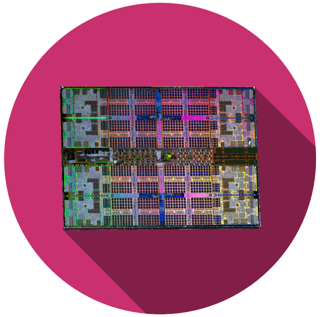
Computer Architecture

Instruction Set Architecture

ISA

Information and Communication Systems program

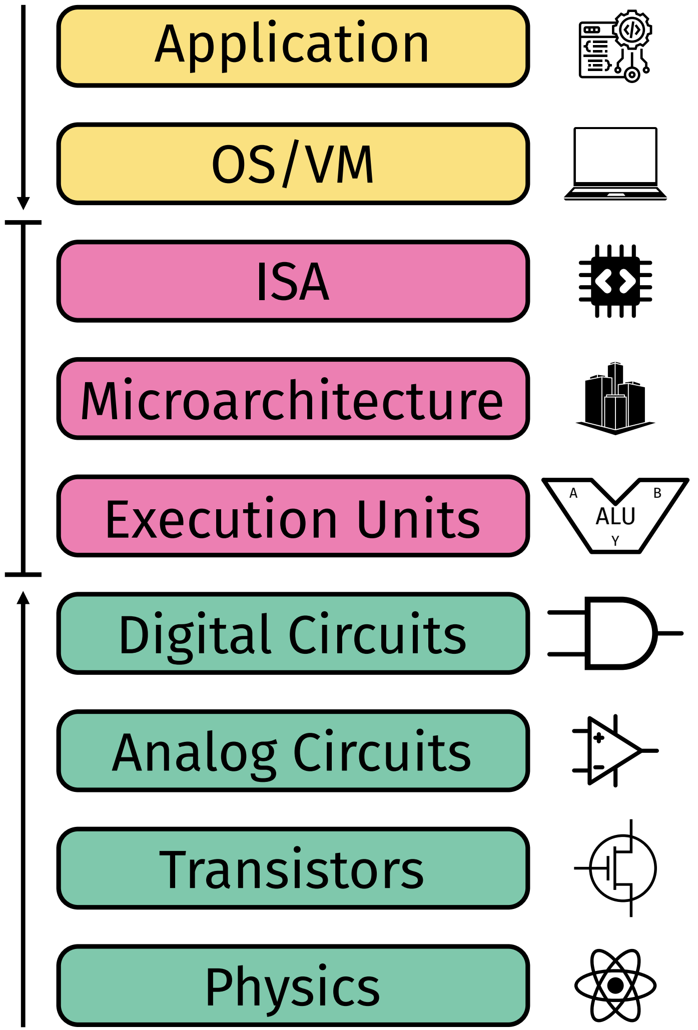


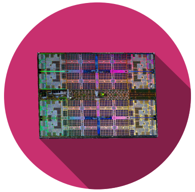
Silvan Zahno silvan.zahno@hevs.ch

Where are we in the course?

Instruction Set Architecture

- Assembly language
- Programming
- Machine language
- Addressing Modes
- Compiling, Assembly & Loading





Assembly language syntax

Assembly Language: human readable format of instructions

```
init: addi    s0,        s0,     4         ; s0 = s0 + 4
```

Assembly language syntax

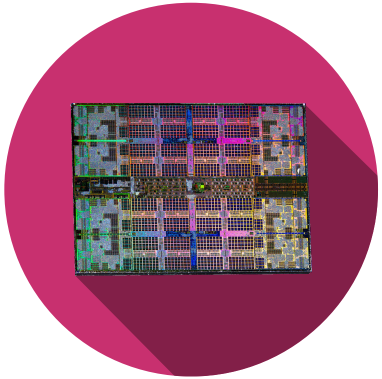
Assembly Language: human readable format of instructions

```
Label OpCode   Destination Source   Source   Comment
      mnemonic Operand     Operand1 Operand2
init: addi     s0,         s0,      4        ; s0 = s0 + 4
```

Assembly language syntax

Machine language: computer-readable format

```
0b00001000 000000000100 00000 000 00000 0010011
```



Classes of Instructions

Classes of Instructions

- Data Transfer
  - `lb, lh, lw, sb, sh, sw, lui`
- ALU
  - `add, sub, and, or, xor, mul, div, rem, sll, srl`
- Control Flow
  - `beq, bge, jal, ecall, ebreak`
- Floating Point
  - `fmadd, fmsub, fadd, fsgnj, fmul, feq`
- SIMD / Vector
  - `vmul`
- String
  - REP MOVSB (x86)

Classes of Instructions

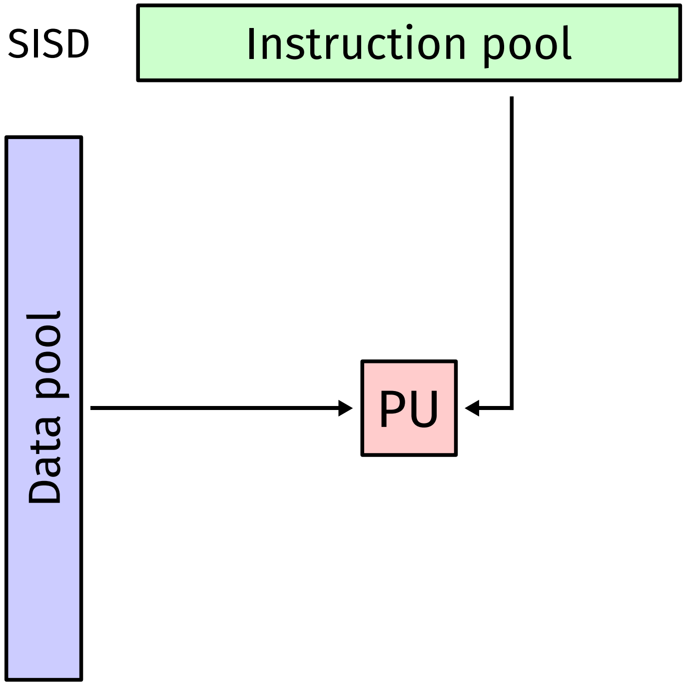
Flynn's Classifications - SISD – Single Instruction Single Data

- Based on the Von Neumann architecture
- Pipelining can be implemented but only one instruction is executed at any given time

Advantages – Cheap, low power

Disadvantages – limited speed

Examples - Microcontroller



Classes of Instructions

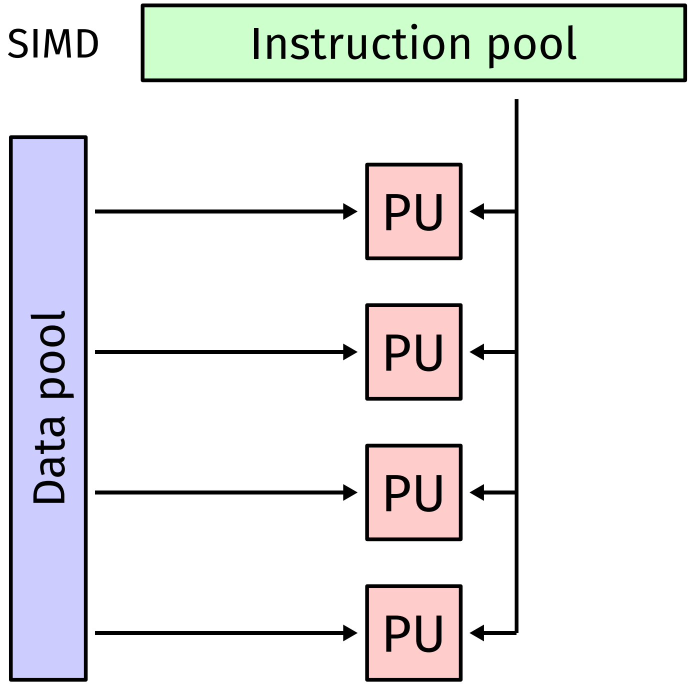
Flynn’s Classifications - SIMD – Single Instruction Multiple Data

- Single instruction is executed on multiple pieces of data
- Instructions can be sequential, or parallel

