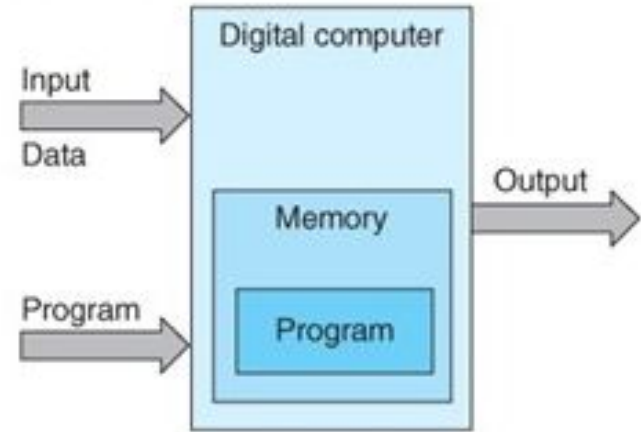


# CASS Exercise session 2

Introduction to RISC-V

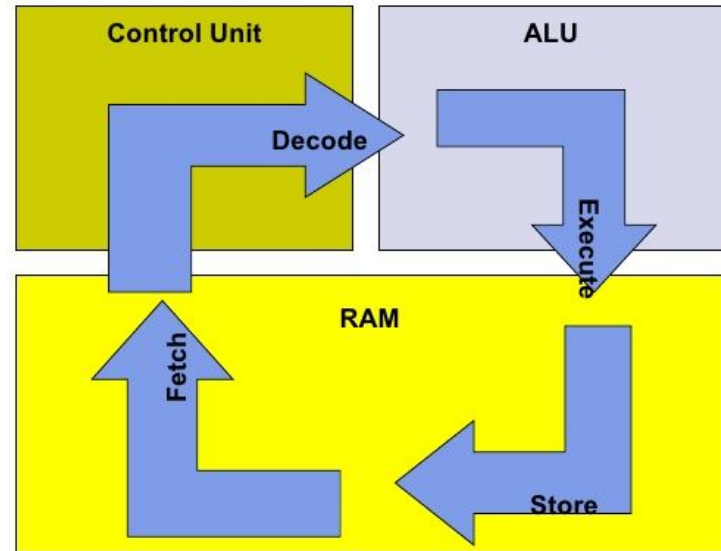
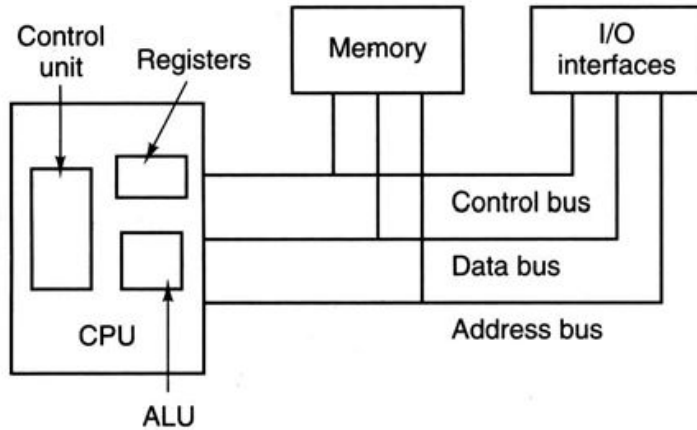
# Stored-program computer

- Stores program instructions in electronic memory
- Von Neumann architecture
  - storing program instructions in memory *along* with the data
  - the majority of modern computers



