50.002 Computation Structures
β Architecture, Instruction Set & Assembly

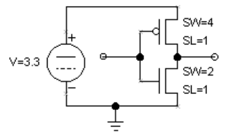
Oliver Weeger

2018 Term 3, Week 4, Session 2

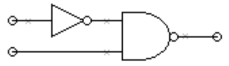
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

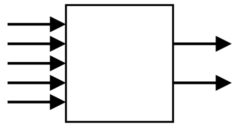
Where are we? The 50.002 roadmap



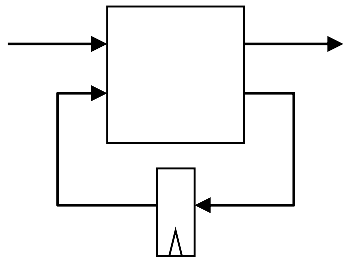
Digital abstraction, Fets & CMOS, Static discipline



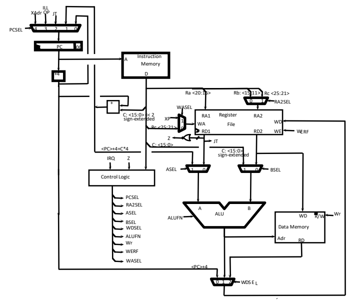
Logic gates & Boolean Algebra (AND, OR, NAND, NOR, etc.)



Combinational logic circuits: Truth tables, Multiplexers, ROMs



Sequential logic & Finite State Machines: Dynamic Discipline, Registers, State Transition Diagrams

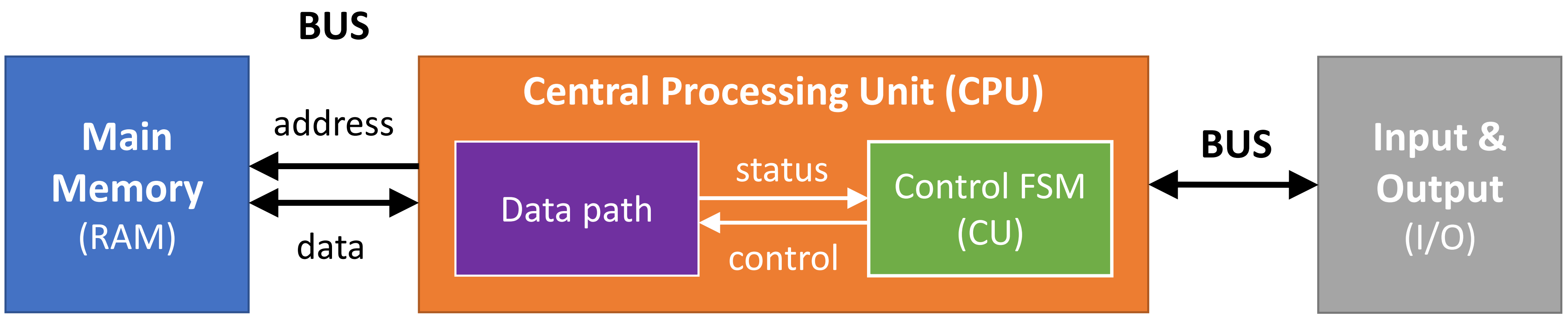


CPU Architecture: *interpreter* for coded *programs*

Use logic to compute mathematical functions, e.g. sums, %3
→ Limitations and models (computability & universality)

Programmability:
- General-Purpose Computer
- Instruction Set Architecture
- Interpretation, Programs, Languages, Translation
- Beta implementation

The von Neumann model – A general-purpose computer



E.g. 16 GB DDR3 64-bit RAM

ROM (read-only memory) &
RAM (randon-access memory),
Array of **words** of k **bits**:
- Early computers:
  8 bits = 1 byte
- Then & “beta”:
  32 bits = 4 bytes
- Now:
  64 bits = 8 bytes