Advantages – Very efficient on big data

Disadvantages – limited to specific applications

Examples – GPU, Vector Processor, Array processors, Scientific processing

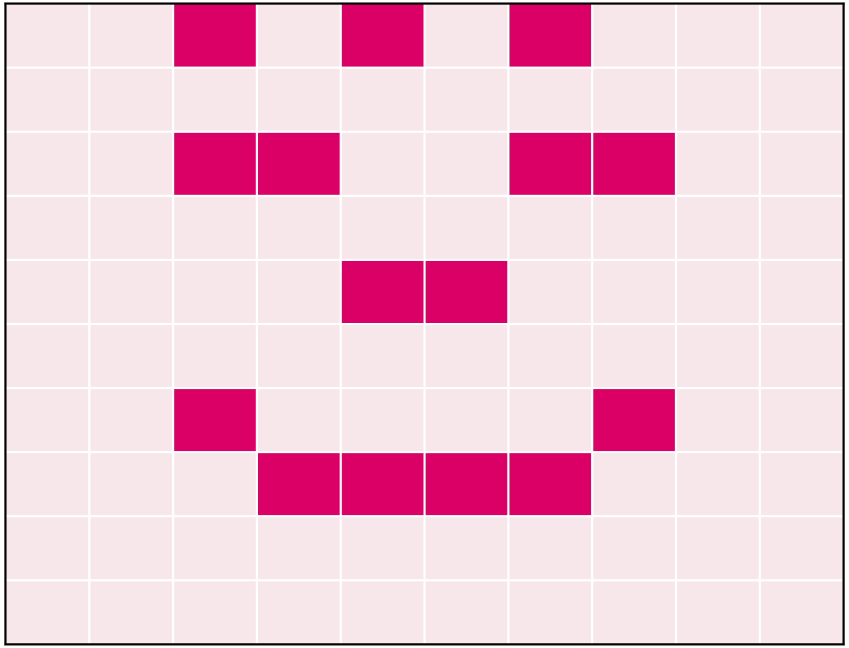


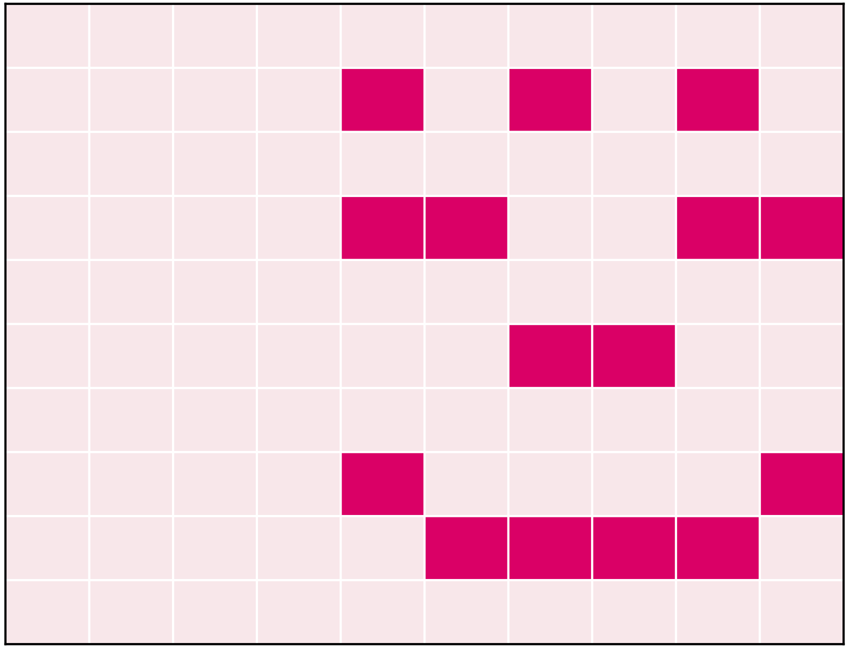
Classes of Instructions

Flynn’s Classifications - SIMD – Single Instruction Multiple Data

Example – Game Development

x+2, y-1





Classes of Instructions

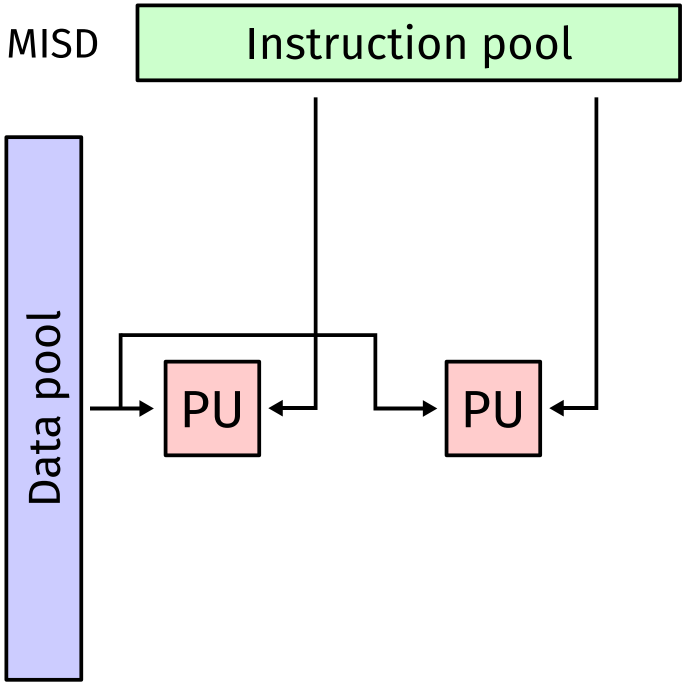
Flynn’s Classifications - MISD – Multiple Instructions Single Data

- Same data being processed differently at the same time
- Very specific, not widely used

Advantages – real-time fault detection

Disadvantages – Very limited application, not available commercially

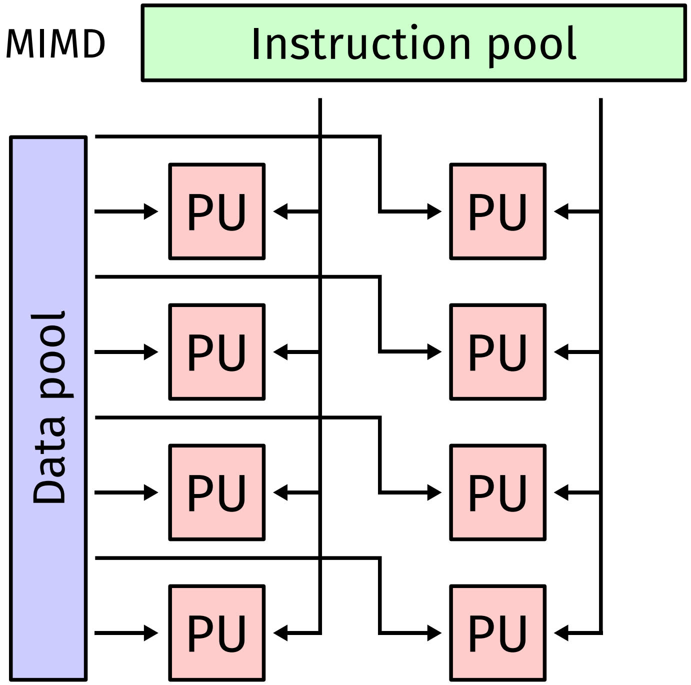
Examples – Space shuttle flight control system



Classes of Instructions

Flynn’s Classifications - MIMD – Multiple Instructions Multiple Data

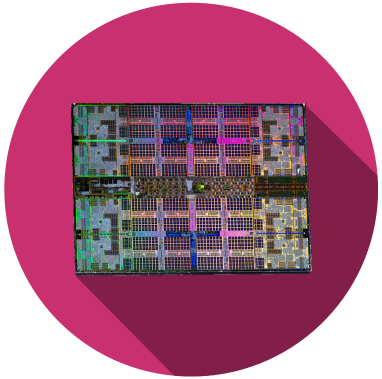
- Multiple processors performing operations on different pieces of data
- Different instructions can be executed at the same time using different datastreams



Advantages – Multitasking

Disadvantages – expensive and complicated architecture

Examples – Modern PC’s, Laptops, Smartphones



RISC vs. CISC

CISC vs. RISC

Comparison

CISC

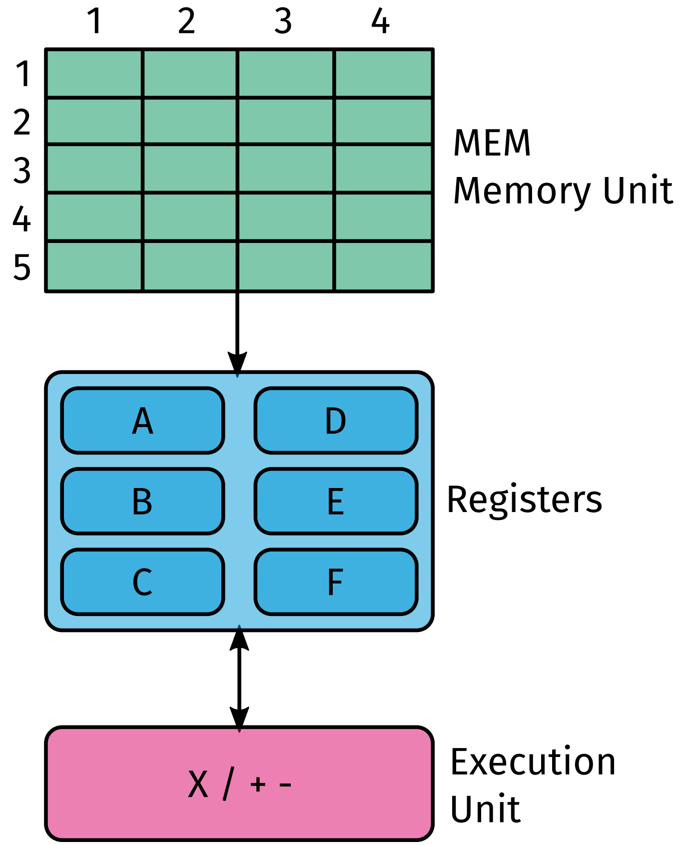
RISC

CISC vs. RISC

Example multiplication

CISC

```
MULT 2:3, 5:2
```

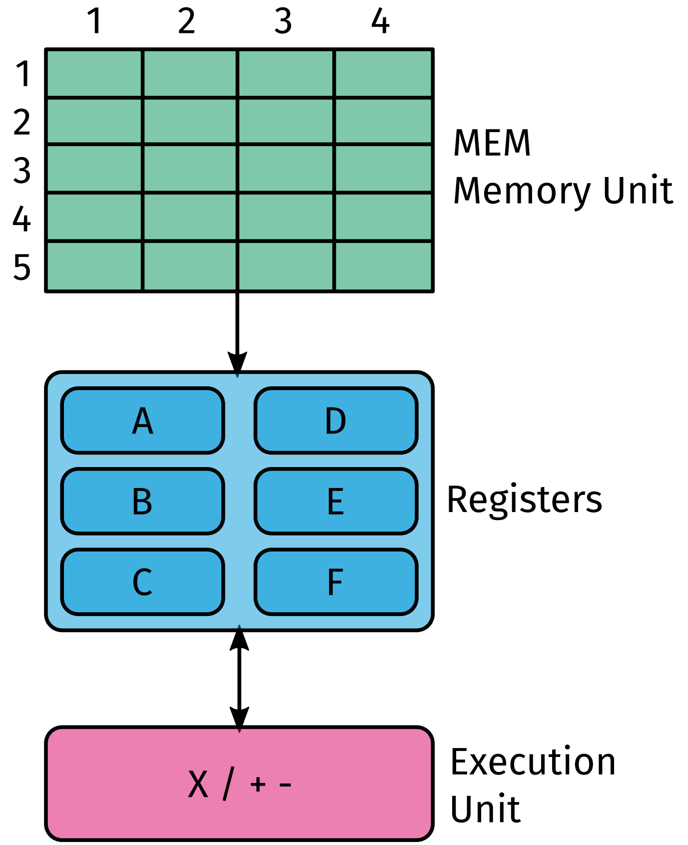


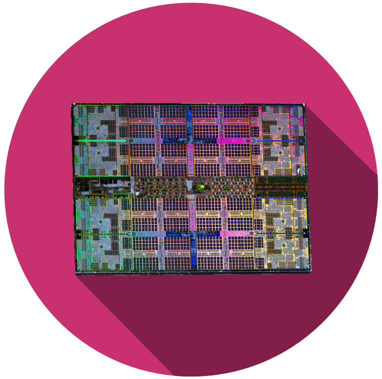
CISC vs. RISC

Example multiplication

RISC

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```





RISC-V ISA

RISC-V Architecture

- Developed by Krste Asanovic, David Patterson at UC Berkley 2010
- First widely accepted open-source computer architecture
- Gains a lot of attraction since 2019