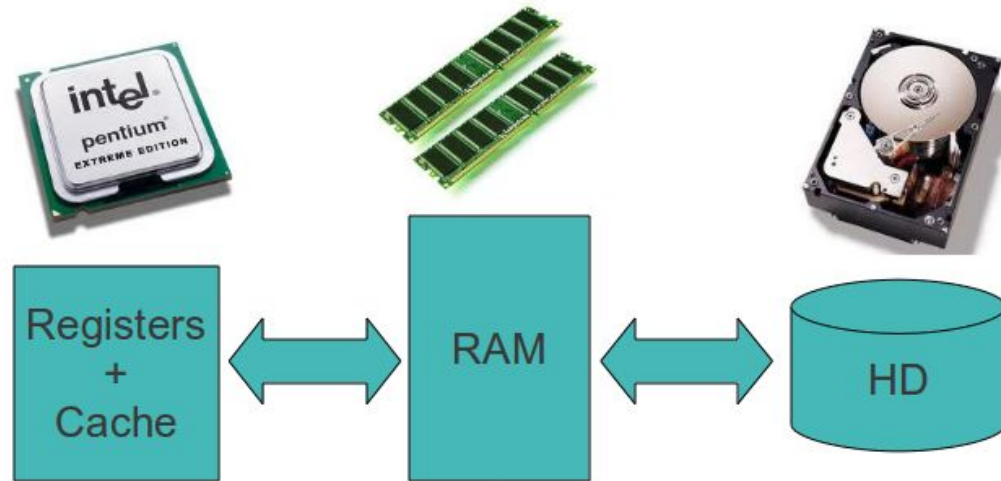
# Basic computer operational process

## Fetch Decode Execute Cycle



# Processor registers

- Temporary storage locations
- Extremely fast on-chip memory
- Limited amount
- Include [special-purpose registers](#)



# What is assembly/machine language?

- The only language a processor “understands”
- Manipulates the hardware directly on the level of individual instructions
- Directly uses processor on-chip registers
  - Faster than memory – improves performance
  - Most assembly instructions (only) operate on registers
  - Registers are a finite resource that need to be managed

# What is assembly/machine language?

- The only language a processor “understands”
- Manipulates the hardware directly on the level of individual instructions
- Directly uses processor on-chip registers
- Limited set of instructions, different for each architecture
  - x86, ARM, PowerPC, MIPS, **RISC-V**, AVR...
- Two different representations
  - 01101001110100101001110010101100 (binary form)
  - **addi** *t1*, *t2*, 12 (mnemonic form)
- C requires no knowledge of registers
  - Compiler takes care of this

# RISC-V Instruction Set Architecture

- This course focuses on RISC-V assembly
  - **Open standard** Instruction Set Architecture (ISA)
    - Open source ISA !!
    - Ecosystem of open-source hardware, toolchaining, OSes, ...
  - ISA of the future?
- Reduced Instruction Set Computer (RISC)
  - Simple, elegant, easy in usage
  - Good for teaching principles
  - x86: CISC (Complex)
- Easy to learn other assembly ‘dialects’

# RISC-V ISA

- 32 General Registers (x0 - x31)
  - zero – the hardwired value 0
  - a0 - a7 – for arguments and return values
  - s0, s1, ... s11 – for save values
  - t0, t1, ... t6 – for temporary values
  - sp – stack pointer
  - fp – frame pointer - *Mapped to the same register as s0! (x8)*
  - gp – global area pointer
  - tp – thread pointer
  - ra – return address
- Other registers
  - Control and status
  - Floating Point
  - Vector



# RISC-V ISA

- Registers can be 32-bit, 64-bit or 128-bit
  - 1 register used for addressing RAM
  - 32-bit:  $2^{32}$  possible different addresses (byte-addressing)
    - Size of RAM limited to ~4GB:  $(2^{32}) * 8 \text{ bits} = 4.29\text{GB}$
  - 64-bit: Size of RAM limited to  $2^{64}$  bytes = 16 exabytes
- Processor designer: choose word-width + extensions
  - RV32I: 32-bit registers with Integer Instruction Set
  - RV32E: 32-bit registers with Integer Instruction Set, only 15 total registers
  - RV64IF: 64-bit registers with Integer Instruction Set and Floating Point instruction set
  - RV128I: 128-bit registers with Integer Instruction Set
- Great overview
  - <https://github.com/riscv/riscv-asm-manual/>

# RARS simulator

- To run RISC-V you need a RISC-V processor
- RARS: Simulator for RISC-V assembly
  - <https://github.com/TheThirdOne/rars>
  - Install for exercise sessions!
- Supports RV32 and RV64 with many extensions
- Exercise sessions: RV32 with Integer instruction set and 32 registers

# Breakdown of assembly instructions

- Basic type instruction:

Operation	Destination	1st source	2nd source
<b>add</b>	<b>s1</b>	<b>s2</b>	<b>s3</b>