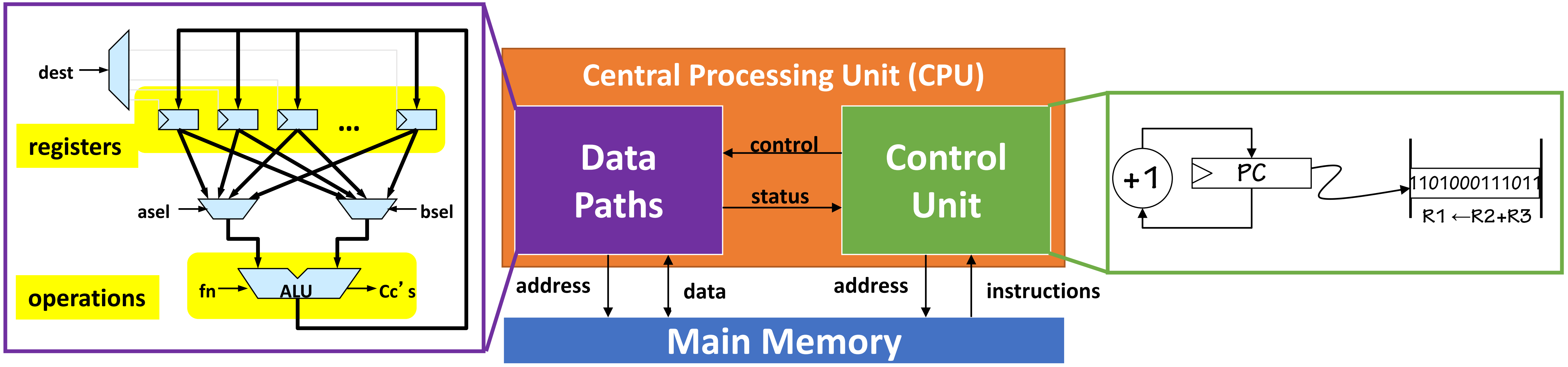
E.g. Intel Core i7 64-bit

- Several **registers** (8, 16, 32, …) in **control unit** (CU – a FSM)
- Data paths (logic) performing specified set of operations (instructions)
  → **Instruction set**
  → **Arithmetic logic unit** (ALU)

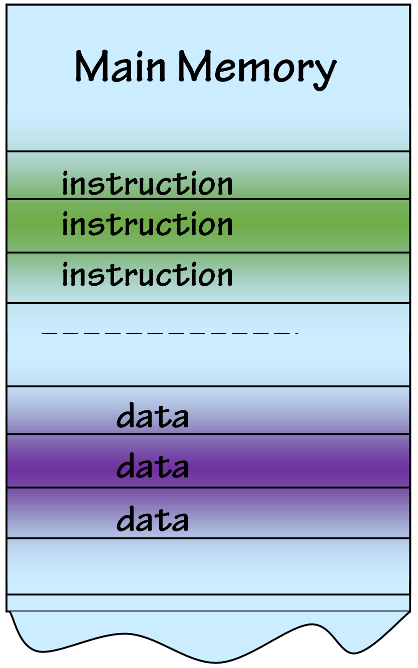
E.g. keyboard, mouse, monitor

Communication with the outside world

Anatomy of von Neumann computer



- **Program Counter (PC)**: Address of next instruction to be executed
- **Instructions** coded as binary data
- **Control unit to interpret (translate) instructions** into control signals for data path
- Logic to **feed data from memory** into registers
- **ALU** to perform operations on data in registers
- PC advances



Pseudo code of FSM:

```
Reset     PC ← 0
Repeat    - CU reads word of instruction
            from mem[PC] & interprets it
          - PC ← PC+1
          - ALU executes instruction
```

Summary of von Neumann machine

- It is a Finite State Machine; state is in PC, registers, memory and input/output devices
- Main components: CPU (CU, PC, registers, ALU), RAM memory, I/O devices, Buses
- Memory contains instructions and data
- PC contains address of next instruction in memory
- CU fetches and interprets instructions, sends signals to ALU, Memory & Registers
- ALU performs operations on registers (not directly on values in memory)
- In every step, the machine executes 1 instruction and advances the PC
- Sequence of instructions is a programme

Instruction Set Architecture

- Need to implement **Arithmetic Logic Unit** (ALU) that performs operations/instructions on **integer binary numbers** (addition, subtraction, and, or, compare, bit shift, etc.),
- Furthermore, we need **memory** instructions (load/save into/from registers) and **PC** instructions (jump, branch)
- Designing an **Instruction Set Architecture** is subject to certain trade-offs:
  - Performance
  - Compactness (size of chip)
  - Programmability
  - Uniformity (size & execution time of instructions)
  - Complexity (how many different instructions)
- **Reduced Instruction Set Computer** (RISC) philosophy: simple instructions, optimized for speed
- $\boldsymbol{\beta}$ architecture: an educational RISC by MIT

