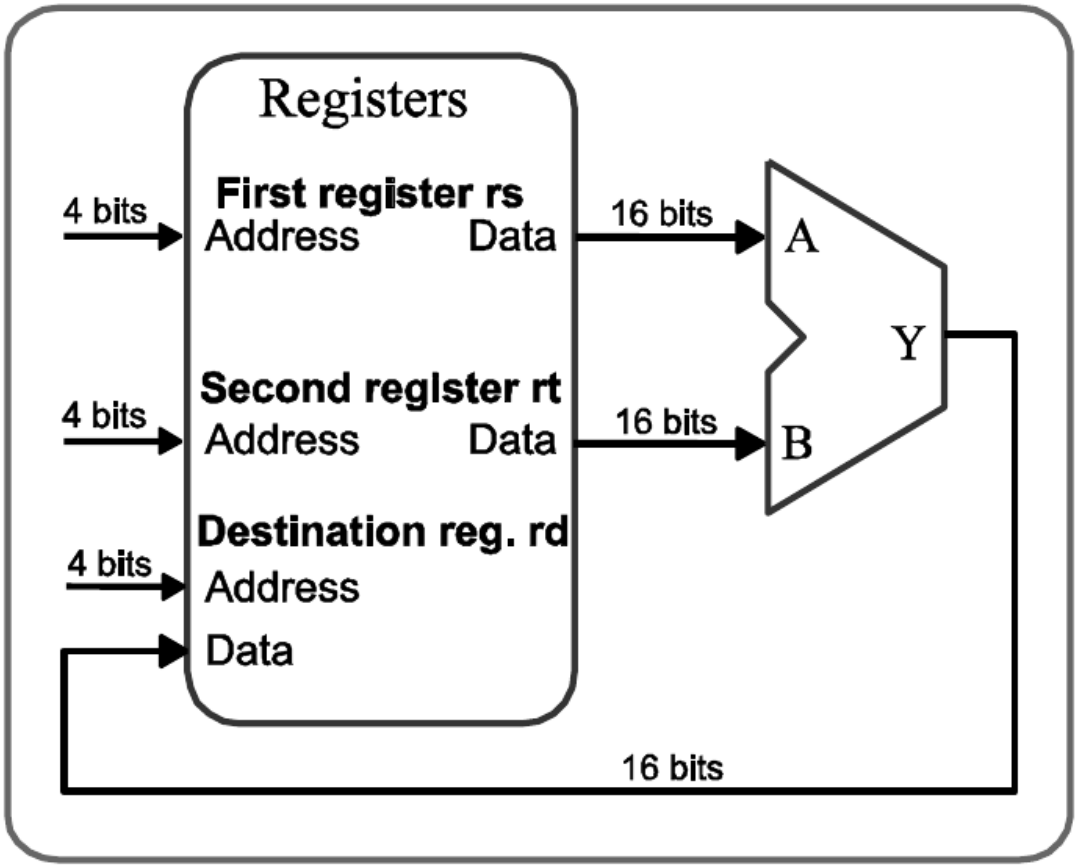


Arithmetic Logic Unit (ALU)



S2	S1	S0	Operator	Function
0	0	0	+ (plus)	Y = A + B
0	0	1	- (min)	Y = A - B
0	1	0	& (bitwise AND)	Y = A AND B
0	1	1	| (bitwise OR)	Y = A OR B
1	0	0	^ (bitwise XOR)	Y = A XOR B
1	0	1	<< (Shift left)	Y = A SHL B
1	1	0	>> (Shift right)	Y = A SHR B
1	1	1	B is passed to the output	Y = B

ALU and Register

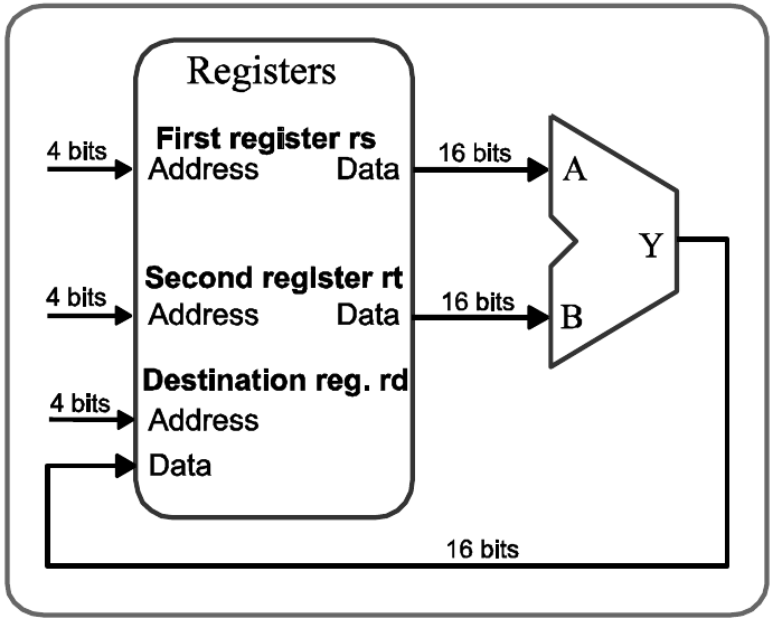


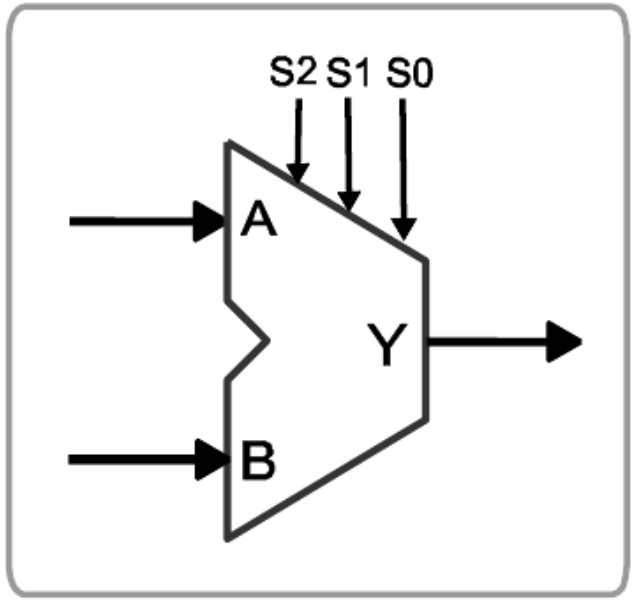
How to control ALU/Registers?
Machine Code