C:  $a = b + c$

# Breakdown of assembly instructions

- Basic type instruction:

Operation	Destination	1st source	2nd source
<b>add</b>	<b>s1</b>	<b>s2</b>	<b>s3</b>

C:  $a = b + c$

- More complex statements?
  - Break into multiple instructions:

Operation	Destination	1st source	2nd source
<b>add</b>	<b>t0</b>	<b>s1</b>	<b>s2</b>
<b>add</b>	<b>t1</b>	<b>t0</b>	<b>s3</b>
<b>add</b>	<b>s0</b>	<b>t1</b>	<b>s4</b>

C:  $a = b + c + d - e$

#  $t0 = b + c$

#  $t1 = t0 + d$

#  $a = t1 - e$

# Simple RISC-V program

simple-sum.asm

```
addi t0, zero, 5
addi t1, zero, 6
add t2, t0, t1
```

Registers

zero	0x00000000
....	...
t0	0x00000005
t1	0x00000006
t2	0x0000000b
...	...

# Simple RISC-V program

simple-sum.asm

Zero register always contains value 0,  
hard wired!

```
addi t0, zero, 5
addi t1, zero, 6
add  t2, t0, t1
```

*addi* adds an immediate value in  
between  $-2^{31}$  and  $2^{31} - 1$

For *add* the operands must be  
registers

Registers

zero	0x00000000
....	...
t0	0x00000005
t1	0x00000006
t2	0x0000000b
...	...

# Main memory & Segments

main-memory.asm

```
.data
    myvalue: .word    10
.globl main
.text
main:
    la  a0, myvalue
    lw  t0, 0(a0)
    add t0, t0, t0
    sw  t0, 4(a0)
```

Registers

...	...
a0	0x10010000
....	...
t0	0x00000014
...	...

Main Memory

...	...
0x10010000	0x0000000a
0x10010004	0x00000014
...	...



# Main memory & Segments

Segment of file where data is declared

```
.data
    myvalue: .word    10
.globl main
.text
main:
    la  a0, myvalue
    lw  t0, 0(a0)
    add t0, t0, t0
    sw  t0, 4(a0)
```

Segment of file where instructions are declared

Registers

...		...
a0		0x10010000
....		...
t0		0x00000014
...		...

Main Memory

...	...
0x10010000	0x0000000a
0x10010004	0x00000014
...	...



# Main memory & Segments

main\_memory.asm

Needed so main is visible externally

```
.data
    myvalue: .word    10
.globl main
.text
main:
    la  a0, myvalue
    lw  t0, 0(a0)
    add t0, t0, t0
    sw  t0, 4(a0)
```

Registers

...	...
a0	0x10010000
....	...
t0	0x00000014
...	...

Main Memory

...	...
0x10010000	0x0000000a
0x10010004	0x00000014
...	...

# Data segment: data types

data-segment.asm

```
.data
empty: .space 4
numb:  .word 11
abcd:  .string "ABCD"
array: .word 1, 2, 3, 4
doubl: .double 99.99, 100.1
```

Main Memory

0x10010000	0x00000000	0x10010020	0x28f5c28f
0x10010004	0x0000000b	0x10010024	0x4058ff5c
0x10010008	0x44434241	0x10010028	0x66666666
0x1001000c	0x00000000	0x1001002c	0x40590666
0x10010010	0x00000001	0x10010030	????
0x10010014	0x00000002	0x10010034	????
0x10010018	0x00000003	0x10010038	????
0x1001001c	0x00000004	0x1001003c	????

# Q: why is 0x1001000c empty?

data-segment.asm

```
.data
empty: .space 4
numb: .word 11
abcd: .string "ABCD"
array: .word 1, 2, 3, 4
doubl: .double 99.99, 100.1
```

Hint! Think about how strings are represented in memory

Main Memory

0x10010000	0x00000000	0x10010020	0x28f5c28f
0x10010004	0x0000000b	0x10010024	0x4058ff5c
0x10010008	0x44434241	0x10010028	0x66666666
0x1001000c	0x00000000	0x1001002c	0x40590666
0x10010010	0x00000001	0x10010030	????
0x10010014	0x00000002	0x10010034	????
0x10010018	0x00000003	0x10010038	????
0x1001001c	0x00000004	0x1001003c	????