β programming model

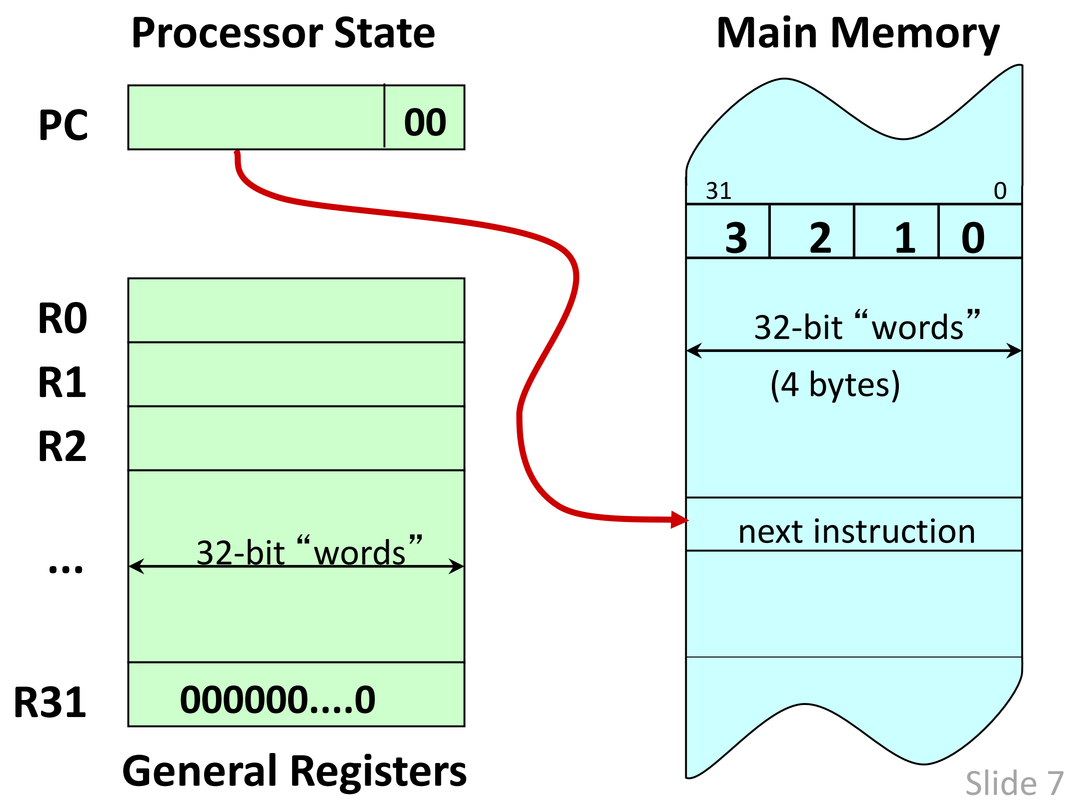
- 32 registers (R0–R31)
- 32-bit (4-byte) words in memory, using byte addressing:

0x 0000 0000	0000 0100	0000 0011	0000 0010	0000 0001
0x 0000 0004	1111 1011	1000 0101	1100 0101	1110 0011
0x 0000 0008	…			
0x 0000 000C				
…				

⇔

0x	04	03	02	01
0x	FB	85	C5	E3
	…			

- Memory contains instructions and data
- Fetch/Execute loop:
  - fetch Mem[PC]
  - PC = PC + 4[†]
  - execute fetched instruction (may change PC!)
  - repeat!



β instruction formats

- 32-bit word with
  - 6-bit OPCODE,
  - Several 5-bit operand locations (referring to registers)
  - One optional 16-bit constant (“literal”)
- OPCODE, 3 register operands (1 destination, 2 sources)