$s_2s_1s_0rs_3rs_2rs_1rs_0rt_3rt_2rt_1rt_0rd_3rd_2rd_1rd_0$

OP regS regT regD

regD = regS [OP] regT



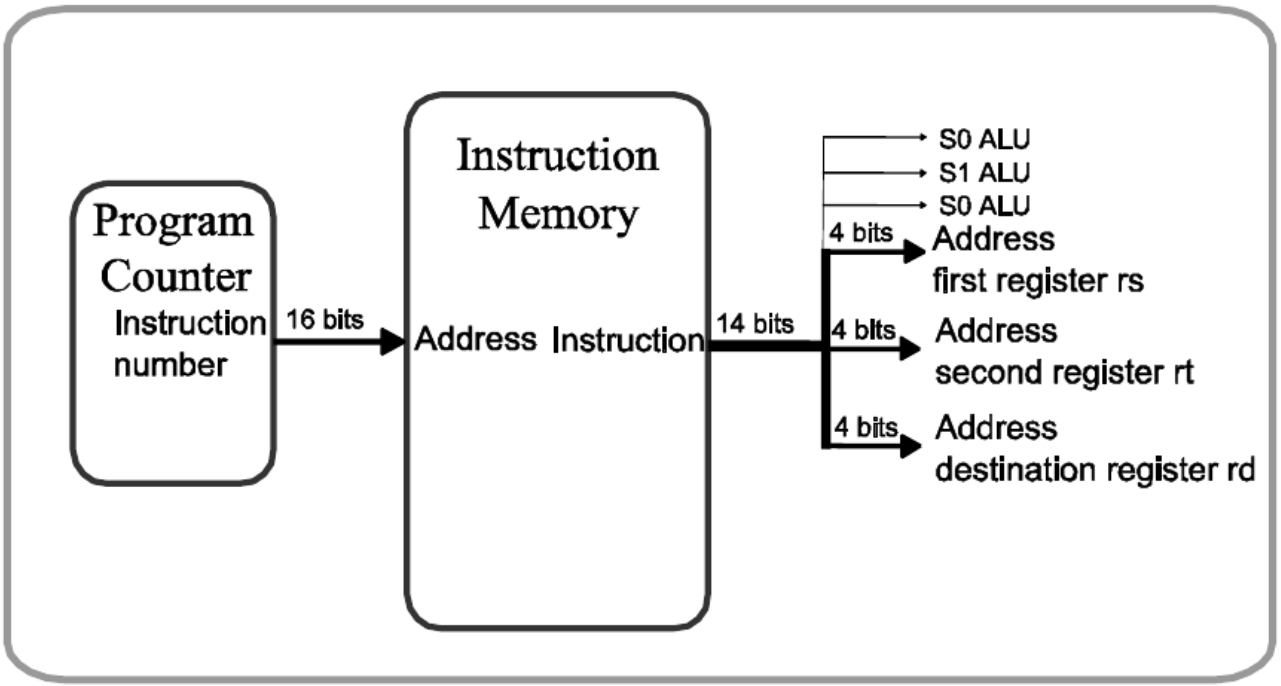


PC + Instruction Memory

$s_2 s_1 s_0 rs_3 rs_2 rs_1 rs_0 rt_3 rt_2 rt_1 rt_0 rd_3 rd_2 rd_1 rd_0$

OP regS regT regD

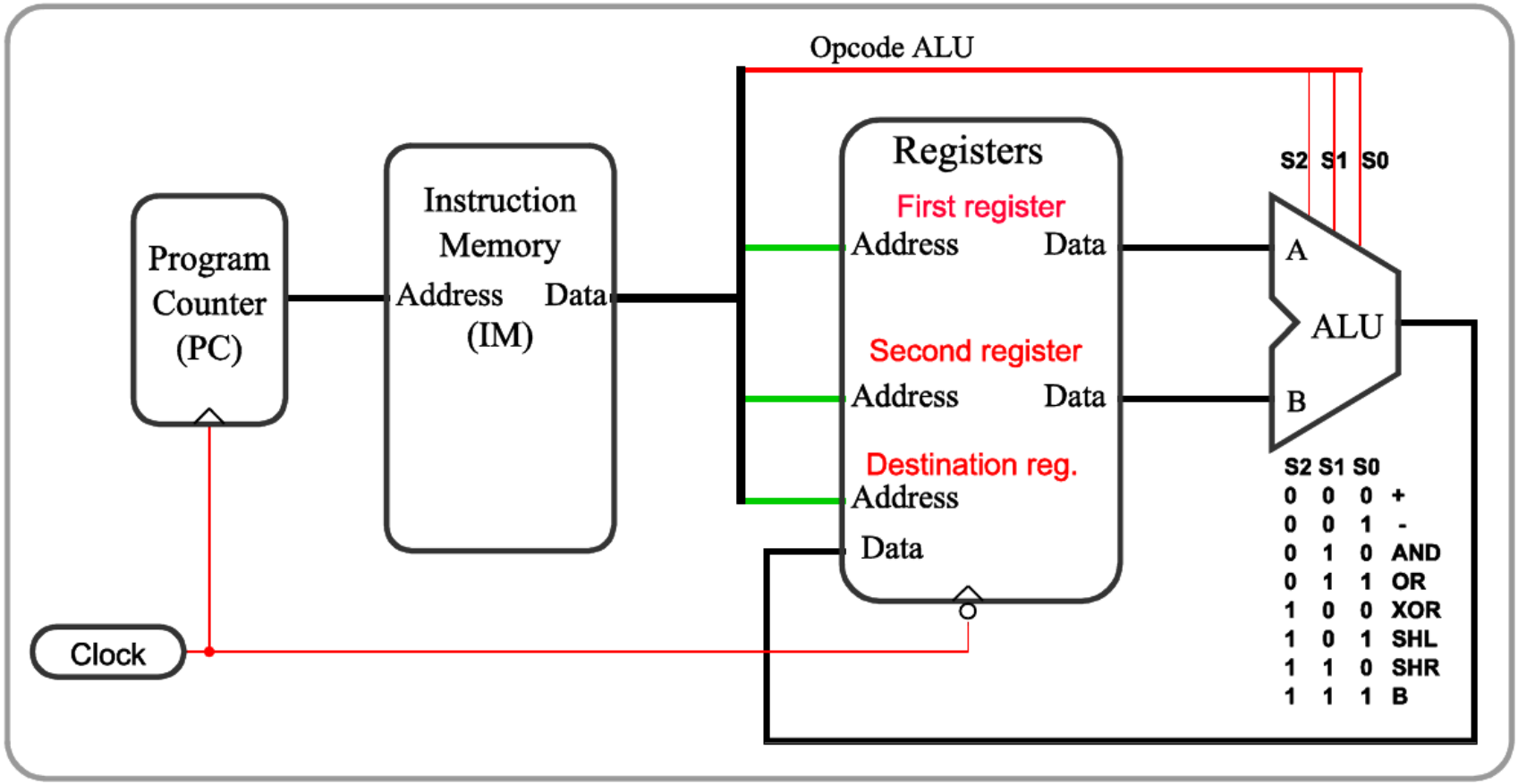
regD = regS [OP] regT



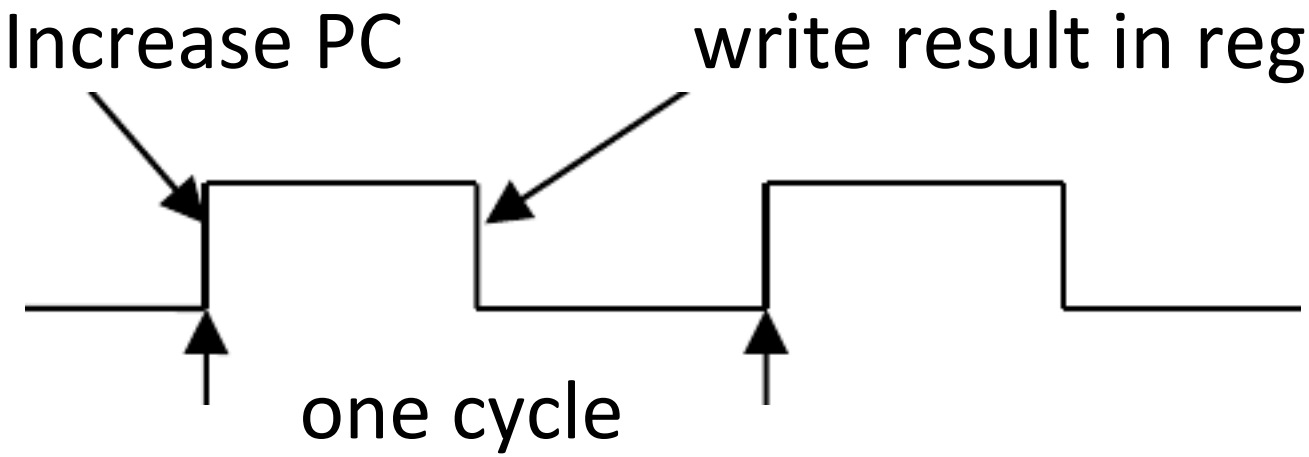
Calculator Instructions

Instruction	Description	Example	
ADD rd, rs, rt	Add registers	ADD $7, $3, $4	r7 ← r3 + r4
SUB rd, rs, rt	subtract registers	SUB $7, $3, $4	r7 ← r3 – r4
AND rd, rs, rt	Bitwise AND registers	AND $7, $3, $4	r7 ← r3 & r4
OR rd, rs, rt	Bitwise OR registers	OR $7, $3, $4	r7 ← r3 | r4
XOR rd, rs, rt	Bitwise XOR registers	XOR $7, $3, $4	r7 ← r3 ^ r4
SHL rd, rs, rt	Shift left register	SHL $7, $3, $4	r7 ← r3 << r4
SHR rd, rs, rt	Shift right register	SHR $7, $3, $4	r7 ← r3 >> r4
COPY rd, rt	Copy register	COPY $7, $4	r7 ← r4

Calculator

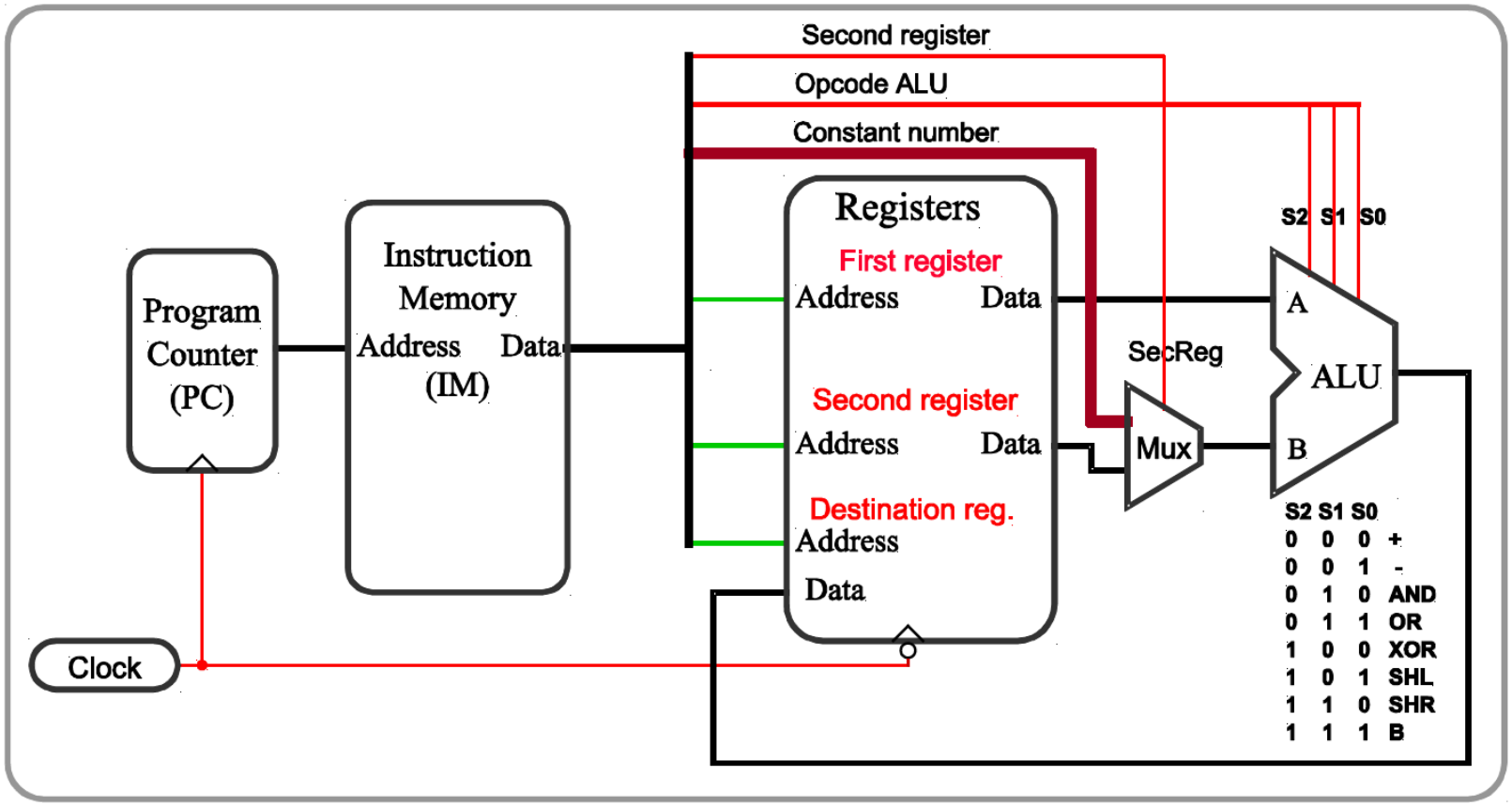

Opcode ALU
Registers
First register
Address
Data
Second register
Address
Data
Destination reg.
Address
Data
Instruction Memory
Address
Data
(IM)
Program Counter (PC)
Clock
S2 S1 S0
A
ALU
B
S2 S1 S0
0 0 0 +
0 0 1 -
0 1 0 AND
0 1 1 OR
1 0 0 XOR
1 0 1 SHL
1 1 0 SHR
1 1 1 B

Summary measures



Calculator with Immediate

Calculator II



Instructions

type= I

Type= 0

Instructie	Description	example	Description
ADD rd, rs, rt	Add registers	ADD $5, $6, $7	r5 ← r6 + r7
SUB rd, rs, rt	subtract registers	SUB $5, $6, $7	r5 ← r6 - r7
AND rd, rs, rt	Bitwise AND registers	AND $5, $6, $7	r5 ← r6 & r7
OR rd, rs, rt	Bitwise OR registers	OR $5, $6, $7	r5 ← r6 | r7
XOR rd, rs, rt	Bitwise XOR registers	XOR $5, $6, $7	r5 ← r6 ^ r7
SHL rd, rs, rt	Shift Left register	SHL $5, $6, $7	r5 ← r6 << r7
SHR rd, rs, rt	Shift Right register	SHR $5, $6, $7	r5 ← r6 >> r7
COPY rd, rt	Copy register	COPY $3, $2	r3 ← r2
ADDI rd, rs, imm	Add register and constant	ADDI $5, $6, 0x1234	r5 ← r6 + 0x1234
SUBI rd, rs, imm	subtract register and constant	SUBI $7, $6, 0x1234	r7 ← r6 - 0x1234
ANDI rd, rs, imm	Bitwise AND register and constant	ANDI $5, $6, 0d34	r5 ← r6 & 0d34
ORI rd, rs, imm	Bitwise OR register and constant	ORI $5, $6, 0d34	r5 ← r6 | 0d34
XORI rd, rs, imm	Bitwise XOR register en constant	XORI $5, $6, 0d34	r5 ← r6 ^ 0d34
SHLI rd, rs, imm	Shift Left register	SHLI $5, $6, 0d5	r5 ← r6 << 5
SHRI rd, rs, imm	Shift Right register	SHRI $5, $6, 0d5	r5 ← r6 >> 5
LOADI rd, imm	Load a num in register	LOAD $1, 0x 0020	r1 ← 0x0020

Machine Code

Instructions with 2 registers

$t_0 s_2 s_1 s_0 rs_3 rs_2 rs_1 rs_0 rt_3 rt_2 rt_1 rt_0 rd_3 rd_2 rd_1 rd_0$

Type OP regS regT regD

regD = regS [OP] regT if type= 1

Type with immediates

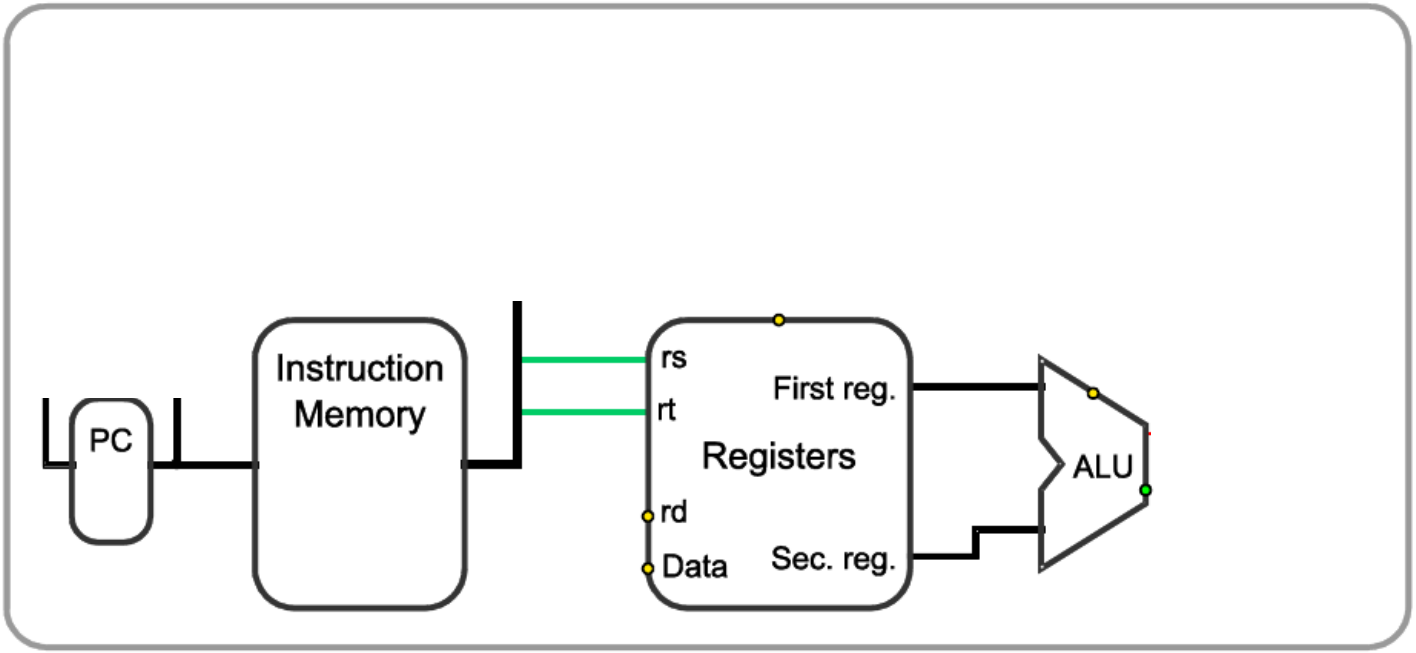
$t_0 s_2 s_1 s_0 rs_3 rs_2 rs_1 rs_0$ xxxx $rt_3 rt_2 rt_1 rt_0 rd_3 rd_2 rd_1 rd_0$