RISC-V Architecture

Design Principles

1. Simplicity favors regularity
    - Consistent instruction format
    - Same number of operands (2 rs, 1 rd)
    - Easier to encode and handle in hardware
2. Make the common case fast
    - Only simple commonly used instructions
    - Decode hw sould be simple, small and fast
3. Smaller is faster
    - Only a small number of registers
4. Good design demands good compromises

RISC-V Instructions

Extentions

- ISA Namings are RVXXY
  - RV – RISC-V
  - XX – Bit number [32, 64, 128]
  - Y – one or more extentions

- Commonly used namings
  - RV32I – Basis for this course
  - RV64G – Linux compatible

Extention	Name
I	Integer base instructions
M	Math, Integer, multiplication and division instr.
A	Atomic instructions
F	Single-precision floating-point instructions
D	Double-precision floating-point instructions
G	General (I+M+A+F+D)
Q	Quad-precision floating-point instructions
L	Decimal floating-point instructions
C	Compressed instructions (16-bit)
B	Bit manipulation instructions
J	Dynamically translated languages
T	Tranactional memory instructions
P	Packed-SIMD instructions
V	Vector operations instructions
N	User-level interrupt instructions

RISC-V Instructions

Registers

RV32 has 32 32-bit registers

Register	ABI Name	Description	Saver	Preserved
x0	zero	Hard-wired zero	-	n/a
x1	ra	Return Address	Caller	No
x2	sp	Stack Pointer	Callee	Yes
x3	gp	Global Pointer	-	n/a
x4	tp	Thread Pointer	-	n/a
x5	t0	Temporary/alternate link register	Caller	No
x6-x7	t1-t2	Temporaries	Caller	No
x8	s0/fp	Saved register	Callee	Yes
x9	s1	Saved register	Callee	Yes
x10-x11	a0-a1	Function arguments/return values	Caller	No
x12-x17	a2-a7	Function arguments	Caller	No
x18-x27	s2-s11	Saved registers	Callee	Yes
x28-x31	t3-t6	Temporaries	Caller	No

RISC-V Instructions

Caller vs Callee

Caller: The calling function, which invokes another function (the callee) by issuing a function call.

Callee: The called function, which is invoked by another function (the caller) and executes a specific task.

Caller-Saved Registers:

Registers that a calling function (caller) must save if it wants to preserve their values across a function call. These registers typically include those holding temporary values or function parameters.

Callee-Saved Registers:

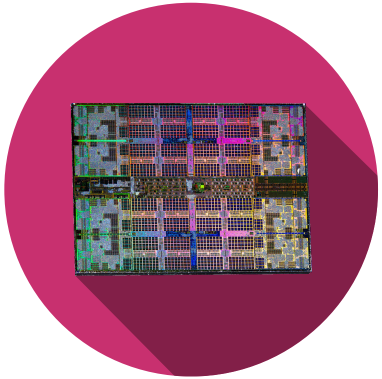
Registers that a called function (callee) must preserve if it modifies them, ensuring that their original values are restored before returning control to the caller. These registers usually include those holding critical information that the caller expects to remain unchanged.

RISC-V Instructions

Caller vs Callee

```
# Caller function: algo
# Arguments: a, b
# Return value: result
algo:
    addi sp, sp, -4     # Decrease stack pointer by 4 bytes
    sw   ra, 0(sp)      # Save return address (ra) on the stack

    # Load arguments into registers
    li   a0, 10         # Load argument 'a' (10) into register a0
    li   a1, 20         # Load argument 'b' (20) into register a1

    # Call the callee function
    jal  ra, calculate  # Jump and link to calculate function,
                        # return address stored in ra

    # Retrieve the result from the return value register
    mv   s0, a0         # Move return value from a0 to s0

    # Restore return address from the stack
    lw   ra, 0(sp)      # Load return address from the stack
    addi sp, sp, 4      # Increase stack pointer by 4 bytes

    # Return
    ret
```

```
# Callee function: calculate
# Arguments: x, y
# Return value: temp
calculate:
    # Allocate space on the stack for local variables
    addi sp, sp, -4     # Decrease stack pointer by 4 bytes
    sw   ra, 0(sp)      # Save return address (ra) on the stack

    # Perform calculation
    lw   t0, 0(a0)      # Load argument 'x' from register a0 into temp reg t0
    lw   t1, 0(a1)      # Load argument 'y' from register a1 into temp reg t1
    add  a0, t0, t1     # Add x and y, store result in a0

    # Restore return address from the stack
    lw   ra, 0(sp)      # Load return address from the stack
    addi sp, sp, 4      # Increase stack pointer by 4 bytes

    # Return
    ret
```



Instructions

RISC-V Instructions

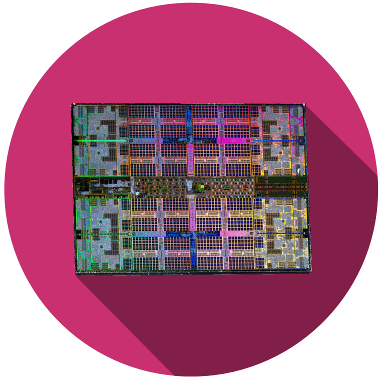
Simple and Complex instructions

C-Code | **RISC-V**

```
a = b + c
```

```
a = b + c - d
```

http://blog.translusion.com/images/posts/RISC-V-cheatsheet-RV32I-4-3.pdf
https://risc-v.guru/



Operands

RISC-V Instructions

Operands

Operands are physical locations in the computer

- Registers
- Memory
- Constants (also called immediates)

```
init: addi    s0,        s0,     4          ; s0 = s0 + 4
```

RISC-V Instructions

Revisit add instruction

C-Code

```
a = b + c
```

```
a = b + c - d
```

RISC-V

```
# x8=a, x9=b, x18=c
add x8, x9, x18 # a = b + c
```

```
# x8=b, x9=c, x18=d, x19=a
add x5,  x8, x9  # t = b + c
sub x19, x5, x18 # a = t - d
```

RISC-V Instructions

Rerevisit add instruction

C-Code

```c
a = b + c
```

```c
a = b + c - d
```

RISC-V

```
# s0=a, s1=b, s2=c
add s0, s1, s2 # a = b + c
```

```
# s0=b, s1=c, s2=d, s3=a
add t0, s0, s1 # t = b + c
sub s3, t0, s2 # a = t - d
```

RISC-V Instructions

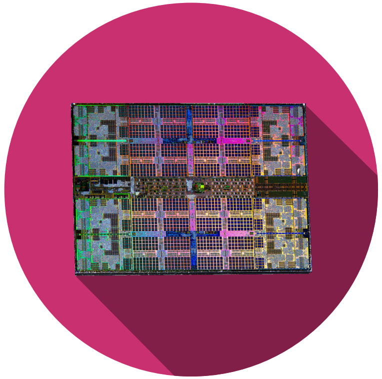
`addi` instruction

C-Code

```
a = b + 6
```

RISC-V

```
# s0 = a, s1 = b
```



Memory

Memory

- There is too much data to put in the 32 registers
- Store more data in memory
- Memory is large but slow
- Commonly used variables are kept in registers