6 bits	5 bits	5 bits	5 bits	11 bits
OPCODE	**r_c**	**r_a**	**r_b**	***unused***

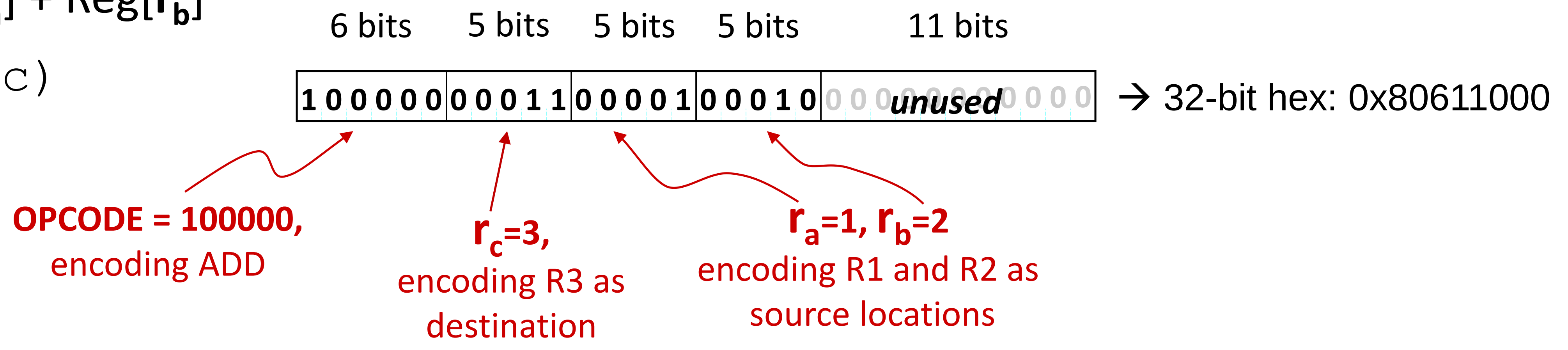
- OPCODE, 2 register operands (1 destination, 1 source) and 16bit literal constant

6 bits	5 bits	5 bits	16 bits
OPCODE	**r_c**	**r_a**	**16-bit signed constant**

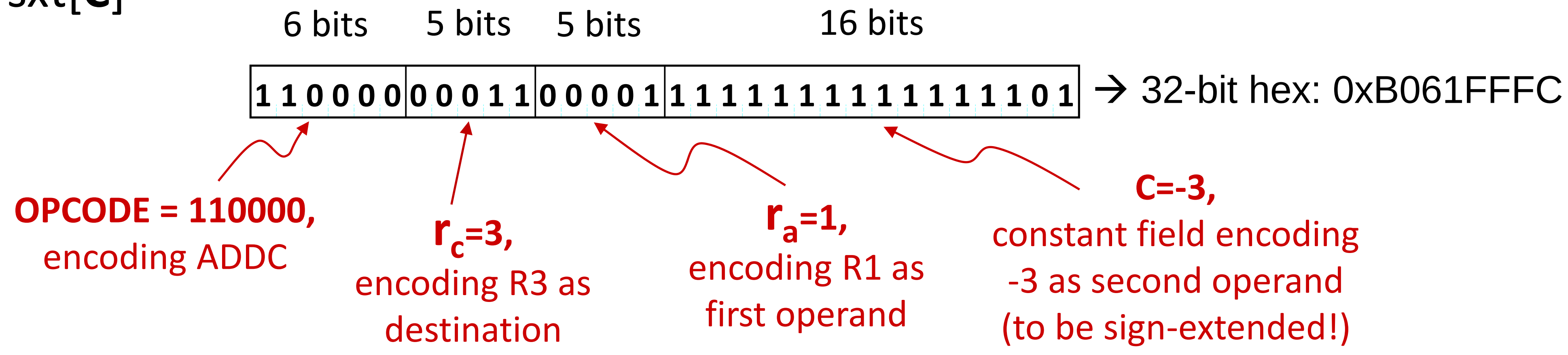
→ Need to define OPCODE and format for all [ALU], [Mem] and [PC] instructions

β [ALU] instructions: ADD & ADDC

- "Add the contents of $\mathbf{r_a}$ to the contents of $\mathbf{r_b}$ and store the result in $\mathbf{r_c}$"
- $\mathrm{Reg}[\mathbf{r_c}] \leftarrow \mathrm{Reg}[\mathbf{r_a}] + \mathrm{Reg}[\mathbf{r_b}]$
- `ADD(ra,rb,rc)`



- "Add the contents of $\mathbf{r_a}$ to **C** and store the result in $\mathbf{r_c}$"
- $\mathrm{Reg}[\mathbf{r_c}] \leftarrow \mathrm{Reg}[\mathbf{r_a}] + \mathrm{sxt}[\mathbf{C}]$
- `ADDC(ra,C,rc)`


6 bits
5 bits
5 bits
16 bits
1 1 0 0 0 0
0 0 0 1 1
0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
→ 32-bit hex: 0xB061FFFC
OPCODE = 110000, encoding ADDC
r_c=3, encoding R3 as destination
r_a=1, encoding R1 as first operand
C=-3, constant field encoding -3 as second operand (to be sign-extended!)