Type OP regS regT regD

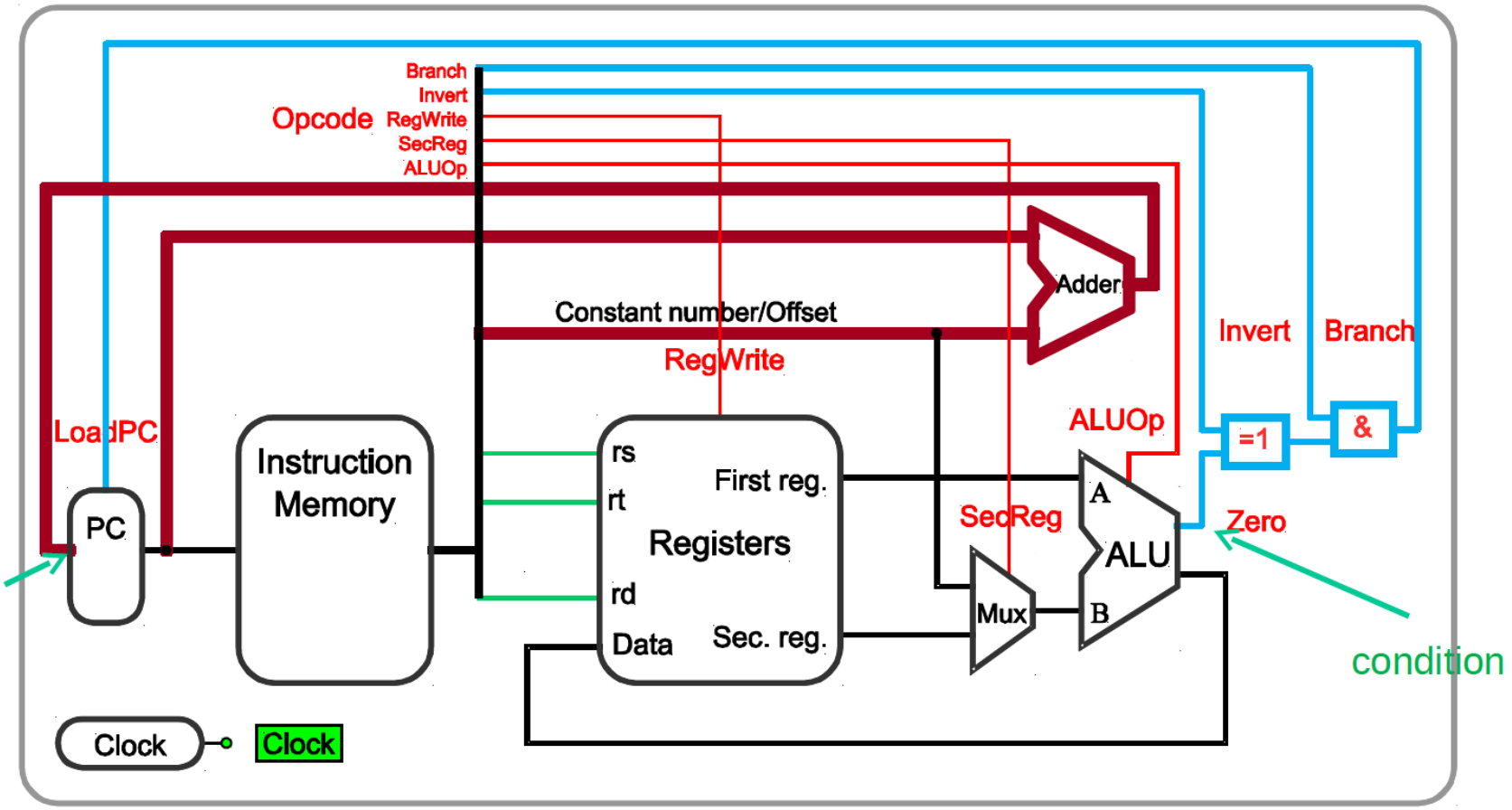
regD = regS [OP] IMM if type= 0

Loop Instruction

BZ rt, label	Branch if rt equal to 0	BZ $6, end	If (r6 == 0) goto 'end'
BNZ rt, label	Branch if rt not equal to 0	BNZ $6, end	If (r6 != 0) goto 'end'
BEQ rs, rt, label	Branch if rs equal to rt	BEQ $6, $8, loop	If (r6 == r8) goto 'loop'
BNQ rs, rt, label	Branch if rs not equal to rt	BNQ $6, $8, loop	If (r6 != r8) goto 'loop'
BRA label	Branch always	BRA label	PC ← PC + offset

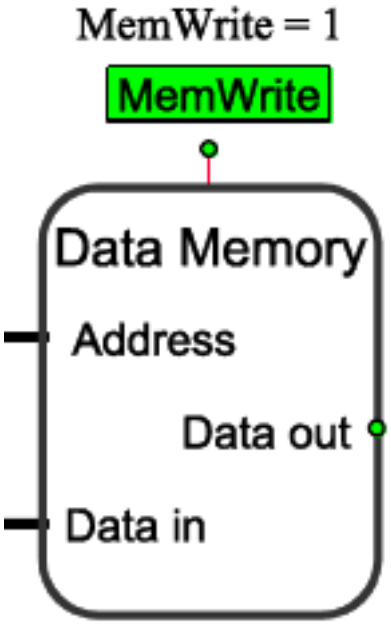


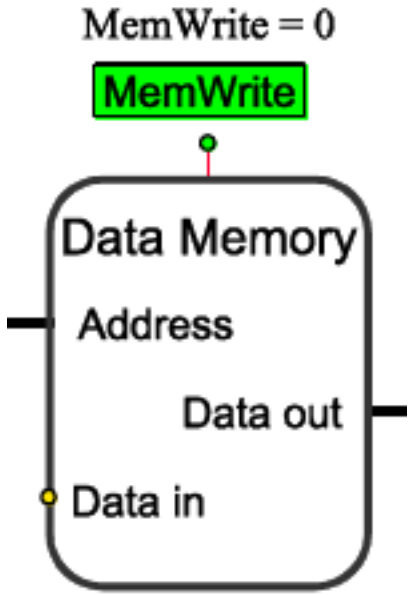
Calculator with Loops



Registers are very limited ...

SW rt, index, rs	Store Word to memory	SW $0, 0x1234, $1	r0 → Mem(r1 +1234_{Hex})
LW rd, index, rs	Load Word to register	LW $0, 0x1234, $1	r0 ← Mem(r1 +1234_{Hex})



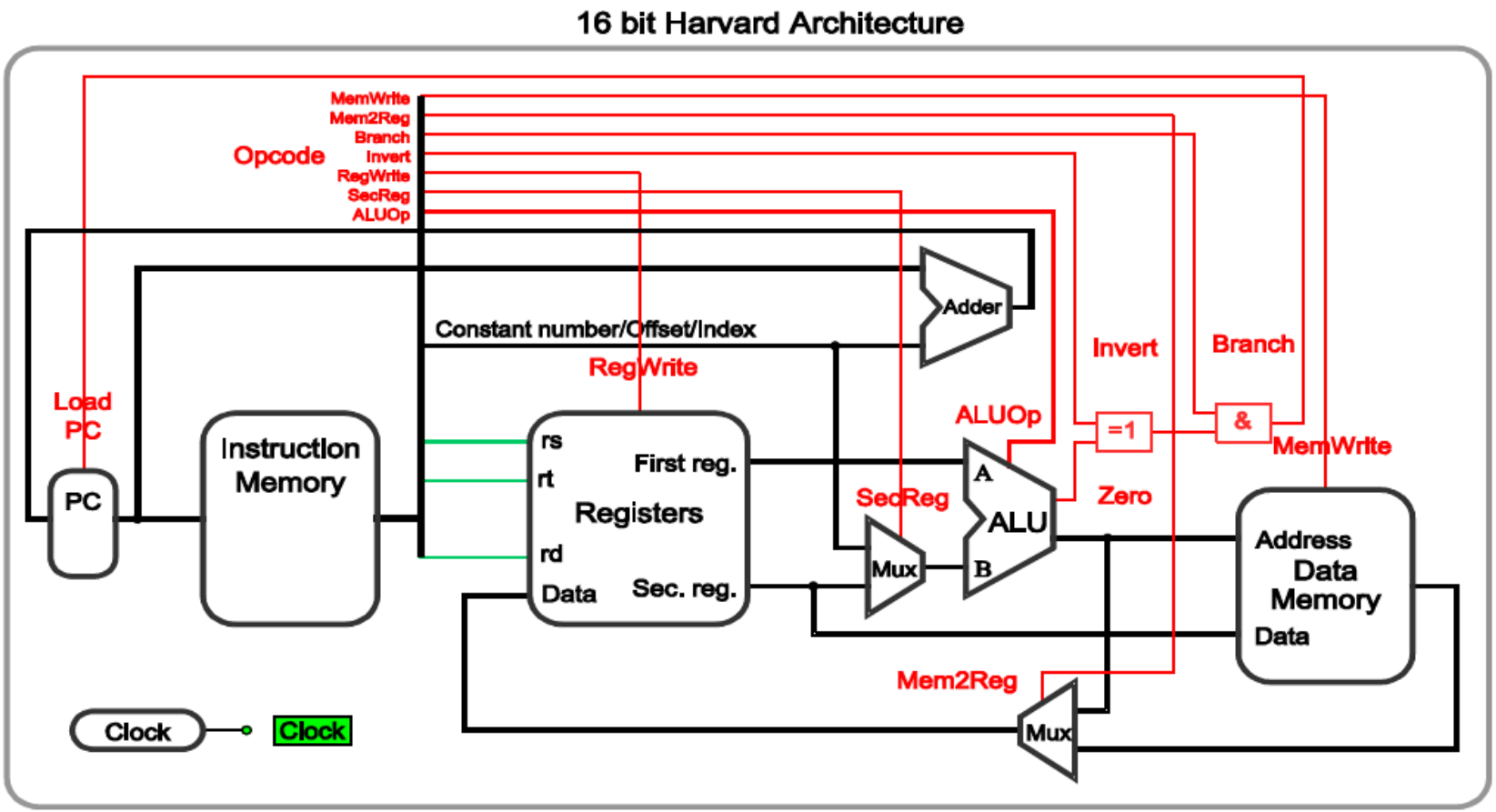


Little Endian/Big Endian

describes order in which bytes are written to memory ...

Number in the register	0A	0B	0C	0D
Memory address	a+3	a+2	a+1	a
Little-endian	0A	0B	0C	0D
Big-endian	0D	0C	0B	0A