Word Addressable Memory

Example of a Word addressable memory

Word Address	Data				Word Number
…					…
…					…
0x00000004	4 0	F 3	0 7	8 8	Word 4
0x00000003	0 1	E E	2 8	4 2	Word 3
0x00000002	F 2	F 1	A C	0 7	Word 2
0x00000001	8 9	A B	C D	E F	Word 1
0x00000000	0 1	2 3	4 5	6 7	Word 0

width = 4 bytes

RISC-V Instructions

`load` instructions

Load word: `lw`

Format:

```
lw destination, offset(base)
lw t1, 5(s0)
```

Address calculation

- Add base address (`s0`) to the offset (5)
- Address = `(s0+5)`

Result

- `t1` holds the data value at address `(s0+5)`

RISC-V Instructions

`load` instructions example

Read a word of data at memory address 2 into s3

- **Address:** `(0+2) = 2`
- `s3 = 0xF2F1AC07`

Assembly code:

```
lw s3, 0x8(zero)
```

Word Address	Data								Word Number
...									...
...									...
0×00000004	4	0	F	3	0	7	8	8	Word 4
0×00000003	0	1	E	E	2	8	4	2	Word 3
0×00000002	F	2	F	1	A	C	0	7	Word 2
0×00000001	8	9	A	B	C	D	E	F	Word 1
0×00000000	0	1	2	3	4	5	6	7	Word 0

width = 4 bytes

RISC-V Instructions

`store` instructions

Store word: `sw`

Format:

```
sw source, offset(base)
sw t4, 0x3(zero)
```

Address calculation

- Add base address (zero) to the offset (0x3)
- Address = `(0+3)`

Result

- Data at address `(0+3) holds the data from t4`

RISC-V Instructions

`store` instructions example

Write (store) a word in `t4` into memory address 3

- **Address:** `(0+0x3) = 3`
- `t4 = 0x01EE2842`

Word Address	Data	Word Number
...		...
...		...
0×00000004	4 0 F 3 0 7 8 8	Word 4
0×00000003	0 1 E E 2 8 4 2	Word 3
0×00000002	F 2 F 1 A C 0 7	Word 2
0×00000001	8 9 A B C D E F	Word 1
0×00000000	0 1 2 3 4 5 6 7	Word 0

width = 4 bytes

Assembly code:

```
sw t4, 12(zero)
```

Byte Addressable Memory

- RISC-V is byte-addressable memory
- Each databyte has a unique address
- Load/store word or single bytes
  - `lb` – load byte
  - `sb` – store byte
- 32-bit word = 4 bytes, word address increments by 4

Address	00	01	02	03
...				
...				
0×00000010	40	F3	07	88
0×0000000C	01	EE	28	42
0×00000008	F2	F1	AC	07
0×00000004	89	AB	CD	EF
0×00000000	01	23	45	67

width = 4bytes

Byte Addressable Memory

The address of a memory word must be multiplied by 4.

- Address of memory word 2 is 2*4=8
- Address of memory word 10 is 10*4=40 =0x28

RISC-V is byte-addressed

Address	00	01	02	03
…				
…				
0×00000010	40	F3	07	88
0×0000000C	01	EE	28	42
0×00000008	F2	F1	AC	07
0×00000004	89	AB	CD	EF
0×00000000	01	23	45	67

width = 4bytes

RISC-V Instructions

load instructions example

Load a word of data at memory address 12 into s3

Assembly code:

```
lb s3, 0xC(zero)
s3 = 0x00000001


lb s3, 0xD(zero)
s3 = 0x000000ee
```

Address	00	01	02	03
...				
...				
0×00000010	40	F3	07	88
0×0000000C	01	EE	28	42
0×00000008	F2	F1	AC	07
0×00000004	89	AB	CD	EF
0×00000000	01	23	45	67

width = 4bytes

RISC-V Instructions

store instructions example

Store the value held in t7 into memory address 0x10 (16)

- t7 = 0x00000088

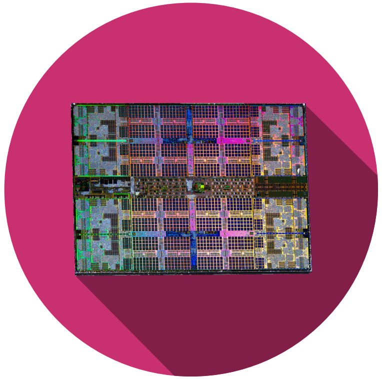
Assembly code:

```
sb t7, 0x10(zero)
```

0x00000010 = 88F30788

Address	00	01	02	03
…				
…				
0x00000010	40	F3	07	88
0x0000000C	01	EE	28	42
0x00000008	F2	F1	AC	07
0x00000004	89	AB	CD	EF
0x00000000	01	23	45	67

width = 4bytes



Constants

RISC-V Instructions

Constants 12-bit using addi

C-Code

```c
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

RISC-V

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

Any immediate that needs more than 12bits cannot use this method

RISC-V Instructions

Constants 12-bit using `addi`

C-Code

```c
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

RISC-V

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

Any immediate that needs more than 12bits cannot use this method

[-2048 - + 2047]

RISC-V Instructions

Constants 32-bit using `lui` and `addi`

`lui`: puts and immediate in the upper 20 bits of destination register and 0’s in lower 12 bits. `addi` adds the remaining 12 bits

C-Code

```c
int a = 0xFEDC8765;
```

0b1111 1110 1101 1100 1000 0111 0110 0101

RISC-V

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

Remember that `addi` sign-extends its 12-bit immediate

RISC-V Instructions

Constants 32-bit using `lui` and `addi`

If bit 11 of 32bit constant is 1, increment upper 20 bits by 1 in lui

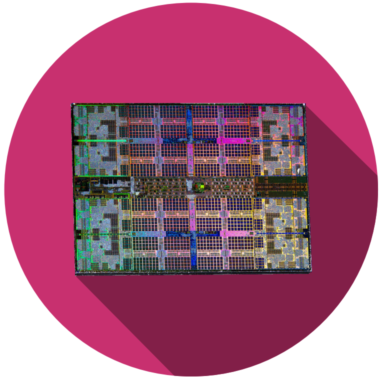
C-Code

Note: `-341 = 0xEAB`

```
int a = 0xFEDC8EAB;
0b1111 1110 1101 1100 1000 1110 1010 1011
```

RISC-V

```
# s0 = a
lui  s0, 0xFEDC9      # s0 = 0xFEDC9000
addi s0, s0, -341     # s0 = 0xFEDC9000 + 0xFFFFFEAB = 0xFEDC8EAB
```



Logical Instructions

RISC-V Instructions

Logical instructions `and`, `or` and `xor`

`and`: useful for masking bits

- E.g. masking all but the least significant byte of a value

  `0xF234012F and 0x000000FF = 0x0000002F`

`or`: useful for combining bit fields

- E.g. combing two values:

  `0xF2340000 or 0x000012BC = 0xF23412BC`

`xor`: useful for inverting bits

- E.g remember that `-1 = 0xFFFFFFFF`

  `A xor -1 = not A`

RISC-V Instructions

Example 1: Logical instructions and, or and xor

Source Registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

RISC-V Code

```
and s3, s1, s2
or  s4, s1, s2
xor s5, s1, s2
```

Results

s3				
s4				
s5				

RISC-V Instructions

Example 2: Logical instructions and, or and xor

Source Registers

t3	0011	1010	0111	0101	0000	1101	0110	1111
imm	1111	1111	1111	1111	1111	1010	0011	0100

sign extended

RISC-V Code

```
andi s5, t3, -1484
ori  s6, t3, -1484
xori s7, t3, -1484
```

Results

s5								
s6								
s7								

RISC-V Instructions

Shift instructions `sll`, `srl` and `sra`

Shift amount is in (lower 5 bits of) a register (0 to 31)

`sll`: shift left logical

- E.g. `sll t0, t1, t2 # t0 = t1 << t2`

`srl`: shift right logical

- E.g. `srl t0, t1, t2 # t0 = t1 >> t2`
- Fills upper bits with zeros

`sra`: shift right arithmetic

- E.g. `sra t0, t1, t2 # t0 = t1 >>> t2`
- Fills upper bits with MSb

RISC-V Instructions

Shift instructions `sll`, `srl` and `sra`

Shift amount is an immediate between 0 to 31 (5 bits)

`slli`: shift left logical immediate

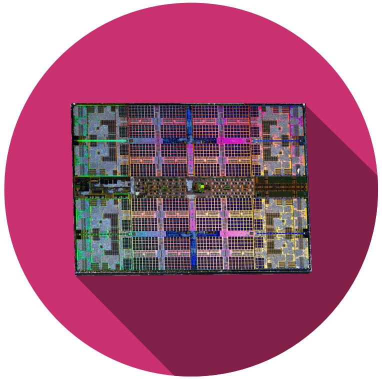
- E.g. `slli t0, t1, 23 # t0 = t1 << 23`

`srli`: shift right logical immediate

- E.g. `srli t0, t1, 18 # t0 = t1 >> 18`
- Fills upper bits with zeros

`srai`: shift right arithmetic immediate

- E.g. `srai t0, t1, 5 # t0 = t1 >>> 5`
- Fills upper bits with MSb



Multiplication and Division Instructions

RISC-V Instructions

Multiplication instructions `mul`, `mulh`

32x32 multiplication => 64 bit result

```
mul  s0, s1, s2
```

`s0` = lower 32 bit of result

```
mulh s0, s1, s2
```

`s0` = upper 32 bit of result, treats operands as signed

For full 64-bit result

```
mul  s3, s1, s2
mulh s4, s1, s2
```

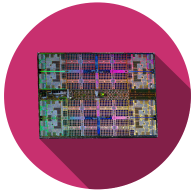
Example:

s1 = 0x40000000 = 2^{30}; s2 = 0x80000000 = -2^{31}

s1 x s2 =

s4 = ; s3 =

RISC-V Instructions

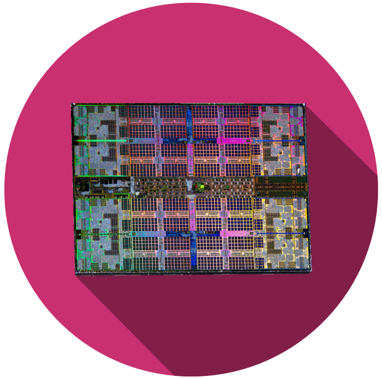
Division instructions `div`, `rem`

32-division => 32-bit quotient &remainder

```
div s3, s1, s2 # s3 = s1/s2
rem s4, s1, s2 # s4 = s1%s2
```

Example:

```
s1 = 0x00000011 = 17; s2 = 0x00000003 = 3
s1 / s2 =
s1 % s2 =
s3 =             ; s4 =
```



Branch Instructions

RISC-V Instructions

Branch instructions `beq`, `bne`, `blt`, `bge` , `j`, `jr`, `jal`, `jalr`

Conditional

- `beq` = Branch if equal
- `bne` = Branch if not equal
- `blt` = Branch if less than
- `bge` = Branch if greater than