β [Mem] instructions: LD & ST

- Load: "Load into $\mathbf{r_c}$ the contents of the memory location whose address is the content of $\mathbf{r_a}$ plus **CONST**"
- Reg[$\mathbf{r_c}$] ← Mem[Reg[$\mathbf{r_a}$] + sxt(**CONST**)]
- `LD(ra,Const,rc)`
  or `LD(Const,rc)=LD(R31,Const,rc)`

6 bits	5 bits	5 bits	16 bits
OPCODE	$\mathbf{r_c}$	$\mathbf{r_a}$	**16-bit signed CONST**

- Store: "Store the contents of $\mathbf{r_c}$ into the memory location whose address is the content of $\mathbf{r_a}$ plus **CONST**"
- Mem[Reg[$\mathbf{r_a}$] + sxt(**CONST**)] ← Reg[$\mathbf{r_c}$]
- `ST(rc,Const,ra)`
  or `ST(rc,Const)=ST(rc,Const,R31)`

BYTE ADDRESSES, but only 32-bit/4-byte word accesses to word-aligned addresses are supported. Low two address bits are ignored!

β [PC] instructions: BEQ, BNE & JMP

- Branch instructions for conditionals: "If $\mathbf{r_a}$ is 0 (not 0), save the current location (PC) into $\mathbf{r_c}$ and continue at **label** location (add **CONST** to PC)"

- `BEQ(ra,label,rc)` (branch if equal)

```
PC = PC + 4;
Reg[rc] = PC;
if (REG[ra] == 0)
    PC = PC + 4*CONST
```

`BNE(ra,label,rc)` (branch if not equal)

```
PC = PC + 4;
Reg[rc] = PC;
if (REG[ra] != 0)
    PC = PC + 4*CONST
```

6 bits	5 bits	5 bits	16 bits
OPCODE	$\mathbf{r_c}$	$\mathbf{r_a}$	**16-bit signed CONST**

CONST = (<u>label</u> - <addr of BNE/BEQ>)/4 – 1
(up to 32767 instructions before/after BNE/BEQ)

- Here, the **<u>label</u>** refers directly to an address, which needs to be converted to the **CONST** that specifies the word offset of the address from the current PC.

- Abbreviations:

```
BEQ(ra,label)=BEQ(ra,label,R31)
BNE(ra,label)=BNE(ra,label,R31)
```

- Unconditional branches:

```
BR(label,rc)=BEQ(R31,label,rc)
BR(label)   =BEQ(R31,label,R31)
```

Overview of β instruction set

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

3 registers:

OPCODE	r_c	r_a	r_b	*unused*

arithmetic: ADD, SUB, MUL, DIV
compare: CMPEQ, CMPLT, CMPLE
boolean: AND, OR, XOR
shift: SHL, SHR, SRA

Ra and Rb are the operands,
Rc is the destination.
R31 reads as 0, unchanged by writes

2 registers, 1 const:

OPCODE	r_c	r_a	16-bit signed constant

arithmetic: ADDC, SUBC, MULC, DIVC
compare: CMPEQC, CMPLTC, CMPLEC
boolean: ANDC, ORC, XORC
shift: SHLC, SHRC, SRAC
branch: BNE/BT, BEQ/BF (const = word displacement from PC_{NEXT})
jump: JMP (const not used)
memory access: LD, ST (const = byte offset from Reg[ra])

Two's complement 16-bit constant for numbers from −32768 to 32767; sign-extended to 32 bits before use.

6-bit OPCODES:

2:0 / 5:3	000	001	010	011	100	101	110	111
000								
001								
010								
011	LD	ST		JMP		BEQ	BNE	LDR
100	ADD	SUB	MUL*	DIV*	CMPEQ	CMPLT	CMPLE	
101	AND	OR	XOR		SHL	SHR	SRA	
110	ADDC	SUBC	MULC*	DIVC*	CMPEQC	CMPLTC	CMPLEC	
111	ANDC	ORC	XORC		SHLC	SHRC	SRAC	