Harvard Machine

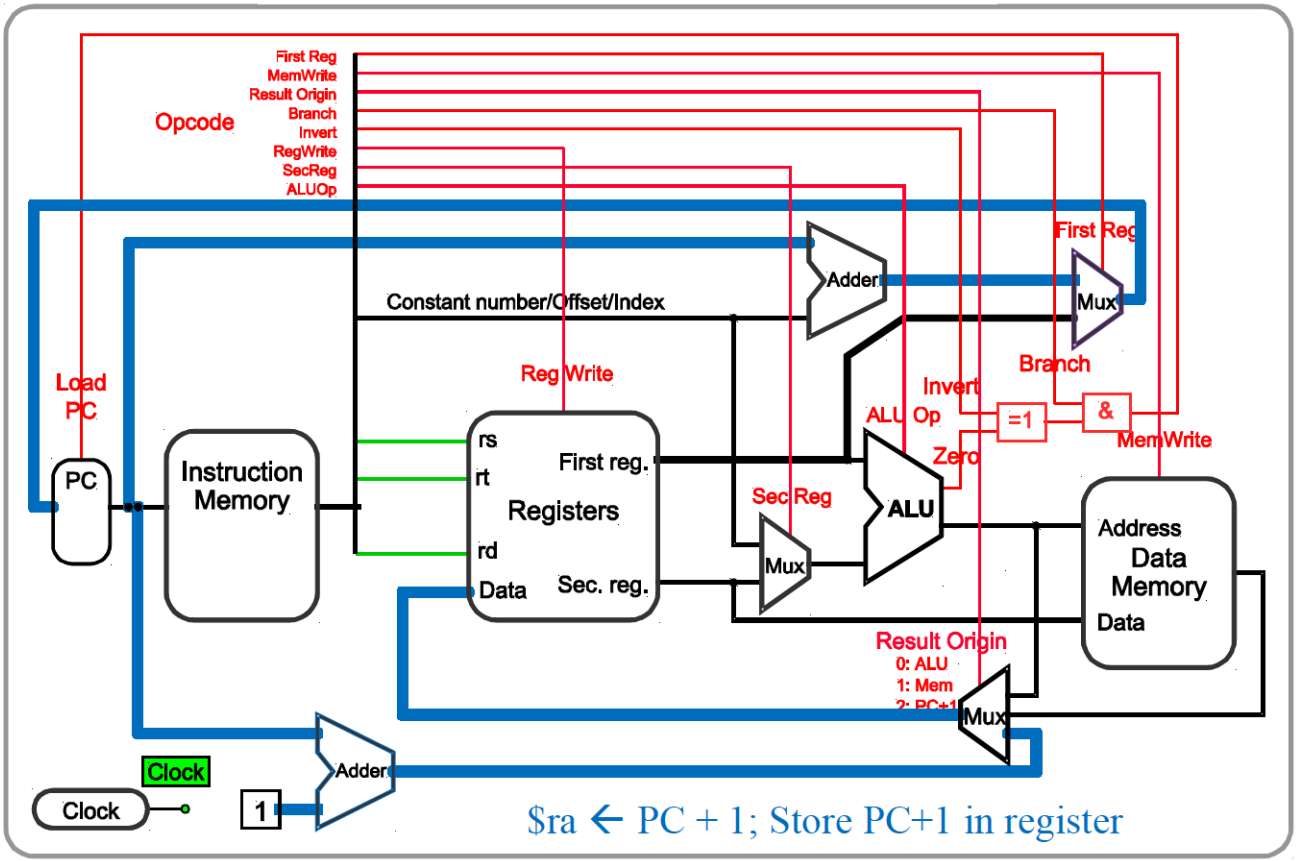


Complete Instruction Set

Mnemonic	Betekenis	Voorbeeld	Betekenis
ADD rd, rs, rt	Optellen registers	ADD $5, $6, $7	r5 ← r6 + r7
SUB rd, rs, rt	Aftrekken registers	SUB $5, $6, $7	r5 ← r6 – r7
AND rd, rs, rt	Bitwise AND registers	AND $5, $6, $7	r5 ← r6 & r7
OR rd, rs, rt	Bitwise OR registers	OR $5, $6, $7	r5 ← r6 | r7
XOR rd, rs, rt	Bitwise XOR registers	AND $5, $6, $7	r5 ← r6 ^ r7
SHL rd, rs, rt	Shift Left register	SHL $5, $6, $7	r5 ← r6 << r7
SHR rd, rs, rt	Shift Right register	SHR $5, $6, $7	r5 ← r6 >> r7
COPY rd, rt	Copy register	COPY $3, $2	r3 ← r2
ADDI rd, rs, imm	Optellen register en const.	ADDI $5, $6, 0x1234	r5 ← r6 + 0x1234
SUBI rd, rs, imm	Aftrekken register en const.	SUBI $7, $6, 0x1234	r7 ← r6 - 0x1234
ANDI rd, rs, imm	Bitwise AND register en const	ANDI $5, $6, 0d34	r5 ← r6 & 0d34
ORI rd, rs, imm	Bitwise OR register en const.	ORI $5, $6, 0d34	r5 ← r6 | 0d34
XORI rd, rs, imm	Bitwise XOR register en const	XORI $5, $6, 0d34	r5 ← r6 ^ 0d34
SHLI rd, rs, imm	Shift Left register	SHLI $5, $6, 5	r5 ← r6 << 5
SHRI rd, rs, imm	Shift Right register	SHRI $5, $6, 5	r5 ← r6 >> 5
LOADI rd, imm	Laad constante in register	LOADI $1, 0x 0020	r1 ← 0x0020
BZ rt, label	Branch if rt gelijk is aan 0	BZ $6, end	If (r6 == 0) goto ‘end’
BNZ rt, label	Branch if rt ongelijk is aan 0	BNZ $6, end	If (r6 != 0) goto ‘end’
BEQ rs, rt, label	Branch if rs gelijk is aan rt	BEQ $6, $8, loop	If (r6 == r8) goto ‘loop’
BNE rs, rt, label	Branch if rs ongelijk is aan rt	BNE $6, $8, loop	If (r6 != r8) goto ‘loop’
BRA label	Branch always	BRA label	PC ← PC + offset
SW rt, index, rs	Store Word to memory	SW $0, 0x1234, $1	r0 → Mem(r1 +1234_{Hex})
LW rd, index, rs	Load Word to register	LW $0, 0x1234, $1	r0 ← Mem(r1 +1234_{Hex})

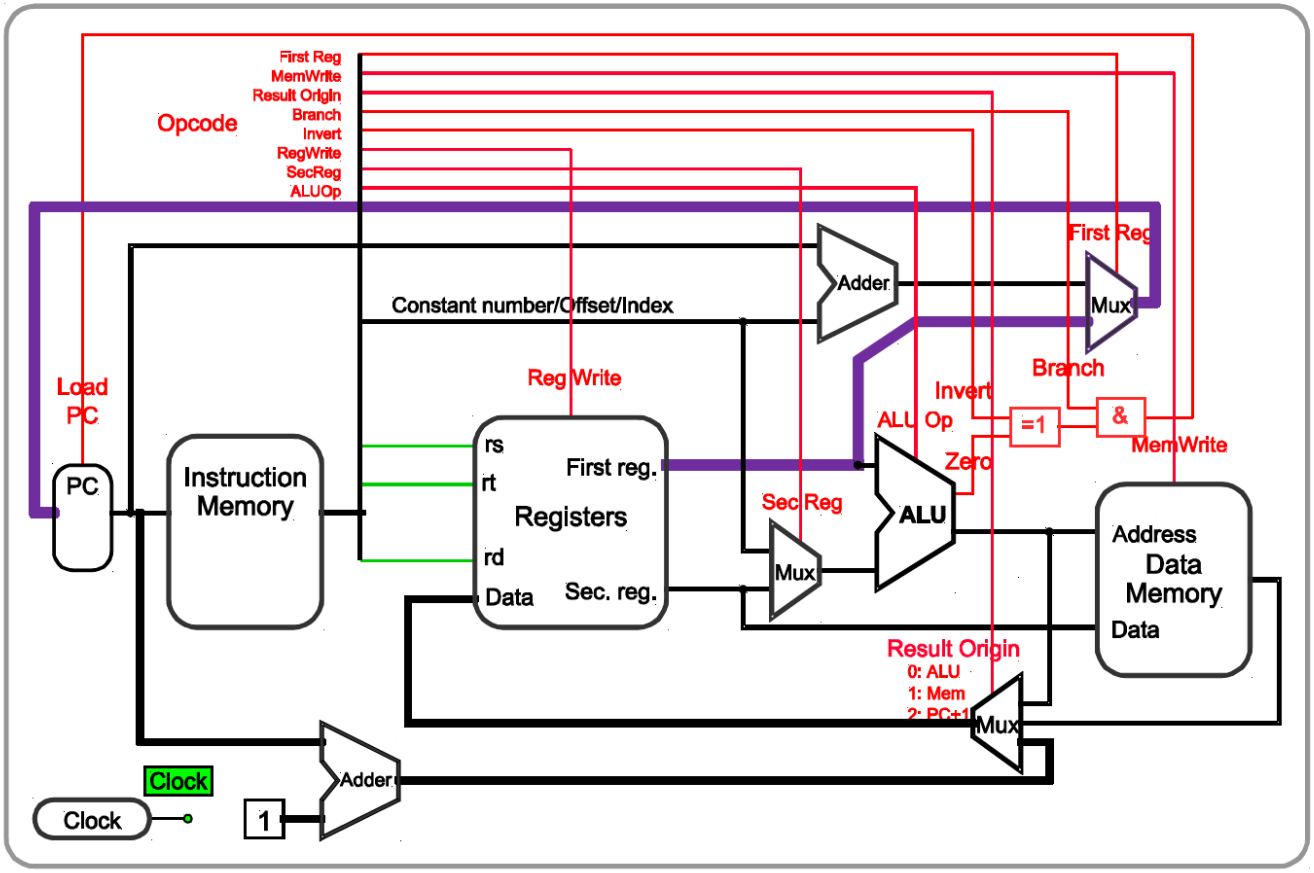
Procedure Call

PC ← PC + Offset



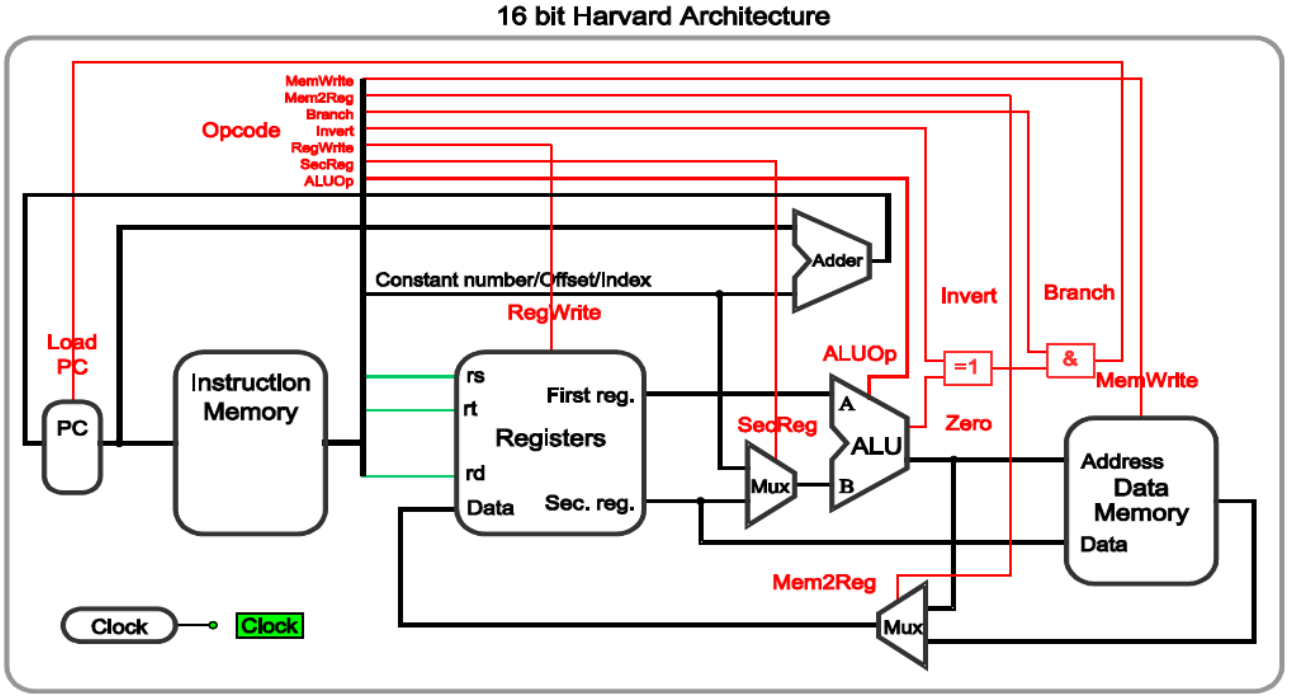
Harvard Machine: Return

Load PC from register $ra



von Neumann Architecture

- Just like Harvard Architecture, with one little difference ➔ Instruction Memory and Data Memory are the same



Turing Complete

- We can now compute whatever we want …
- Further improvements target
  - **increasing usability**
  - **speed.**

CPU Time

- CPU_Time= CPU_Clock_Cycles x Clock Cycle Time
- $$\text{CPU\_Time} = \frac{\text{CPU\_Clock\_Cycles}}{\text{Clock\_Rate}}$$
- Performance improved by
  - Reducing number of clocks cycles
  - Increasing clock rate
  - Hardware designers most often trade off clock rate against number of cycle

Instruction Count and CPU Time

- Instruction count for a program
  - Determined by the program (ISA and complier)
- Average cycles per instruction
  - Determined by the CPU architecture
  - If different instruction have different CPI

Clock_Cycles = Instruction_count * Cycles_per_instruction

CPU_Time = Instuction_count * Cycles_per_instruction

$$\text{CPU\_Time} = \frac{\text{Instruction\_count} * \text{CPI}}{\text{Clock\_Rate}}$$

- Suppose a program of 1000 instructions
  - When running on a pipelined machine it required 1004 cycles
    - CPI=1000/1004 ~1 each cycle is 0.2 ns
    - CPU time= 1004*1*0,2= 0.2 microsecond
  - When the same program is run on the single machine cycle than the
    - CPI 1 (per definition) and the cycle time is 1 ns .The CPU time is : 1000 * 1 * 1 ns = 1 ns

$$\text{CPU time} = \text{Instruction count} * \text{CPI} * \text{Clock cycle time} = \frac{\text{Instruction count} * \text{CPI}}{\text{Clock rate}}$$

Interrupts

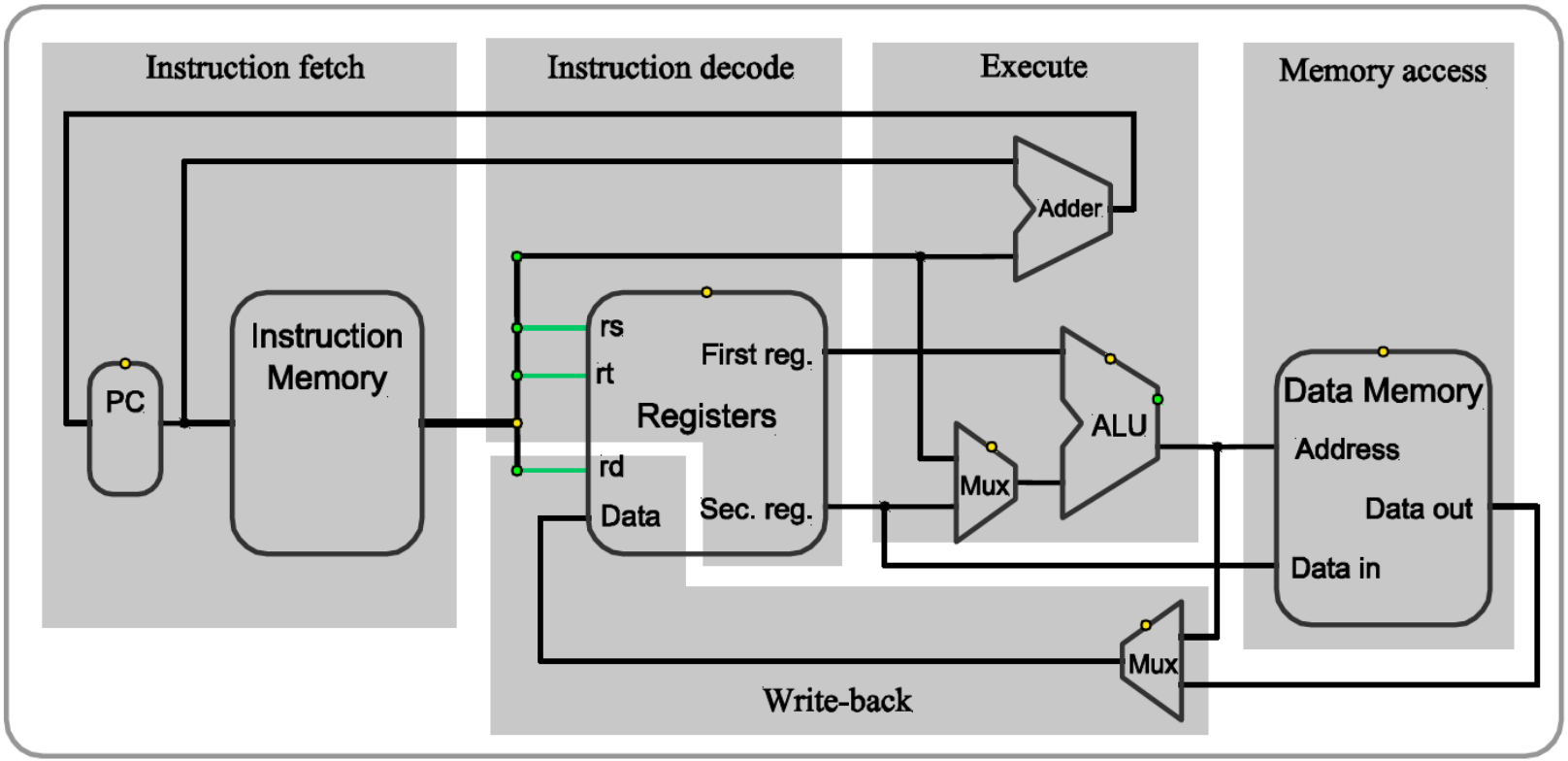
- ways to interrupt current execution
- for other stuff
- Examples:
  - pressing a key on the keyboard
  - network data available
  - shutting down processes
  - moving mouse
- more details in the next lecture