Unconditional

- `j` = jump (pseudo instruction)
- `jr` = jump register (pseudo instruction)
- `jal` = jump and link
- `jalr` = jump and link register

RISC-V Instructions

Conditional Branch instructions beq

RISC-V Code

```
  addi s0, zero, 4      # s0 = 0 + 4 = 4
  addi s1, zero, 1      # s1 = 0 + 1 = 1
  slli s1, s1, 2        # s1 = 1 << 2 = 4
  beq  s0, s1, target   # branch is taken
  addi s1, s1, 1        # not executed
  sub  s1, s1, s0       # not executed

target:                 # label
  add  s1, s1, s0       # s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved word and must be followed by a colon :

RISC-V Instructions

Conditional Branch instructions bne

RISC-V Code

```
  addi s0, zero, 4      # s0 = 0 + 4 = 4
  addi s1, zero, 1      # s1 = 0 + 1 = 1
  slli s1, s1, 2        # s1 = 1 << 2 = 4
  bne  s0, s1, target   # branch not taken
  addi s1, s1, 1        # s1 = 4 + 1 = 5
  sub  s1, s1, s0       # s1 = 5 - 4 = 1

target:                 # label
  add  s1, s1, s0       # s1 = 1 + 4 = 5
```

RISC-V Instructions

Unconditional Branch instructions j

RISC-V Code

```
  j    target         # jump to target
  srai s1, s1, 2      # not executed
  addi s1, s1, 1      # not executed
  sub  s1, s1, s0     # not executed

target:
  add  s1, s1, s0     # s1 = 1 + 4 = 5
```

RISC-V Instructions

Conditional Statements and Loops

Conditional Statement

- `if` statements
- `if` / `else` statements

Loops

- `while` loops
- `for` loops

RISC-V Instructions

if Statement

C Code

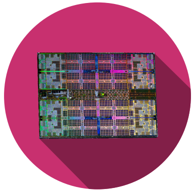
```
if (i == j)
  f = g + h;
f = f - i;
```

RISC-V Assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
```

RISC-V Instructions

if/else Statement



C Code

```c
if (i == j)
  f = g + h;

else

  f = f - i;
```

RISC-V Assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
```

RISC-V Instructions

`loops` Statement

```
for (initialization; condition; loop operation)
   statement
```

- `initialization`: executes before the loop begins
- `condition`: is tested at the beginning of each iteration
- `loop operation`: executes at the end of each iteration
- `statement`: executes each time the condition is met

RISC-V Instructions

for Statement

C Code

```c
// add the numbers from 0 to 9
int sum = 0;
int i;

for(i=0; i!=10; i=i+1){
  sum = sum + i;
}
```

RISC-V Assembly code

```
# s0 = i, s1 = sum
```

RISC-V Instructions

for with less then comparison

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for(i=1;i<101; i=i*2){
  sum = sum + i;
}
```

RISC-V Assembly code

```
# s0 = i, s1 = sum
```

RISC-V Instructions

for with less then comparison v2

C Code

```c
// add the powers of 2 from 1
// to 100

int sum = 0;
int i;

for(i=1;i<101; i=i*2){
  sum = sum + i;
}
```

RISC-V Assembly code

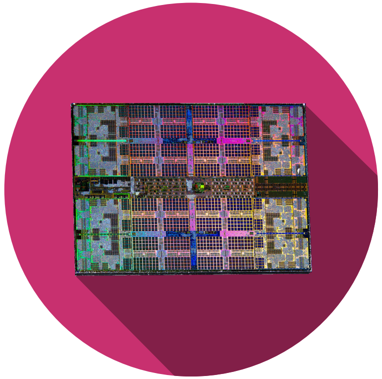
```
# s0 = sum, s1 = i
  addi s0, zero, 0   # sum = 0
  addi s1, zero, 1   # i = 1
  addi t0, zero, 101 # 101
for:
  slt  t2, s1, t0    # i < 101

  beq  t2, zero, done
  add  s0, s0, s1    # sum += i
  slli s1, s1, 1     # i *= 2
  j    for
done:
```

RISC-V Instructions

Compare instructions `slt`

`slt`: set if less than instruction

```
slt t2, s0, t0 # if s0 < t0, t2 = 1
               # otherwise    t2 = 0
```



Arrays

RISC-V Instructions

Arrays

Access large amount of similar data

- Index: access each element
- Size: number of elements

RISC-V Instructions

Arrays

- 5-element array
- Base address = `0x123B4780` = address of first element `array[0]`
- First step is load the base address into a register

Address	Data
0x123B4790	array[4]
0x123B478C	array[3]
0x123B4788	array[2]
0x123B4784	array[1]
0x123B4780	array[0]

Main Memory

RISC-V Instructions

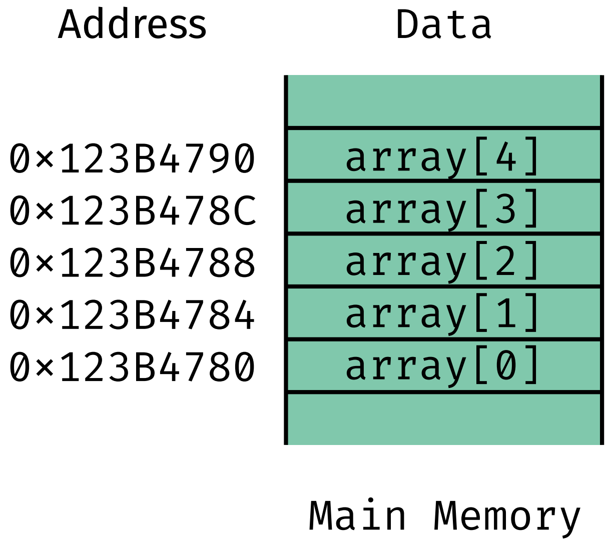
Accessing Arrays

C-Code

```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

RISC-V Assembly

```
# s0 = array base address
lui  s0, 0x123B4    # load upper base address
addi s0, s0, 0x180 # load lower base address
lw   t0, 0(s0)      # array[0]
slli t0, t0, 1      # x2
sw   t0, 0(s0)      # store array back
lw   t0, 4(s0)      # array[1]
slli t0, t0, 1      # x2
sw   t0, 4(s0)      # store array back
```



RISC-V Instructions

Accessing Arrays using for loops

C-Code

```c
int array[1000];
int i;

for (i=0; i<100; i=i+1)
  array[i] = array[i] * 8;
```

RISC-V Assembly

```
# s0, array base address, s1 = i
# initialize code
  lui  s0, 0x23B8F     # s0 = 0x23B8F000
  ori  s0, s0, 0x400   # s0 = 0x23B8F400
  addi s1, zero, 0     # i = 0
  addi t2, zero, 1000  # t2 = 1000

loop:
  bge  s1, t2, done    # if not then done
  slli t0, s1, 2       # t0 = i * 4 (byte offset)
  add  t0, t0, s0      # address of array[i]
  lw   t1, 0(t0)       # t1 = array[i]
  slli t1, t1, 3       # t1 = array[i] * 8
  sw   t1, 0(t0)       # array[i] = array[i] * 8
  addi s1, s1, 1       # i = i + 1
  j    loop            # repeat
done:
```

RISC-V Instructions

Remainder ASCII-Table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

RISC-V Instructions

Accessing Arrays of Characters

C-Code

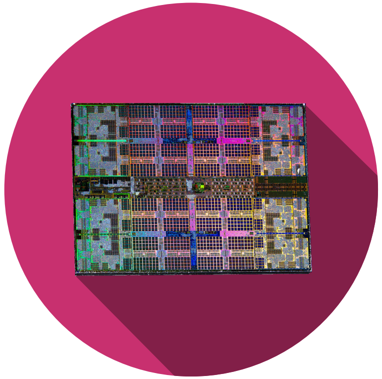
```c
char str[80] = "CAT";
int len = 0;

// compute length of
string
while (str[len]) len++;
```

Memory

```
1004 0x00
1003 0x54
1002 0x41
1000 0x43
```

RISC-V Assembly

```
# s0 = array base address, s1 = len
  addi s1, zero, 0     # len = 0
while:
  add  t0, s0, s1      # t0 = address of str[len]
  lb   t1, 0(t0)       # t1 = str[len]
  beq  t1, zero, done # non zero?
  addi s1, s1, 1       # len++
  j    while
done:
```



Function Calls

RISC-V Instructions

Function Calls

- Caller: calling function (in this case `main`)
- Callee: called function (in this case `sum`)

C Code

```
void main(){
  int y;
  y = sum(42,7);
  ...
}

int sum(int a, int b)
  return (a + b);
```

RISC-V Instructions

Simple Function Call

C-Code

```
int main(){
simple();
  a= a + b;
}

void simple(){
  return;
}
```

RISC-V Assembly

```
0x00000300 main: jal simple # call, ra=0x00000304
0x00000304 add s0, s1, s2
...