# Answer: 0x1001000c stores zero termination byte

data-segment.asm

```
.data
empty: .space 4
numb: .word 11
abcd: .string "ABCD"
array: .word 1, 2, 3, 4
doubl: .double 99.99, 100.1
```

Main Memory

0x10010000	0x00000000	0x10010020	0x28f5c28f
0x10010004	0x0000000b	0x10010024	0x4058ff5c
0x10010008	0x44434241	0x10010028	0x66666666
0x1001000c	0x00000000	0x1001002c	0x40590666
0x10010010	0x00000001	0x10010030	????
0x10010014	0x00000002	0x10010034	????
0x10010018	0x00000003	0x10010038	????
0x1001001c	0x00000004	0x1001003c	????

# Branches

branches.asm

```
.data
    result: .word 5
.globl main
.text
main:
    la    a0, result
    lw    t0, 0(a0)
    addi  t1, zero, 1
    addi  t2, zero, 1
loop:
    add   t3, t1, t2
    mv    t1, t2
    mv    t2, t3
    addi  t0, t0, -1
    bnez  t0, loop
    sw    t3, 0(a0)
```

Registers

...	...
a0	0x10010000
...	...
t0	0x00000005
t1	0x00000001
t2	0x00000001
t3	0x00000000

Memory and register state before  
loop entry shown

Main Memory

...	...
0x10010000	0x00000005
...	...

# Branches

branches.asm

```
.data
    result: .word 5
.globl main
.text
main:
    la    a0, result
    lw    t0, 0(a0)
    addi  t1, zero, 1
    addi  t2, zero, 1
loop:
    add   t3, t1, t2
    mv    t1, t2
    mv    t2, t3
    addi  t0, t0, -1
    bnez  t0, loop
    sw    t3, 0(a0)
```

Registers

...	...
a0	0x10010000
...	...
t0	0x00000005
t1	0x00000001
t2	0x00000001
t3	0x00000002
...	...

Main Memory

...	...
0x10010000	0x00000005
...	...

# Branches

branches.asm

```
.data
    result: .word 5
.globl main
.text
main:
    la    a0, result
    lw    t0, 0(a0)
    addi  t1, zero, 1
    addi  t2, zero, 1
loop:
    add   t3, t1, t2
    mv    t1, t2
    mv    t2, t3
    addi  t0, t0, -1
    bnez  t0, loop
    sw    t3, 0(a0)
```

Registers

...	...
a0	0x10010000
...	...
t0	0x00000005
t1	0x00000001
t2	0x00000001
t3	0x00000002
...	...

Main Memory

...	...
0x10010000	0x00000005
...	...

# Branches

branches.asm

```
.data
    result: .word 5
.globl main
.text
main:
    la    a0, result
    lw    t0, 0(a0)
    addi  t1, zero, 1
    addi  t2, zero, 1
loop:
    add   t3, t1, t2
    mv    t1, t2
    mv    t2, t3
    addi  t0, t0, -1
    bnez  t0, loop
    sw    t3, 0(a0)
```

Registers

...	...
a0	0x10010000
...	...
t0	0x00000005
t1	0x00000001
t2	0x00000002
t3	0x00000002
...	...

Main Memory

...	...
0x10010000	0x00000005
...	...



# Branches

branches.asm

```
.data
    result: .word 5
.globl main
.text
main:
    la    a0, result
    lw    t0, 0(a0)
    addi  t1, zero, 1
    addi  t2, zero, 1
loop:
    add   t3, t1, t2
    mv    t1, t2
    mv    t2, t3
    addi  t0, t0, -1
    bnez  t0, loop
    sw    t3, 0(a0)
```

Registers

...	...
a0	0x10010000
...	...
t0	0x00000004
t1	0x00000001
t2	0x00000002
t3	0x00000002
...	...

Main Memory

...	...
0x10010000	0x00000005
...	...