Pipelining

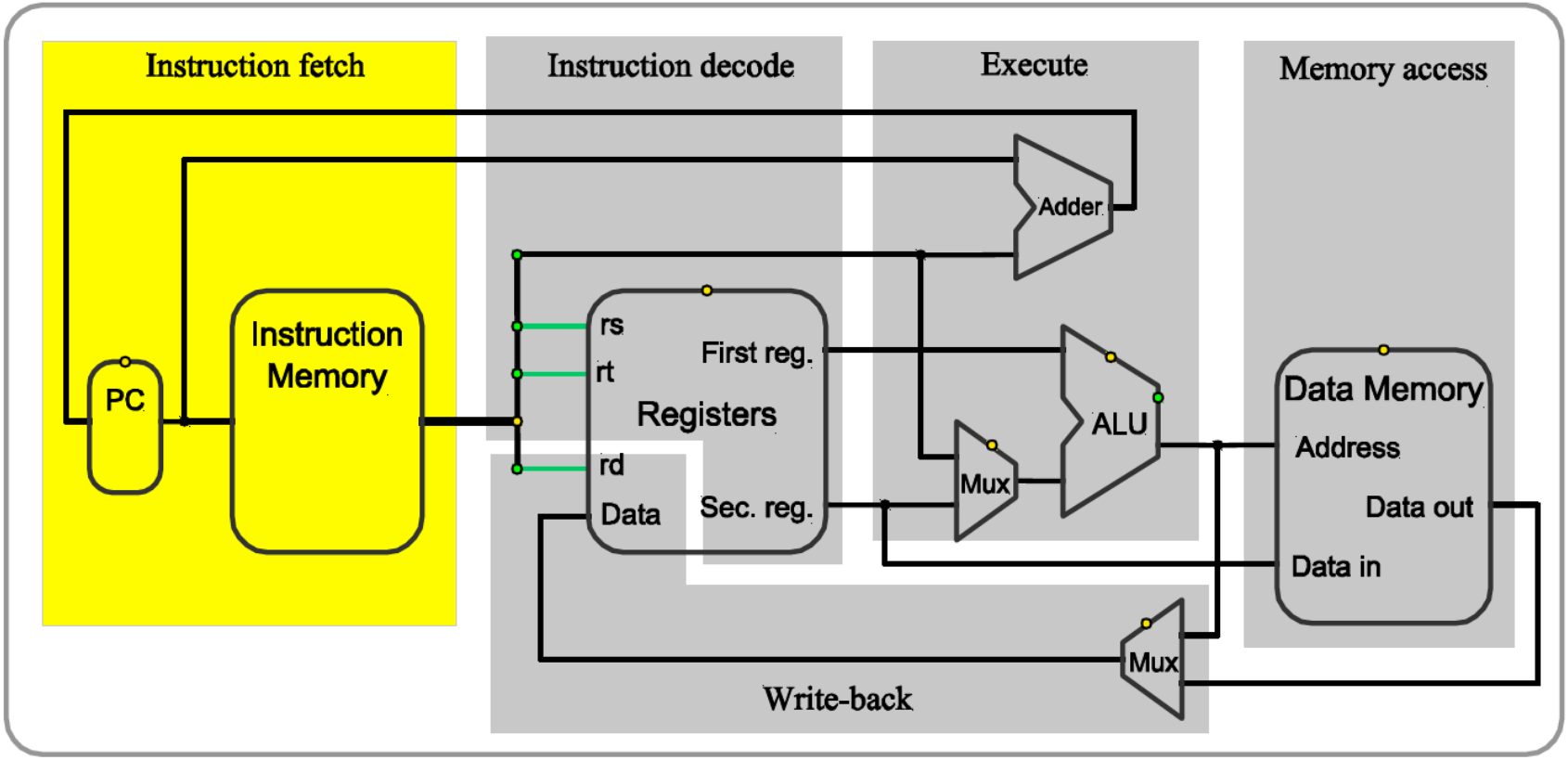
- currently:
  - **multiple cycles for one instruction**
  - large parts of the pipeline are idle
- with pipelining
  - one cycle for one instruction
  - nearly all parts are nearly always busy

- (similar to conveyor belt)

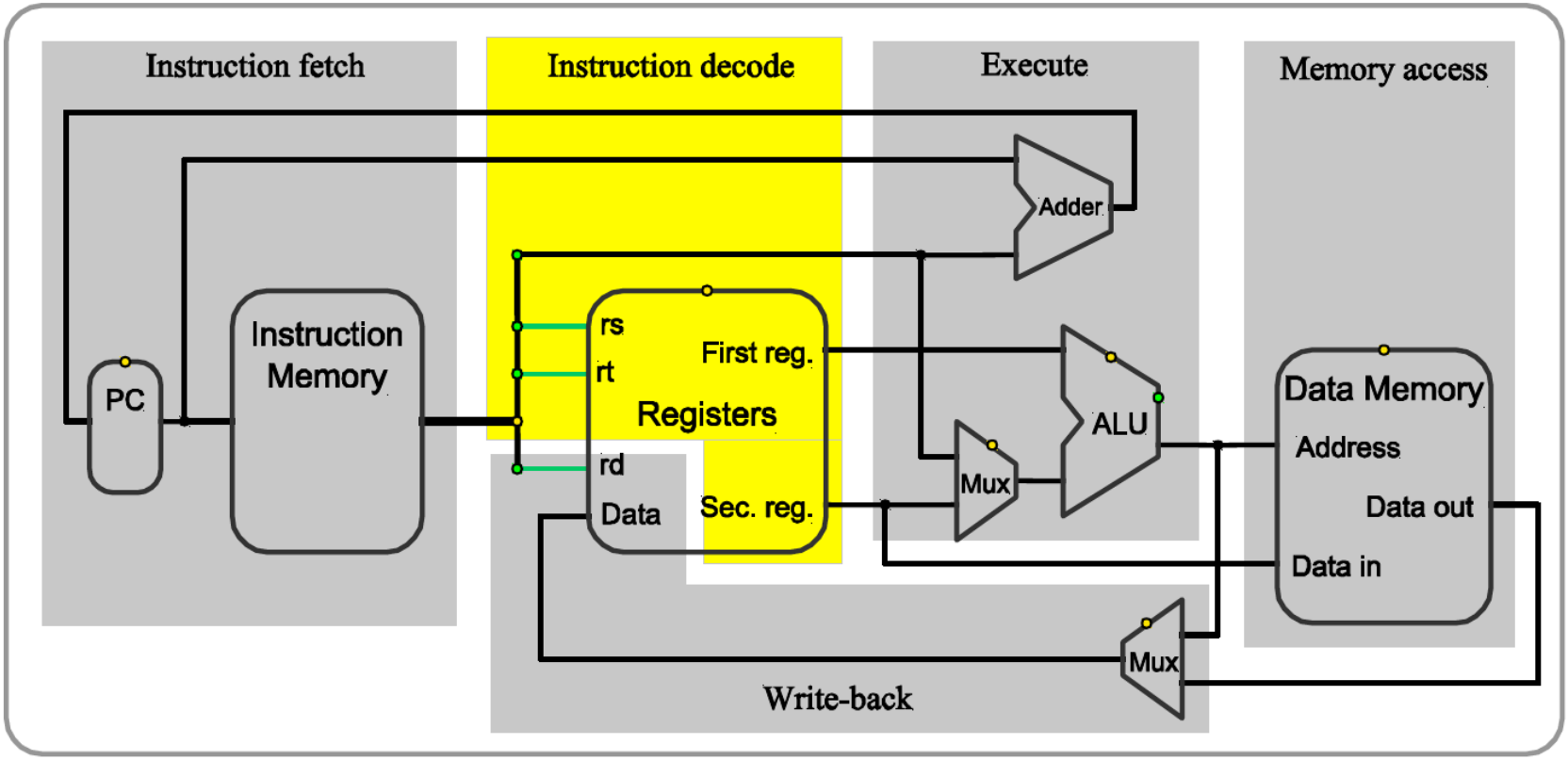
Pipelining


Instruction fetch
Instruction decode
Execute
Memory access
Adder
PC
Instruction Memory
rs
rt
rd
Data
Registers
First reg.
Sec. reg.
Mux
ALU
Data Memory
Address
Data out
Data in
Mux
Write-back

Pipelining

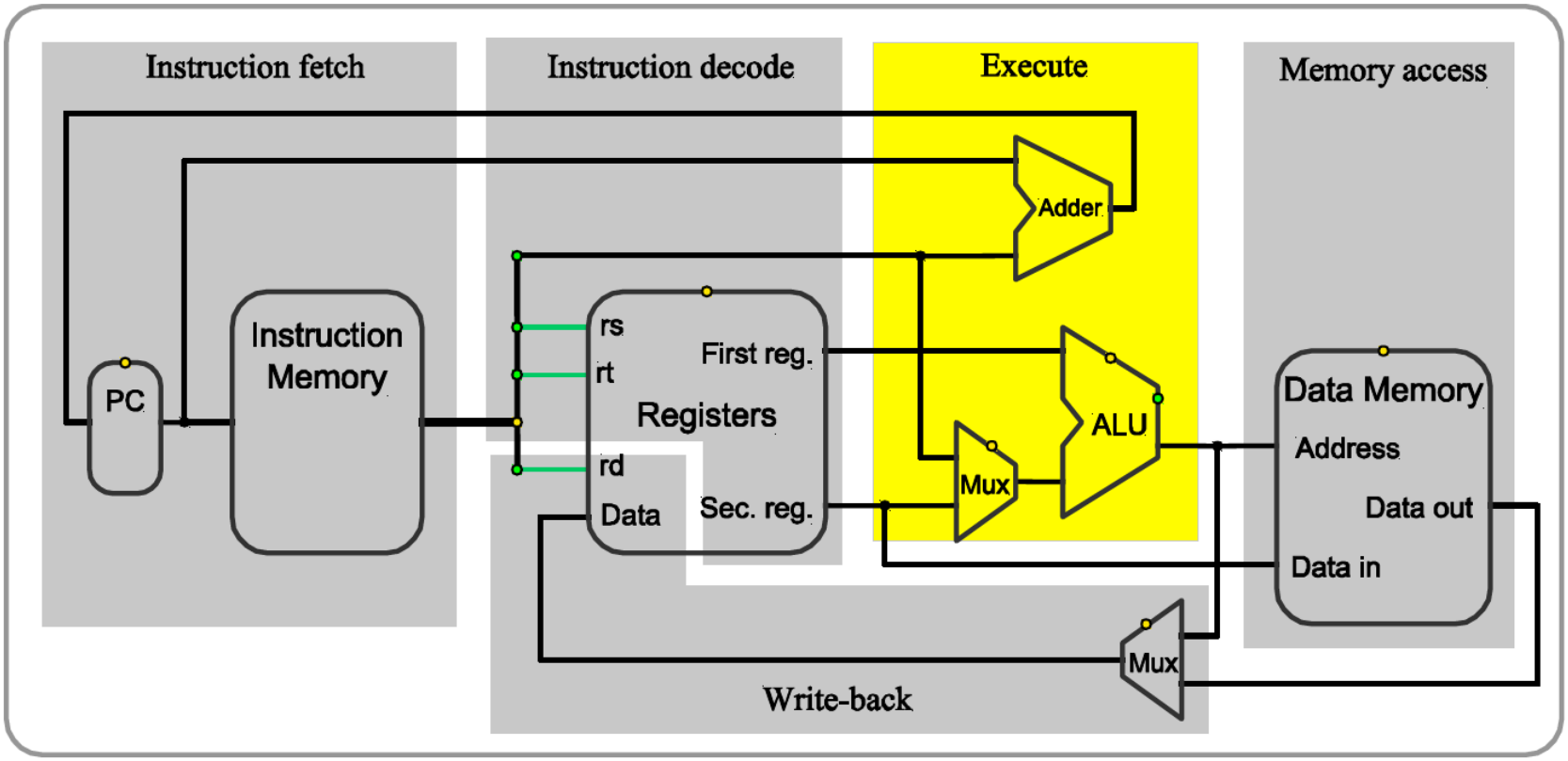

Instruction fetch
Instruction decode
Execute
Memory access
Adder
PC
Instruction Memory
rs
rt
rd
Data
Registers
First reg.
Sec. reg.
Mux
ALU
Data Memory
Address
Data out
Data in
Mux
Write-back