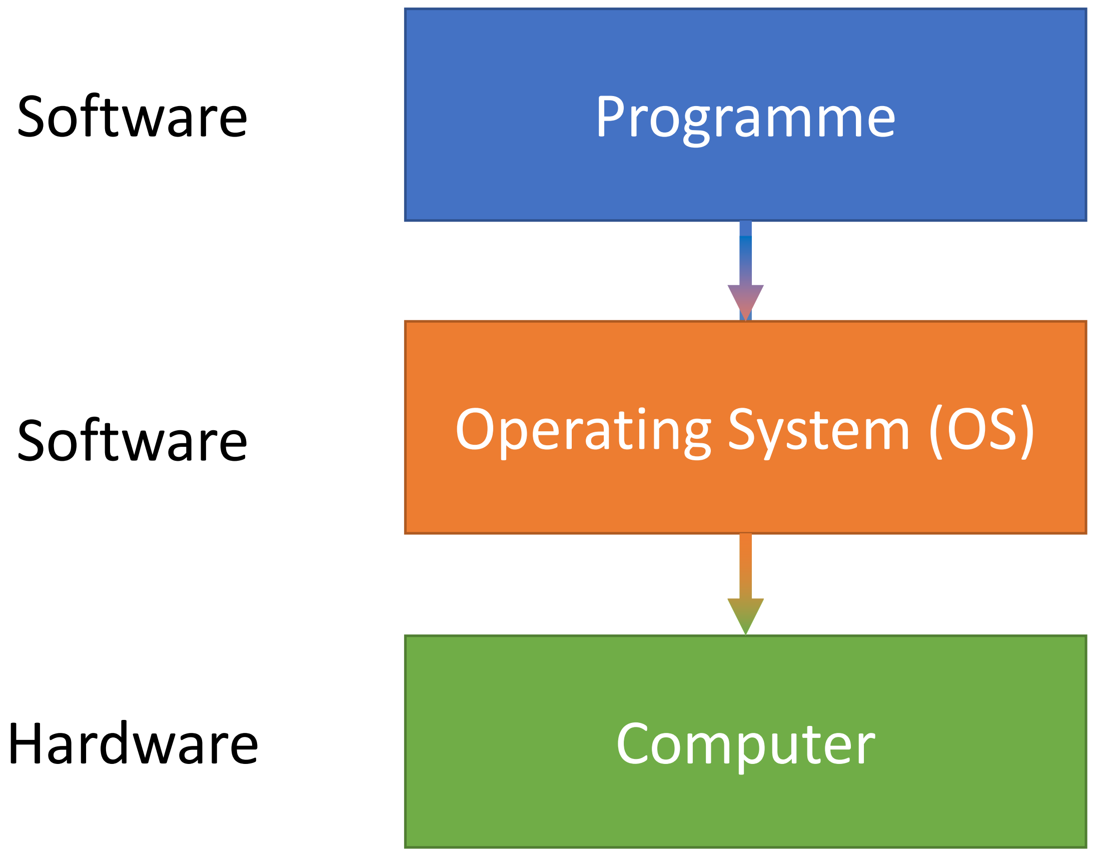
An example programme: Factorial

Input: $n = 3$

Output: $y = n!$

"C" code:	$\boldsymbol{\beta}$ assembly code:		memory location:	memory value:
`int n = 3;`	`n:`	`int32(3)`	`0x00001000`	`0x00000003` `00000000000000000000000000000011`
`int y = 0;`	`y:`	`int32(0)`	`0x00001004`	`0x00000000` `00000000000000000000000000000000`
`r1 = 1;`		`ADDC(r31,1,r1)`	`0x00001008`	`0xC03F0001` `11000000001111110000000000000001`
`r2 = n;`		`LD(n,r2)`	`0x0000100C`	`0x605FFFF4` `01100000010111110001000000000000`
`while (r2 != 0) {`	`loop:`	`BEQ(r2,done,r31)`	`0x00001010`	…
`    r1 = r1 * r2;`		`MUL(r1,r2,r1)`	`0x00001014`	`0x88211000` `10001000001000010001000000000000`
`    r2 = r2 - 1;`		`SUBC(r2,1,r2)`	`0x00001018`	…
`}`		`BEQ(r31,loop,r31)`	`0x0000101C`	
`y = r1;`	`done:`	`ST(r1,y,r31)`	`0x00001020`	

Computers & programmes



- Assembly code
- Compiled local: C, C++, Fortran, …
- Interpreted: Java, Python, …

- Loaded at reset and running perpetually
- Manages programmes
- Virtual machines for interpreting code

Summary

- $\boldsymbol{\beta}$ is a von Neumann computer
- Main components: CPU (CU, PC, registers, ALU), RAM memory, I/O devices, buses
- Need instruction set providing operations (ALU, Mem, PC)
- Instruction format: 32-bit words
- Programmes, compilation, OS