0x0000051c simple: jr ra     # return
```

void means that `simple` does not return a value

```
jal simple:
    ra = PC+4 (0x00000304)
    jumps to simple label (PC = 0x0000051c)
jr ra:
    PC = ra(0x00000304)
```

RISC-V Instructions

Function Call Convention

- Caller:
  - Passes **arguments** to a callee
  - **Jumps** to callee
- Callee:
  - **Performs** the function
  - **Returns** result to caller
  - **Returns** to point of call
  - Must **not overwrite** registers or memory needed by caller

RISC-V Instructions

Input arguments and return value

RISC-V Conventions

- Argument values `a0–a7`
- Return value: `a0`

RISC-V Instructions

Input arguments and return value Example

C Code

```c
int main(){
   int y;
   ...
   y = diffofsums(2, 3, 4, 5); // 4 arguments
   ...
}

int diffofsums(int f, int g, int h, int i){
   int result;
   result = (f + g) - (h + i);
   return result; // return value
}
```

RISC-V Instructions

Input arguments and return value: Example

RISC-V Assembly Code

```
# s7 = y
main:
```

```
# s3 = result
diffofsums:
```

RISC-V Instructions

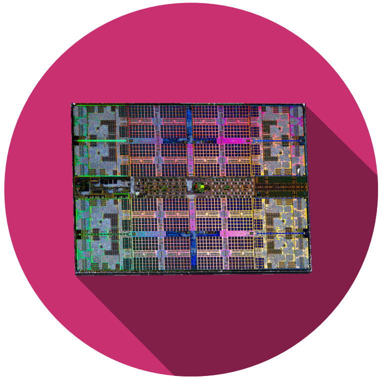
Input arguments and return value: Example

RISC-V Assembly Code

```
diffofsums:
   add t0, a0, a1     # t0 = f + g
   add t1, a2, a3     # t1 = h + i
   sub s3, t0, t1     # result = (f + g) - (h + i)
```

`jal diffofsums` is pseudocode for `jal ra diffofsums`

- Diffofsums overwrote 3 registers: t0, t1, s3

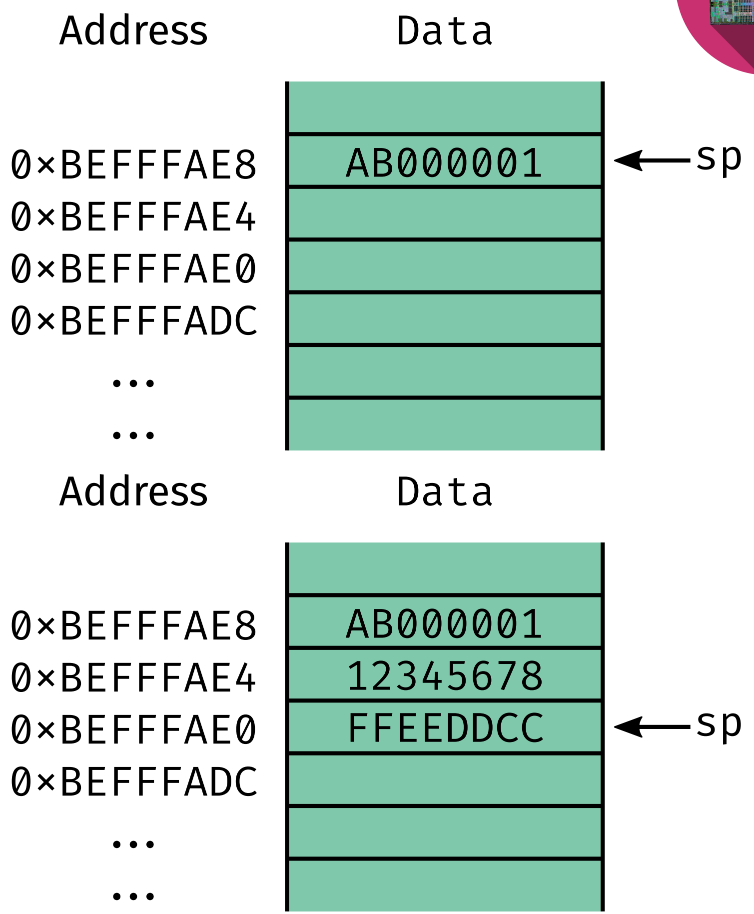


Stack

RISC-V Instructions

The Stack

- Memory used to temporarily save variables
- Last-in-frist-out (LIFO) queue
- Expands: uses more memory when more space is needed
- Contracts: uses less memory when the space in no longer required
- Grows down (high to lower memory addresses)
- Stap pointer: sp points to the top of the stack



RISC-V Instructions

How Functions use the Stack

- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 register: `t0, t1, s3`

RISC-V assembly code

```
# s3 = result
diffofsums:
  add t0, a0, a1 # t0 = f + g
  add t1, a2, a3 # t1 = h + i
  sub s3, t0, t1 # result = (f + g) - (h + i)
  add a0, s3, zero # put return value in a0
  jr  ra             # return to caller
```

RISC-V Instructions

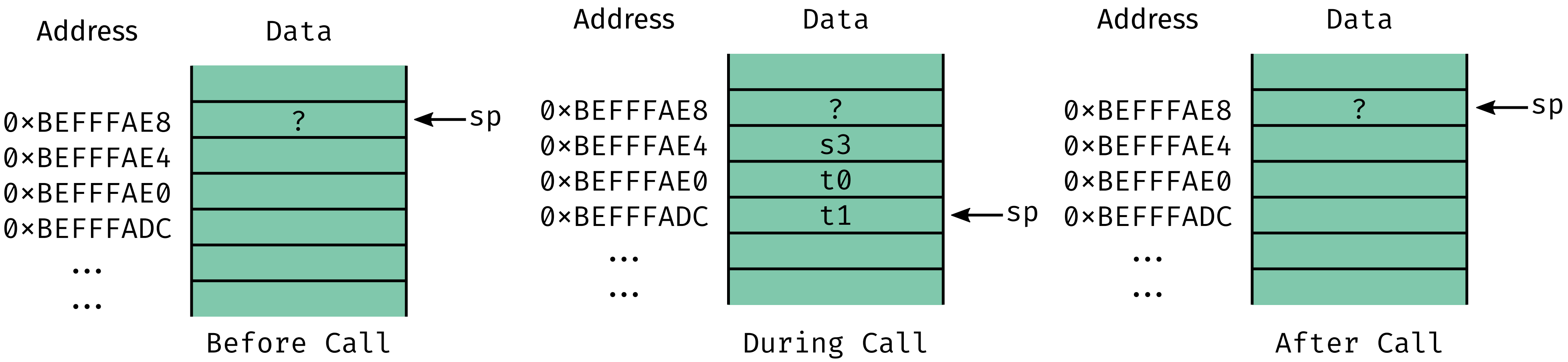
Storing register values in stack

RISC-V assembly code

```
# s3 = result
diffofsums:
  addi sp, sp, -12  # make space on stack to
                    # store three registers
  sw   s3, 8(sp)    # save s3 on stack
  sw   t0, 4(sp)    # save t0 on stack
  sw   t1, 0(sp)    # save t1 on stack
  add  t0, a0, a1   # t0 = f + g
  add  t1, a2, a3   # t1 = h + i
  sub  s3, t0, t1   # result = (f + g) - (h + i)
  add  a0, s3, zero # put return value in a0
  lw   s3, 8(sp)    # restore s3 from stack
  lw   t0, 4(sp)    # restore t0 from stack
  lw   t1, 0(sp)    # restore t1 from stack
  addi sp, sp, 4    # deallocate stack space
  jalr zero, ra, 0  # return to caller
```

RISC-V Instructions

The Stack during `diffofsums` Call



RISC-V Instructions

Preserved Registers

Preserved Callee-Saved	Nonpreserved Caller- Saved
`s0-s11`	`t0-t6`
	`a0-a7`
`sp`	`ra`
`stack above sp`	`stack below sp`

RISC-V Instructions

Registers

Register	ABI Name	Description	Saver	Preserved
x0	zero	Hard-wired zero	-	n/a
x1	ra	Return Address	Caller	No
x2	sp	Stack Pointer	Callee	Yes
x3	gp	Global Pointer	-	n/a
x4	tp	Thread Pointer	-	n/a
x5	t0	Temporary/alternate link register	Caller	No
x6-x7	t1-t2	Temporaries	Caller	No
x8	s0/fp	Saved register	Callee	Yes
x9	s1	Saved register	Callee	Yes
x10-x11	a0-a1	Function arguments/return values	Caller	No
x12-x17	a2-a7	Function arguments	Caller	No
x18-x27	s2-s11	Saved registers	Callee	Yes
x28-x31	t3-t6	Temporaries	Caller	No

RISC-V Instructions

Storing Saved Registers in the Stack

RISC-V assembly code

```
# s3 = result
diffofsums:
  addi sp, sp, -4    # make space on stack to
                     # store one registers
  sw   s3, 0(sp)     # save s3 on stack
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  s3, t0, t1    # result = (f + g) - (h + i)
  add  a0, s3, zero  # put return value in a0
  lw   s3, 0(sp)     # restore s3 from stack
  addi sp, sp, 4     # deallocate stack space
  jr   ra            # return to caller
```

RISC-V Instructions

Optimized diffofsums

RISC-V assembly code

```
# a0 = result
diffofsums:
  add  t0, a0, a1   # t0 = f + g
  add  t1, a2, a3   # t1 = h + i
  sub  a0, t0, t1   # result = (f + g) - (h + i)
  jr   ra           # return to caller
```

RISC-V Instructions

Nested function aka Non-leaf Function Calls

Non-leaf function: a function that calls another function

RISC-V assembly code