Pipelining



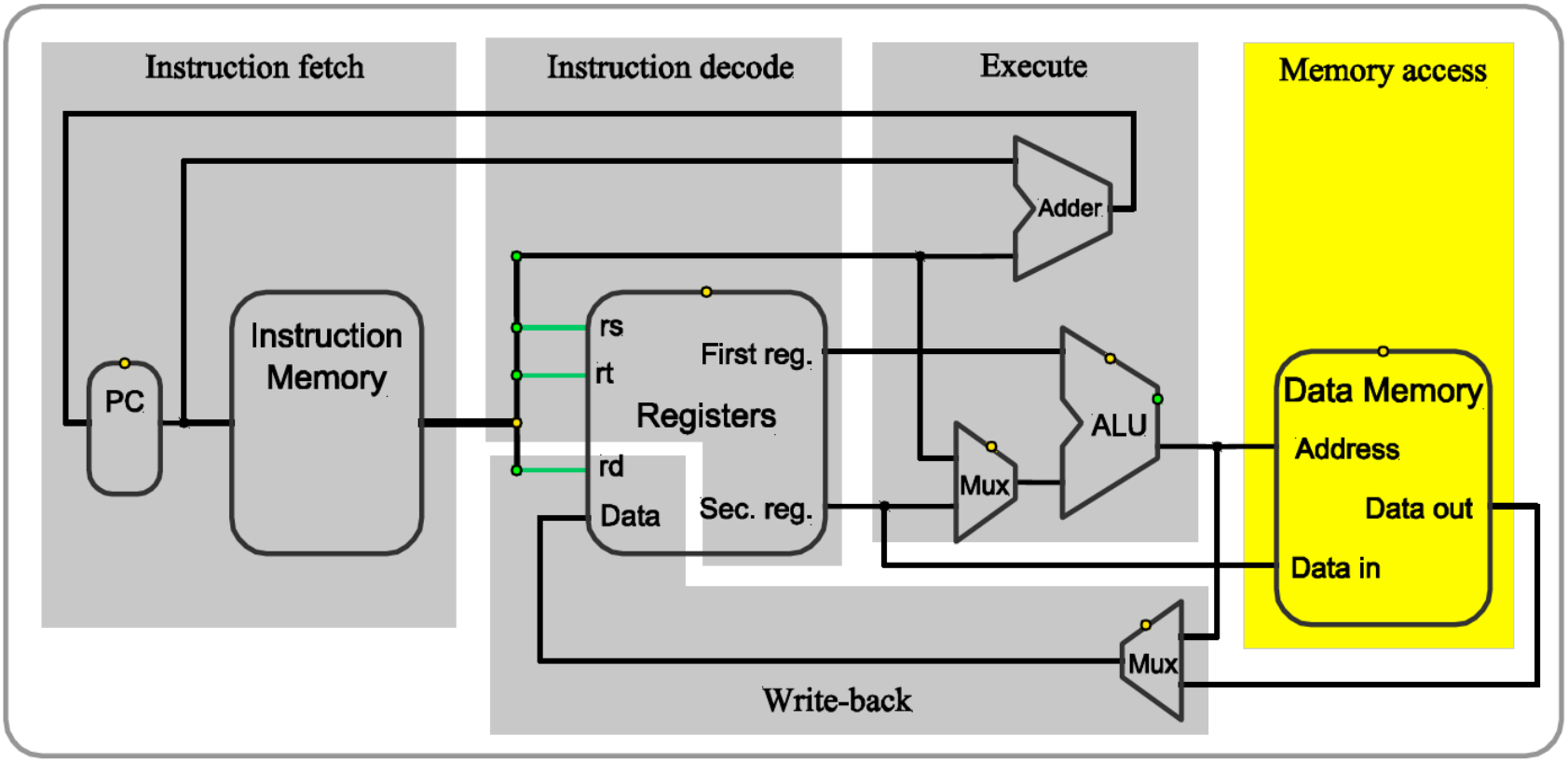
Alison Gibbs and Jeffrey Rosenthal, Statistics Making sense of data, university of Toronto, [coursera]

Pipelining



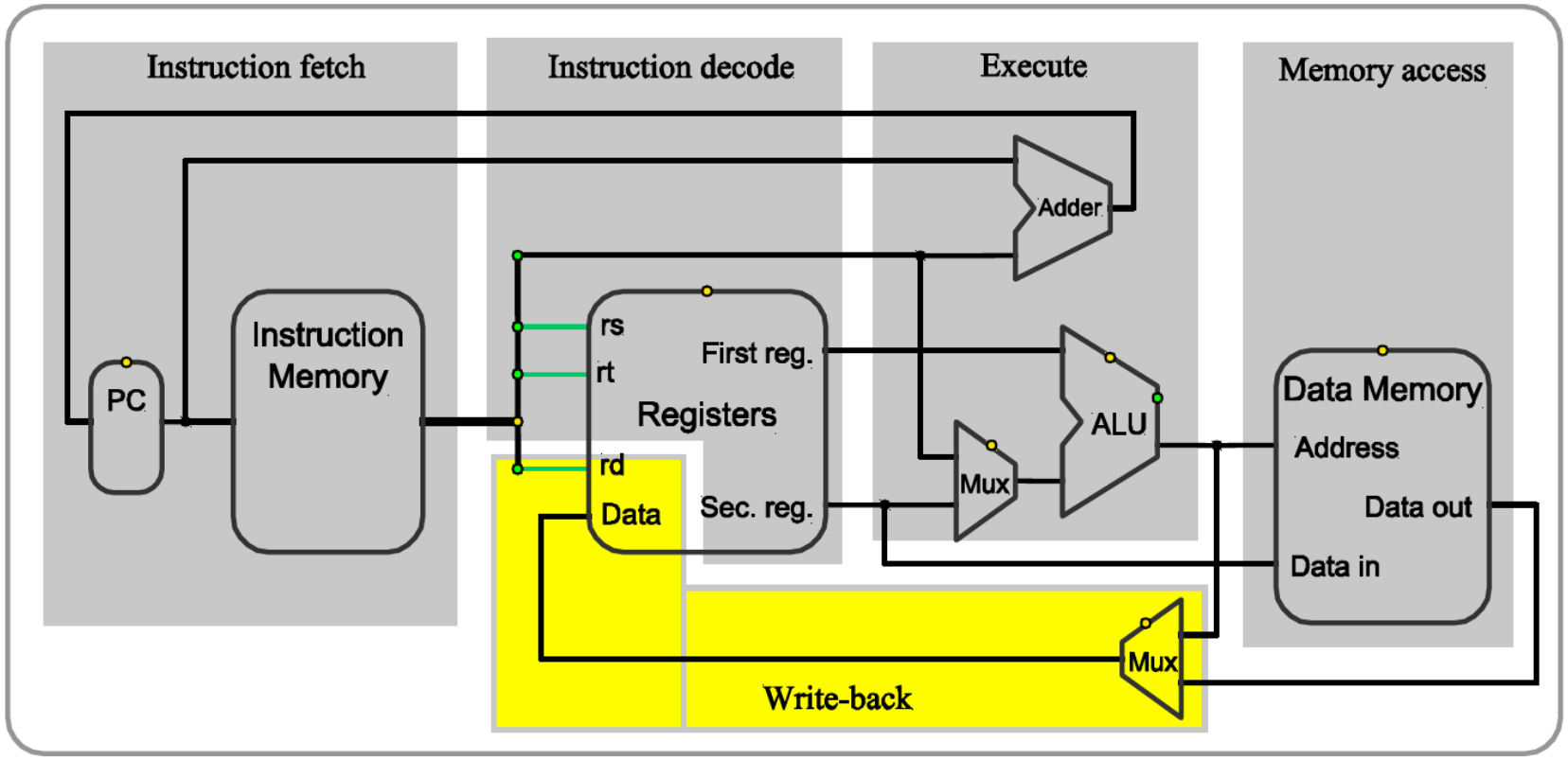
Alison Gibbs and Jeffrey Rroenthal, Statistics Making sense of data,university of Toronto,[coursera]

Pipelining



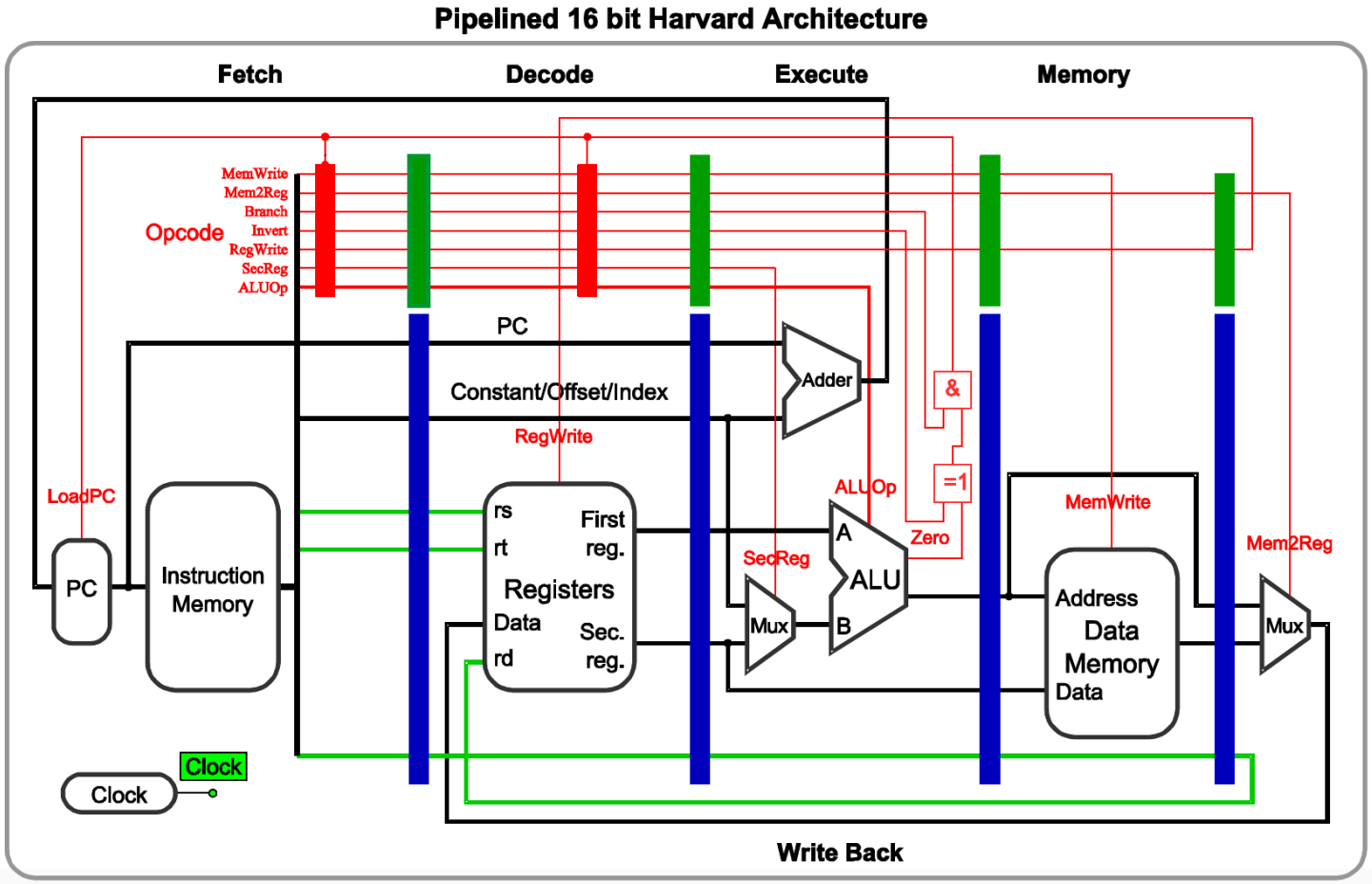
Alison Gibbs and Jeffrey Rroenthal, Statistics Making sense of data, university of Toronto,[coursera]

Pipelining



Alison Gibbs and Jeffrey Rroenthal, Statistics Making sense of data, university of Toronto,[coursera]