# Branches

branches.asm

```
.data
    result: .word 5
.globl main
.text
main:
    la    a0, result
    lw    t0, 0(a0)
    addi  t1, zero, 1
    addi  t2, zero, 1
loop:
    add   t3, t1, t2
    mv    t1, t2
    mv    t2, t3
    addi  t0, t0, -1
    bnez  t0, loop
    sw    t3, 0(a0)
```

Registers

...	
a0	
...	...
t0	0x00000004
t1	0x00000001
t2	0x00000002
t3	0x00000002
...	...

t0 not equal to zero?  
True -> jump to loop  
False -> continue normally

Main Memory

...	...
	0x00000005
...	...

# Q: what will be in stored in result after execution?

## branches.asm

```
.data
    result: .word 5
.globl main
.text
main:
    la    a0, result
    lw    t0, 0(a0)
    addi  t1, zero, 1
    addi  t2, zero, 1
loop:
    add   t3, t1, t2
    mv    t1, t2
    mv    t2, t3
    addi  t0, t0, -1
    bnez  t0, loop
    sw    t3, 0(a0)
```

## Registers

...	...
a0	0x10010000
...	...
t0	0x00000004
t1	0x00000001
t2	0x00000002
t3	0x00000002

Memory and register state after  
first iteration

## Main Memory

...	...
0x10010000	0x00000005
...	...

# Answer: 7th element of Fibonacci sequence (13)

## branches.asm

```
.data
    result: .word 5
.globl main
.text
main:
    la    a0, result
    lw    t0, 0(a0)
    addi  t1, zero, 1
    addi  t2, zero, 1
loop:
    add   t3, t1, t2
    mv    t1, t2
    mv    t2, t3
    addi  t0, t0, -1
    bnez  t0, loop
    sw    t3, 0(a0)
```

## Registers

...	...
a0	0x10010000
...	...
t0	0x00000000
t1	0x00000008
t2	0x0000000d
t3	0x0000000d

Memory and register state after execution

## Main Memory

...	...
0x10010000	0x0000000d
...	...

# C to RISC-V

## conditionals.c

```
int a = 5;
int b = 6;
int* c;

int main() {
    c = &a;
    if(a < b) {
        b -= a;
    }
    else {
        a -= b;
    }
    *c = 10;
}
```

## conditionals.asm

```
.data
    a: .word 5
    b: .word 6
    c: .space 4
.globl main
.text
main:
    la a0, a
    sw a0, c, t3
    la a1, b
    lw t0, 0(a0)
    lw t1, 0(a1)
    slt t2, t0, t1
    beqz t2, else
    sub t1, t1, t0
    sw t1, b, t3
    j endif
else:
    sub t0, t0, t1
    sw t0, a, t3
endif:
    lw a0, c
    addi t0, zero, 10
    sw t0, 0(a0)
```

## Registers & Main Memory

a0	0x10010000
a1	0x10010004
t0	0x0000000a
t1	0x00000001
t2	0x00000001
...	...

...	...
0x10010000	0x0000000a
0x10010004	0x00000001
0x10010008	0x10010000
...	...

# Other useful instructions

- Arithmetic
  - add, addi, mul, div, srl, sll, ...
- Relational
  - slt, ...
- Logic
  - or, xor, ...
- Control
  - beq, blt, bne, j, jr, ...
- Data Transfer
  - lw, sw, lh, sh, lb, sb, ...
- RARS gives description of instructions

# Start the exercises

- Launch the installed RARS simulator
  - If not installed: download jar @ <https://github.com/TheThirdOne/rars>
  - Java is needed to run the application
  - Execute `java -jar <path-to-jar-file>` in console
- In RARS go to settings
  - Enable “Initialize program counter to global main if defined”
- Using RARS
  - Write RISC-V in the “Edit” window
  - Assemble using the wrench icon
  - Run using the green arrow
    - The other arrows can be used for single stepping instructions
  - Memory and registers contents are displayed during/after execution

# Extra: objdump

- After compiling C-code with gcc we generate an executable
  - File with multiple segments just like RISC-V source
- We can view this in readable format using objdump
  - Example: `objdump -S -M intel <executable_name>`
  - “-M intel” specifies the way the assembly code is displayed (Intel syntax instead of AT&T)
  - (Mac OS: `otool -tV <executable_name>`)
- Code usually a lot bigger than manual translation