```
f1:
  addi sp, sp, -4 # make space on stack
  sw   ra, 0(sp)  # save ra on stack
  jal  f2
  ...
  lw   ra, 0(sp)  # restore ra from stack
  addi sp, sp, 4  # deallocate stack space
  jr   ra         # return caller
```

ra must be presewed before a function call

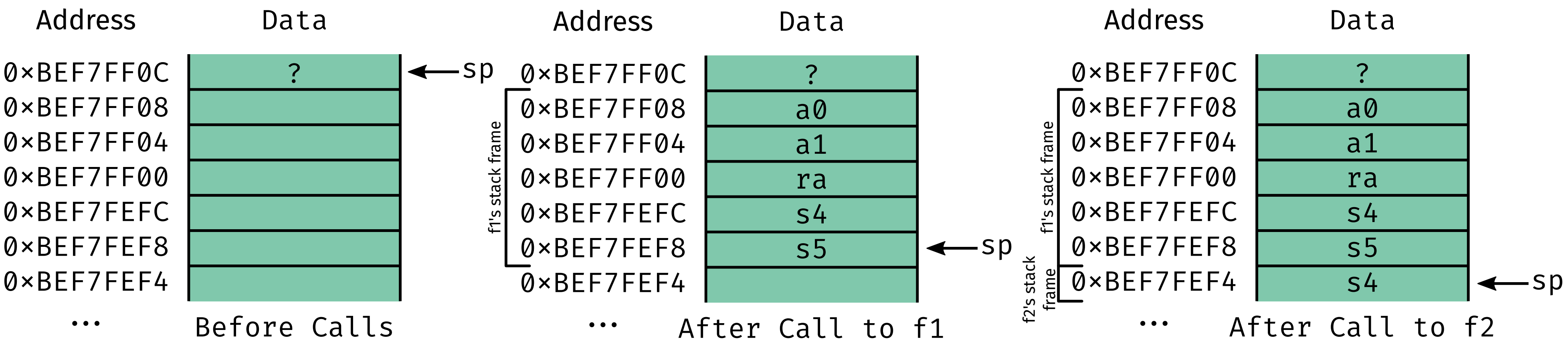
RISC-V Instructions

Non-leaf Function Calls: Example

```
f1:
  addi sp, sp ,-20 # make space on stack for 5 words
  sw a0, 16(sp)
  sw a1, 12(sp)
  sw ra, 8(sp)     # save ra on stack
  sw s4, 4(sp)
  sw s5, 0(sp)
  jal f2
  ...
  lw ra, 8(sp)     # restore ra (and other regs) from stack
  ...
  addi sp, sp, 20  # deallocate stack space
  jr ra

# f2 (leaf-function) only uses s4 and calls no functions
f2:
  addi sp, sp, -4  # make space on stack for 1 word
  sw   s4, 0(sp)
  ...
  lw   s4, 0 (sp)
  addi sp, sp, 4   # deallocate stack space
  jr   ra          # return to caller
```

RISC-V Instructions

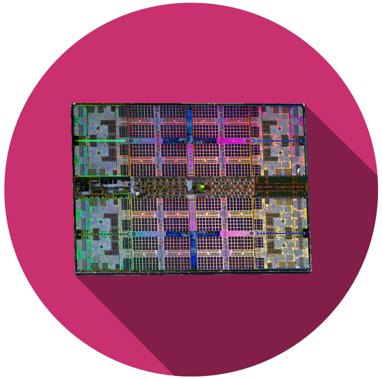
Non-leaf Function Calls: Example



RISC-V Instructions

Function Call Summary

- Caller
  - Put arguments in `a0-a7`
  - Save any needed registers (`ra,t0-t6/a0-a7`)
  - Call function: jal callee
  - Restore any saved registers
  - Look for result in `a0`
- Callee
  - Save registers that might be distrubed (`s0-s11`)
  - Perform function
  - Put result in `a0`
  - Restore registers
  - Return: `jr ra`



Jumps and Pseudoinstructions

RISC-V Instructions

Jumps

- RISC-V has two types of unconditional jumps
  - Jump and link (`jal rd, label`) → (`jal rd,` $\texttt{imm}_{20:0}$)
    - `rd = PC+4; PC =  PC + imm`
  - Jump and link register (`jalr rd, rs,` $\texttt{imm}_{11:0}$)
    - `rd = PC+4; PC = rs + SignExt(imm)`

- Linking is the storage of the next executable return address `rd`

RISC-V Instructions

Pseudoinstructions

- Pseudoinstructions are not RISC-V instructions but often more convenient for the programmer
- Assembler converts them into RISC-V instructions

RISC-V Instructions

Jump Pseudoinstructions

- Four jump pseudoinstructions exist
  - `j    imm => jal x0, imm       # return address ignored`
  - `jal  imm => jal ra, imm       # return address in ra`
  - `jr   rs  => jalr x0, rs, 0    # return address ignored`
  - `                              # jump address rs+0`
  - `ret      => jr  ra           => jalr x0, ra, 0)`

RISC-V Instructions

Labels

- Label indicated where to jump to
- Represented in jump as immediate offset
  - imm = # bytes past jump instruction
  - In example, below, imm = (0x51C-0x300) = 0x21C

RISC-V assembly code

```
0x00000300 main:   jal simple      # call
0x00000304         add s0, s1, s1
...                ...
0x0000051c simple: jr ra           # return
```

RISC-V Instructions

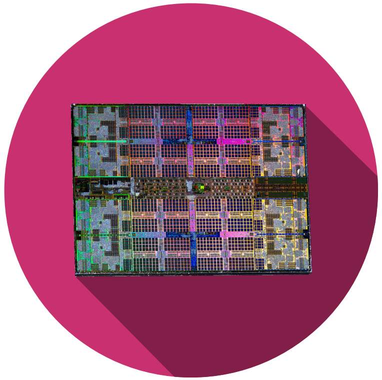
Long Jumps

- The immediate is limited in size
  - 20bits for `jal`, 12bits for `jalr`
  - Limits how far a program can jump
- Special instruction to help jumping further
  - `auipc rd, imm # add upper immediate to PC`
    - $\mathtt{rd = PC + \{imm_{31:12}, 12'b0\}}$
- Pseudoinstruction: `call` $\mathtt{imm_{31:0}}$
  - Behaves like `jal imm`, but allows 32-bit immediate offset
    - `ra` is used as a temporary register
    - `auipc, ra,` $\mathtt{imm_{31:12}}$
    - `jalr ra, ra,` $\mathtt{imm_{11:0}}$

RISC-V Instructions

Common Pseudoinstructions

Pseudoinstruction	RISC-V Instruction
`j label`	`jal zero, label`
`jr ra`	`jalr zero, ra, 0`
`mv t5, s3`	`addi t5, s3, 0`
`not s7, t2`	`xori s7, t2, -1`
`nop`	`addi zero, zero, 0`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L7`	`bge t2, zero, L7`
`call L1`	`auipc ra, imm31:12` `jalr ra, ra, imm11:0`
`ret`	`jalr zero, ra, 0`



Machine Language

RISC-V Instructions

Machine Language

- Binary representation of instructions
- Computers only understand 0’s and 1’s
- 32-bit instructions
- 5 Types of instructions
  - R4-Type (**4 R**egisters)
  - R-Type (**R**egister)
  - I-Type (**I**mmediate)
  - S/B Type (**S**tore & **B**ranches)
  - U/J Type (**U**pper Immediate & **J**ump)

RISC-V Instructions

Instruction Types

31 … 27	26 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
funct7		rs2	rs1	funct3	rd	opcode	R-Type
$imm_{11:0}$			rs1	funct3	rd	opcode	I-Type
$imm_{11:5}$		rs2	rs1	funct3	$imm_{4:0}$	opcode	S-Type
$imm_{12\|10:5}$		rs2	rs1	funct3	$imm_{4:1\|11}$	opcode	B-Type
$imm_{31:12}$					rd	opcode	U-Type
$imm_{20\|10:1\|11\|19:12}$					rd	opcode	J-Type
fs3	funct2	fs2	fs1	funct3	fd	opcode	R4-Type

- Immediate bits mostly occupy **consistent instruction bits**
  - Simplifies hardware to build the microprocessor
- **Sign bit** of signed immediate is in **msb** of instruction

RISC-V Instructions

Immediate Encodings

31	30 … 20	19 … 12	11	10 … 5	4 … 1	0	
$-inst_{31}-$				$inst_{30:25}$	$inst_{24:21}$	$inst_{20}$	I-Type
$-inst_{31}-$				$inst_{30:25}$	$inst_{11:8}$	$inst_{7}$	S-Type
$-inst_{31}-$			$inst_{7}$	$inst_{30:25}$	$inst_{11:8}$	0	B-Type
$inst_{31}$	$inst_{30:20}$	$inst_{19:12}$	$-0-$				U-Type
$-inst_{31}-$		$inst_{19:12}$	$inst_{20}$	$inst_{30:25}$	$inst_{24:21}$	0	J-Type

31 … 21	20 … 12	11 … 1	0	
$-imm_{11}-$		$imm_{11:1}$	imm_{0}	I&S-Type
$-imm_{12}-$		$imm_{11:1}$	0	B-Type
$imm_{31:21}$	$imm_{20:12}$	0		U-Type
$-imm_{20}-$	$imm_{20:12}$	$imm_{11:1}$	0	J-Type

RISC-V Instructions

Example: R-Type

Assembly

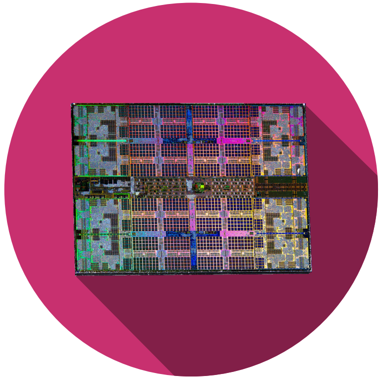
Encoding

add s2, s3 ,s4

funct7	rs2	rs1	funct3	rd	op	
						Field Values
						Machine Code
7bits	5bits	5bits	3bits	5bits	7bits	

sub t0, t1, t2

funct7	rs2	rs1	funct3	rd	op	
						Field Values
						Machine Code
7bits	5bits	5bits	3bits	5bits	7bits	



Decode Machine Language

RISC-V Instructions

Decode Machine Language

- See your own summary for a decoding reference table
- Start with op & `funct3`: it tells how to parse the rest
- Extract fields
- op, `funct3` and `funct7` fields tell the operation

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address][7:0])
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address][15:0])
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address][31:0]
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address][7:0])
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address][15:0])
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical imm.	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than imm.	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less then imm. unsig.	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical imm.	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm

RISC-V Instructions

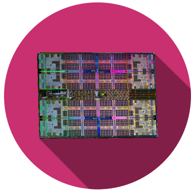
Instruction Types

31 … 27	26 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
funct7		rs2	rs1	funct3	rd	opcode	R-Type
$imm_{11:0}$			rs1	funct3	rd	opcode	I-Type
$imm_{11:5}$		rs2	rs1	funct3	$imm_{4:0}$	opcode	S-Type
$imm_{12\|10:5}$		rs2	rs1	funct3	$imm_{4:1\|11}$	opcode	B-Type
$imm_{31:12}$					rd	opcode	U-Type
$imm_{20\|10:1\|11\|19:12}$					rd	opcode	J-Type
fs3	funct2	fs2	fs1	funct3	fd	opcode	R4-Type

- Immediate bits mostly occupy **consistent instruction bits**
  - Simplifies hardware to build the microprocessor
- **Sign bit** of signed immediate is in **msb** of instruction

RISC-V Instructions

Decode Machine Language