Harvard Pipelining

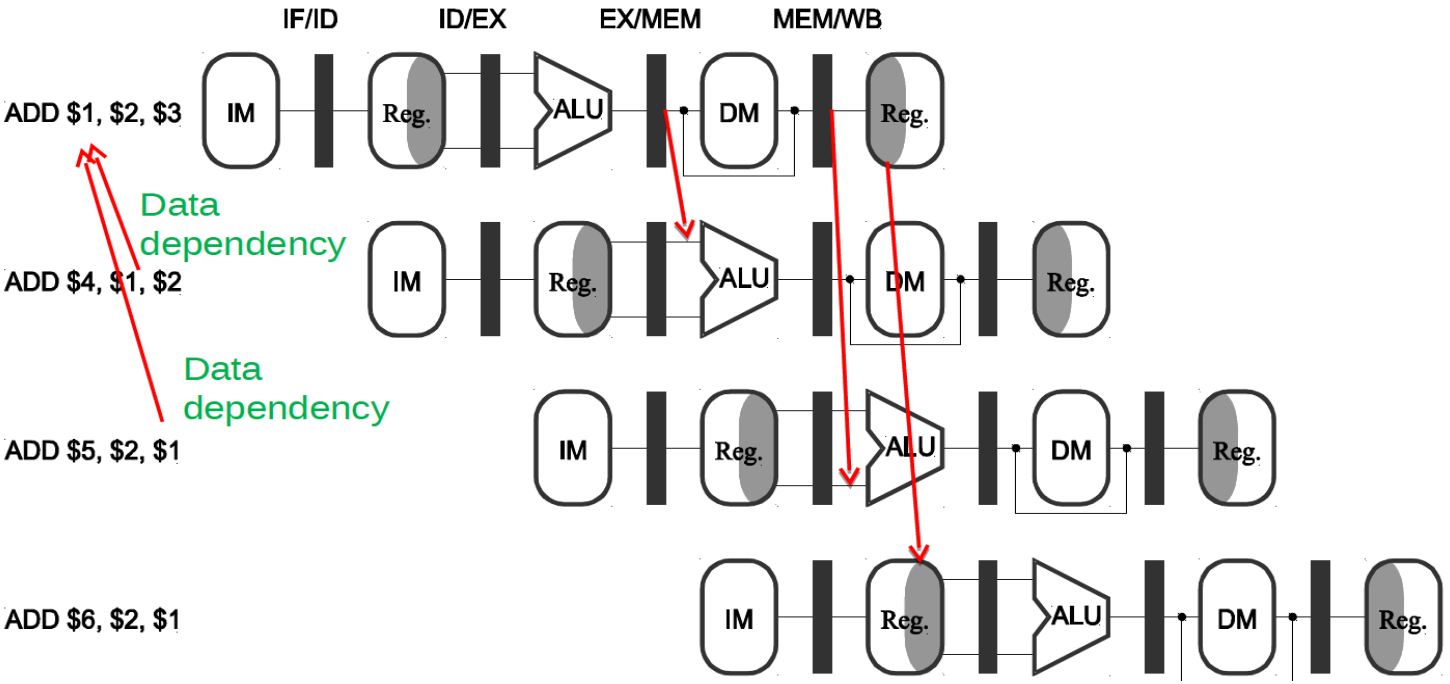


Pipeline Hazards/ Multicycle Instructions

- One instruction per cycle is the best case …
- but that's not always possible:
  - Memory accesses
  - Branches
  - Dependencies

```
ADD $r1, $r2, $r3
ADD $r4, $r1, $r2
```

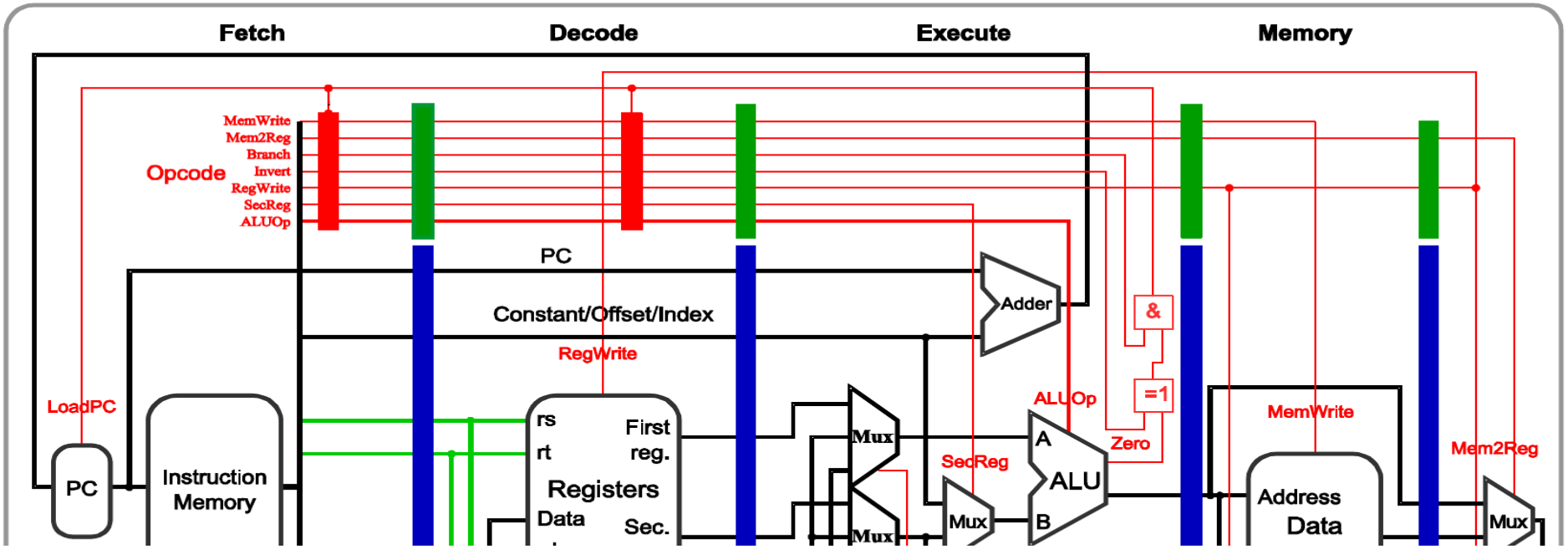
Harvard Pipelining



Instr. nr. \ cycle	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EX	MEM	WB					
2		IF	ID	EX	MEM	WB				
3			IF	ID	EX	MEM	WB			
4				IF	ID	EX	MEM	WB		
5					IF	ID	EX	MEM	WB	
6						IF	ID	EX	MEM	WB

Harvard Machine with Forwarding

Pipelined Harvard Architecture with Forwarding



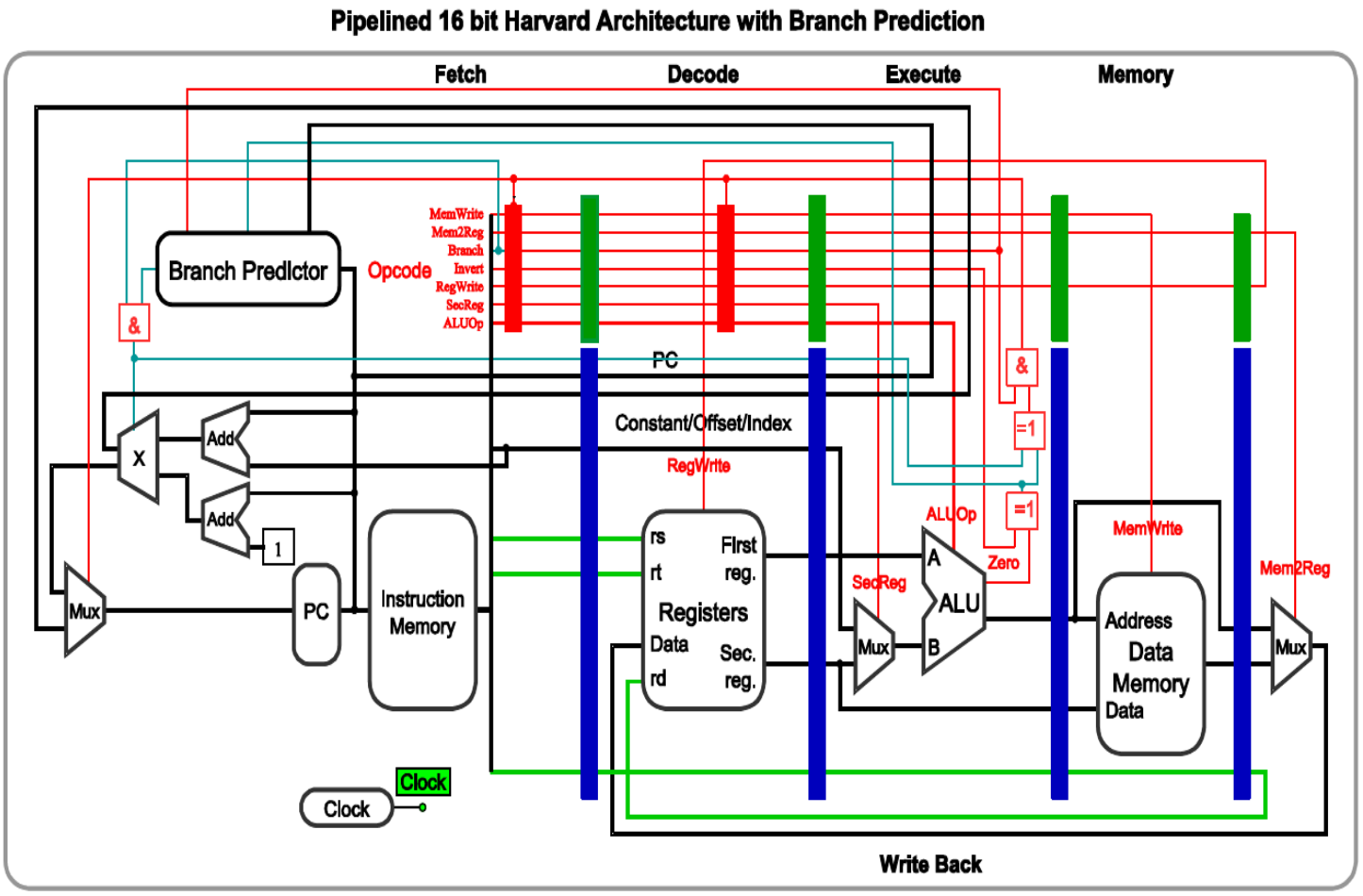
Without operand forwarding

1	2	3	4	5	6	7	8
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result			
	Fetch SUB	Decode SUB	*stall*	*stall*	Read Operands SUB	Execute SUB	Write result

With operand forwarding

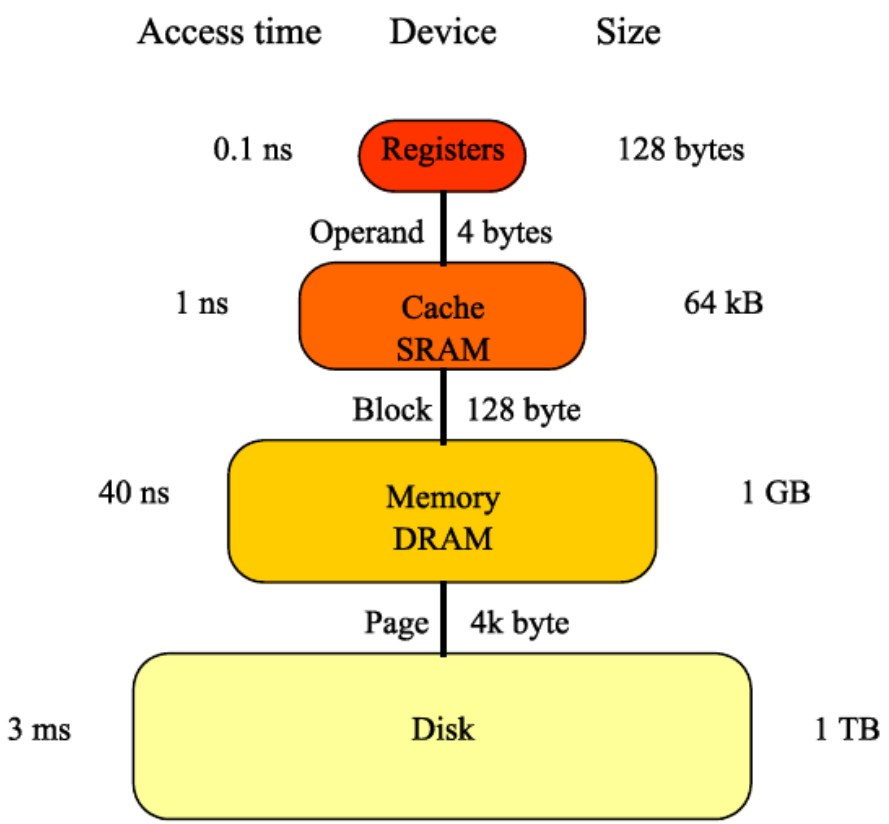
1	2	3	4	5	6
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result	
	Fetch SUB	Decode SUB	Read Operands SUB: use result from previous operation	Execute SUB	Write result

Branch Prediction



Memory Hierarchy

- emulates a fast and large memory
  - on top: small and fast
  - on bottom: large and
- slow
- each level contains a subset of the data below

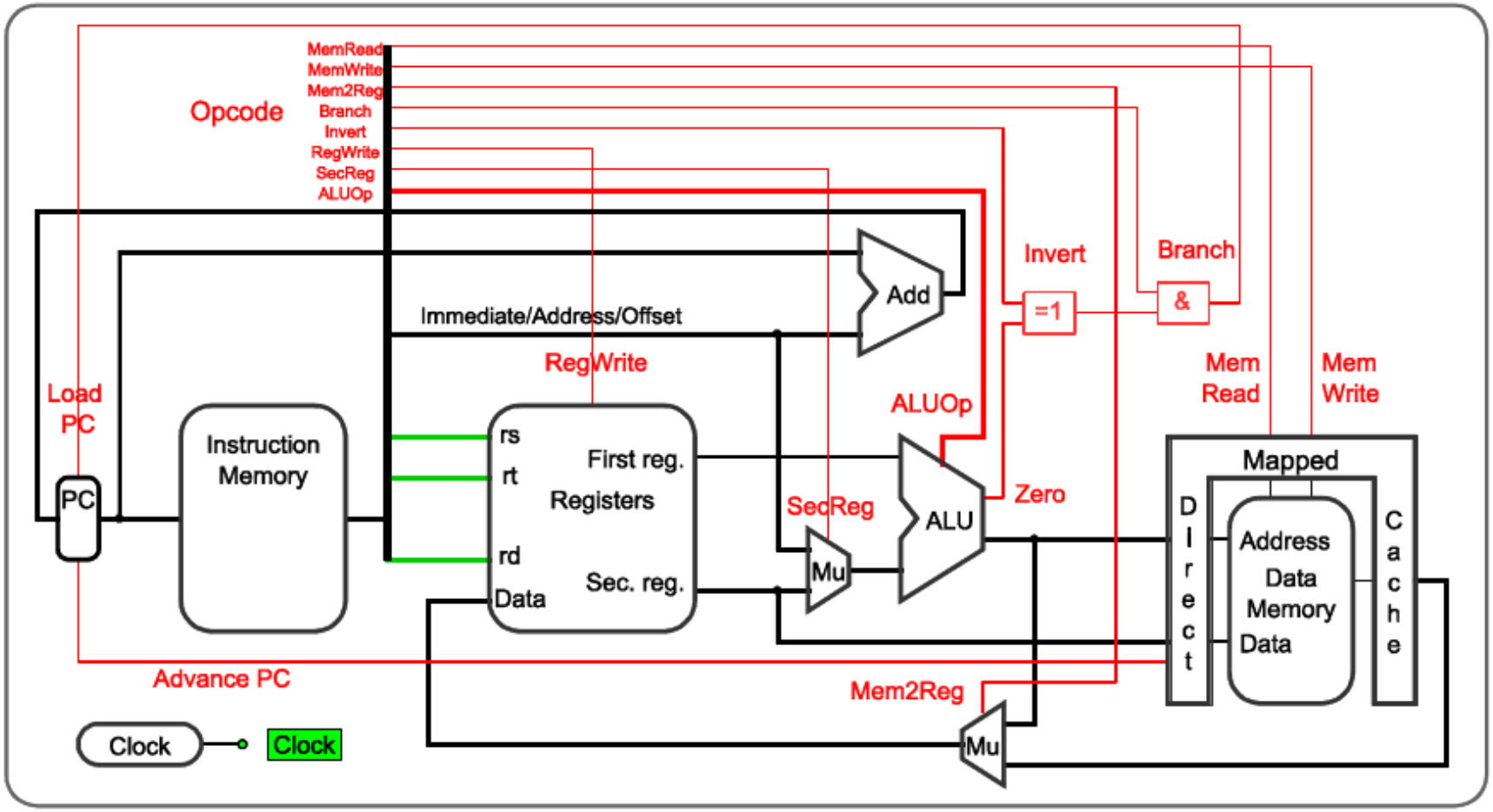


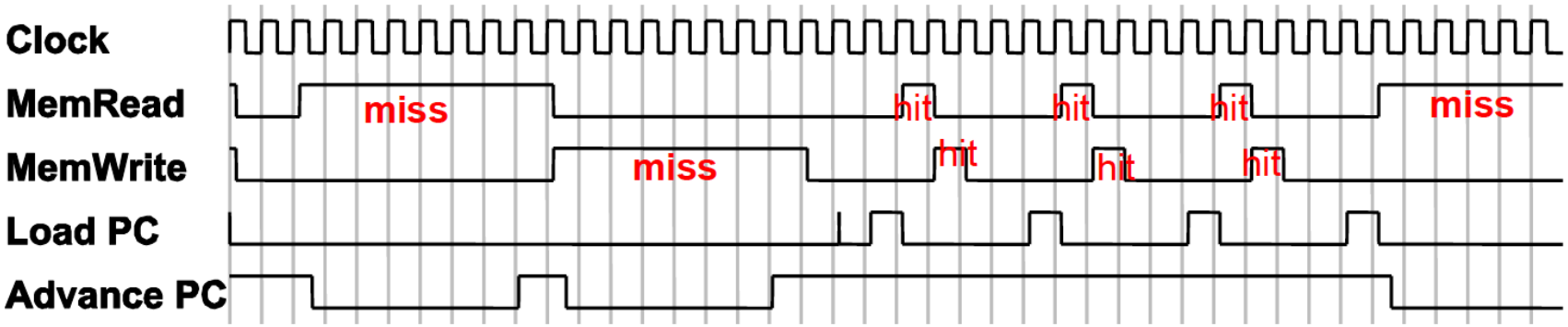
Principle of Locality

- **Spatial Locality** neighboring memory blocks are likely to be accessed contemporary

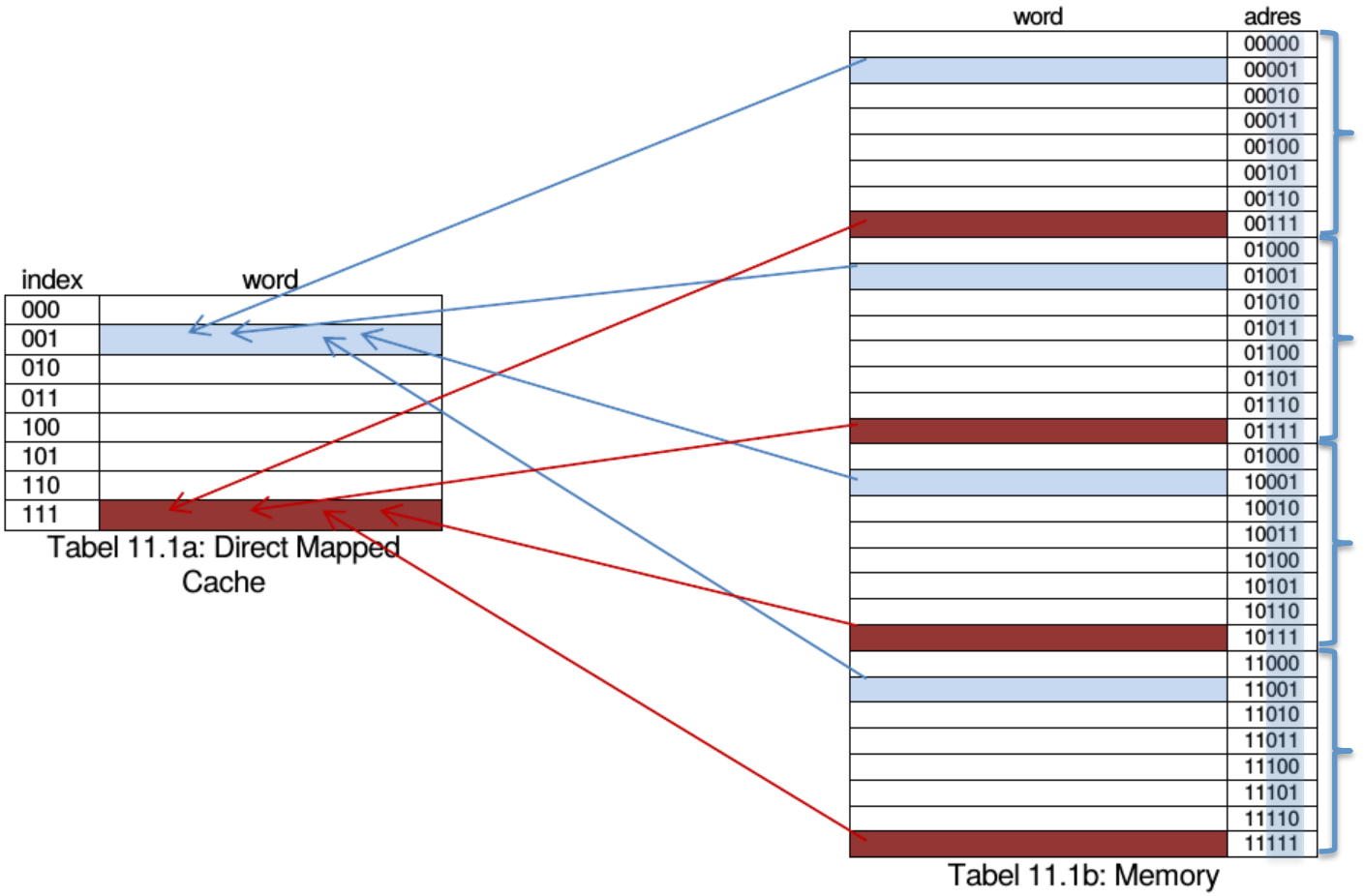
- **Temporal Locality** recently accessed memory blocks are likely to be accessed in the near future again

Harvard Architecture with cache




Clock
MemRead
MemWrite
Load PC
Advance PC
miss
miss
hit
hit
hit
hit
hit
hit
hit
miss

direct-mapped cache



Tabel 11.1a: Direct Mapped Cache

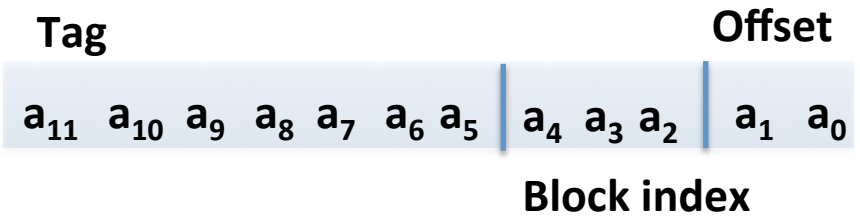
Tabel 11.1b: Memory

Internal cache organization

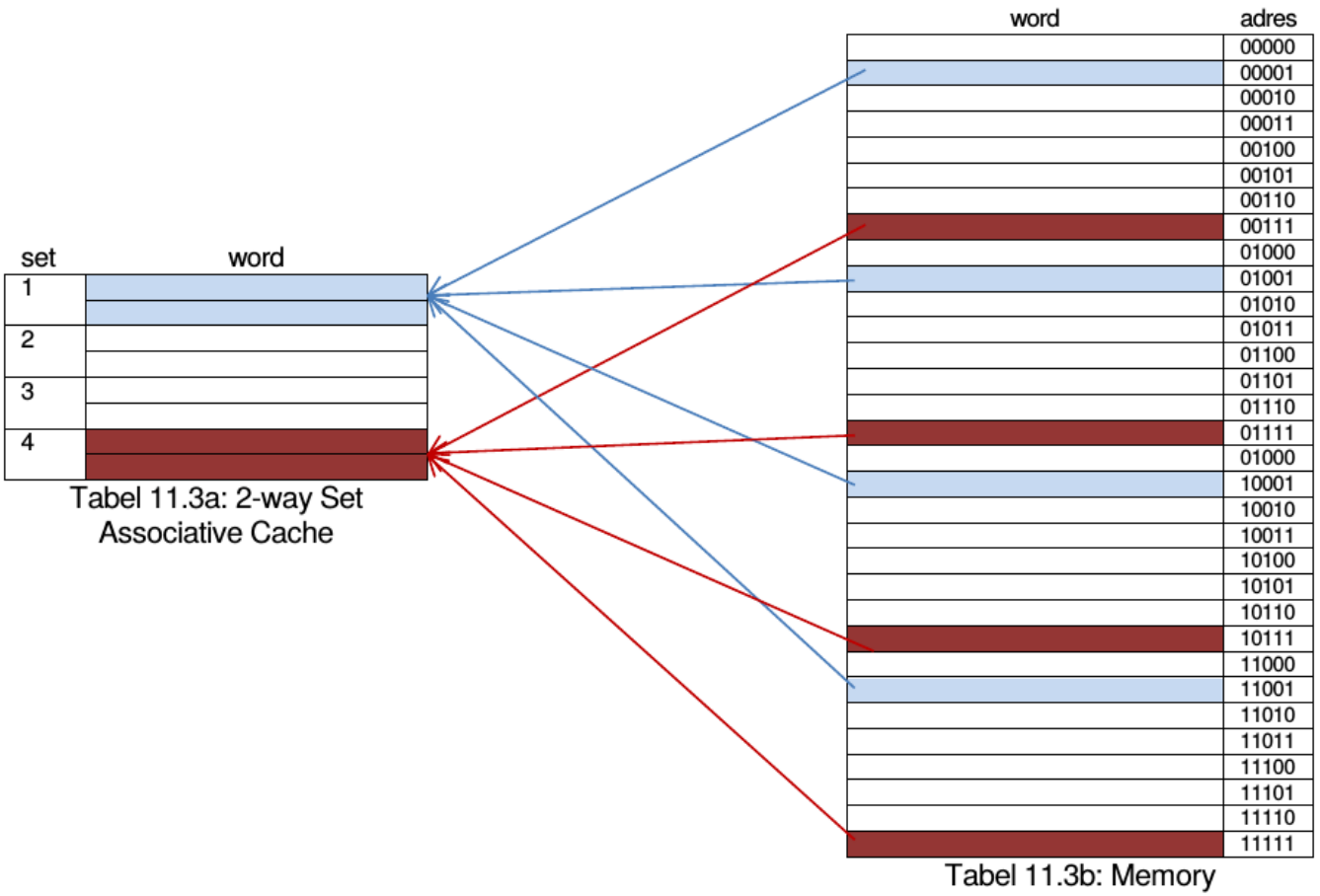
Index	Valid	Tag	Word 4	Word 3	Word 1	Word 0
Nr of bits	1	7	16	16	16	16
000						
001	1	1001001	….	….	….	….
010						
011						
100						
101						
110						
111						

← Cache-line

- Suppose we have a memory of 4096 (2^{12}) words.



2 way set-associative cache



Tabel 11.3a: 2-way Set Associative Cache

Tabel 11.3b: Memory

Different types of cache misses

- **Compulsory** (cold) misses: caches are initially empty, first access is always a miss.
- **Capacity misses** due to the limited cache capacity (i.e. cache is full)
- **Conflict misses** due to an unbalanced cache usage (eviction in one cache set, while other lines are still empty)