```
0x41FE83B3: 0100 0001 1111 1110 1000 0011 1011 0011
```

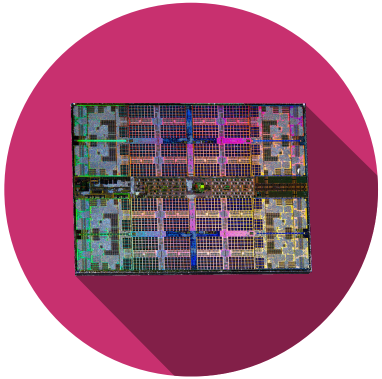
```
0xFDA48393: 1111 1101 1010 0100 1000 0011 1001 0011
```

RISC-V Instructions

Decode Machine Language

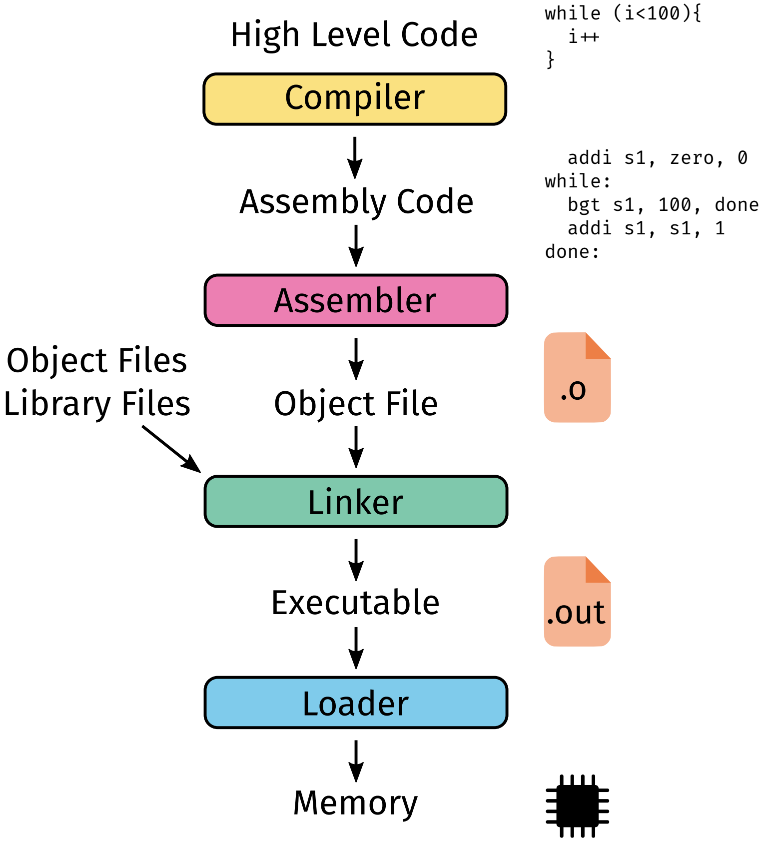
0x41FE83B3: 0100 0001 1111 1110 1000 0011 1011 0011

0xFDA48393: 1111 1101 1010 0100 1000 0011 1001 0011



Compile and Run

Compile and Run a Program



Compile a program

```
$ gcc -g -O -c main.c
```

- Instructions are also called text
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program

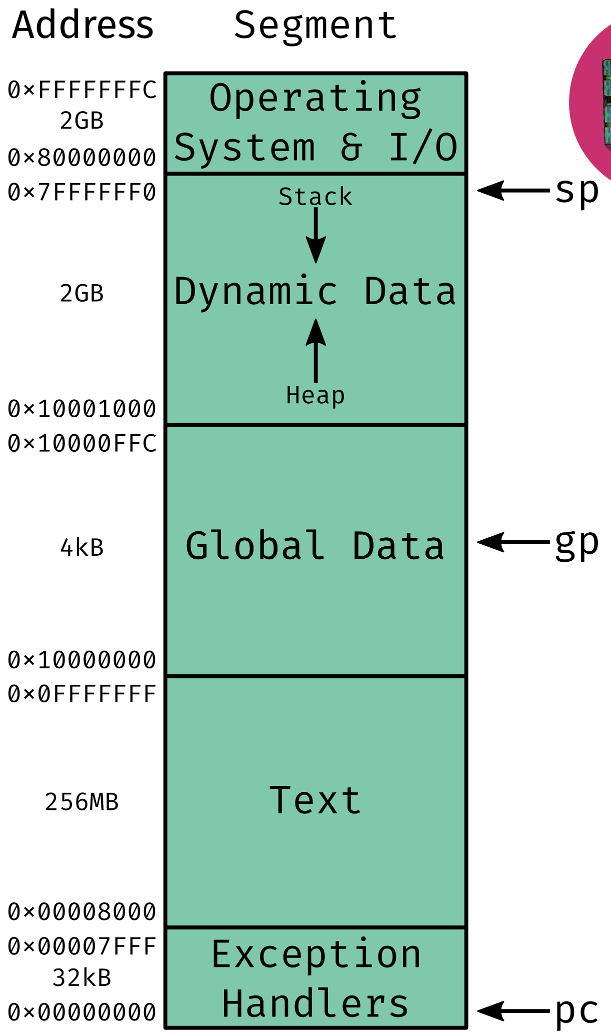
Size of memory = 2^{32} = 4GB

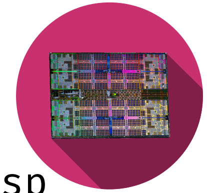
- 0x00000000 – 0xFFFFFFFF

RISC-V

Memory Map: Example

- Not necessarly defined by the processor but by the OS
- Exception handler
- First bytes instruction to load a boot flash into memory and jump to the instructions
- Text: hold the user program
- Global Data hold global variable and can be small
- Dynamic data: heap from the bottom and stack from the top
  - If they collide mem alloc error





Memory

Stack vs Heap

```
fn main
```

```
let s1 = String::from("hello"); // "hello" is stored on the heap,
with a pointer for s1 on the stack
let s2 = s1; // "s2 pointer pointing to the same "hello" is created
on the stack, while s1 pointer is invalidated
println!("s1 = {}, s2 = {}", s1, s2); // WE CANNOT USE s1 ANYMORE!
```

RISC-V

Assembler Directives

Assembler Directive	Description
`.text`	Text section
`.data`	Global data section
`.section .foo`	Section names `.foo`
`.align N`	Align next data/instr on 2^N-byte boundary
`.balign N`	Align next data/instr on N-byte boundary
`.globl sym`	Label sym is global
`.string “str”`	Store string ”str” in memory
`.word w1, w2, …, wN`	Store N 32bit values in successive memory
`.byte b1, b2, …, bN`	Store N 8bit values in successive memory
`.space N`	Reserve N bytes to stroe variables
`.equ name, constant`	Define symbol name with value `constant`
`.end`	`End of assembly code`

RISC-V

Example using Assembler Directives

```
.globl main # make the main label global
.equ N, 5 # N = 5

.data # global data segment
A: .word 5, 42, -88, 2, -5033, 720, 314
str1: .string "RISC-V"
.align 2 # align next data on 2^2-byte boundary
B: .word 0x32A

.bss # bss segment – variables initialized to 0
C: .space 4
D: .space 1

.balign 4 # align next instruction on 4-byte boundary
.text # text segment (code)
main:
la t0, A # t0 = address of A = 0x2150
la t1, str1 # t1 = address of str1 = 0x216C
la t2, B # t2 = address of B = 0x2174
la t3, C # t3 = address of C = 0x2188
la t4, D # t4 = address of D = 0x218C
lw t5, N*4(t0) # t5 = A[N] = A[5] = 720 = 0x2D0
lw t6, 0(t2) # t6 = B = 810 = 0x32A
add t5, t5, t6 # t5 = A[N] + C = 720 + 810 = 1530 = 0x5FA
sw t5, 0(t3) # C = 1530 = 0x5FA
lb t5, N-1(t1) # t5 = str1[N-1] = str1[4] = '-' = 0x2D
sb t5, 0(t4) # D = str1[N-1] = 0x2D
la t5, str2 # t5 = address of str2 = 0x140
lb t6, 8(t5) # t6 = str2[8] = 'r' = 0x72
sb t6, 0(t1) # str1[0] = 'r' = 0x72
jr ra # return

.section .rodata
str2: .string "Hello world!"
.end # end of assembly file
```

References

[1]
D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - RISC-V Edition*, First Edition. Elsevier, 2017.

[2]
S. L. Harris and D. M. Harris, “Digital Design and Computer Architecture RISC-V Edition,” in *Digital Design and Computer Architecture*, First Edition., Elsevier, 2022, pp. IBC1–IBC2. doi: 10.1016/B978-0-12-820064-3.00025-8.

[3]
“Flynn’s taxonomy,” *Wikipedia*. Jul. 03, 2022. Accessed: Aug. 04, 2022. [Online]. Available:
https://en.wikipedia.org/w/index.php?title=Flynn%27s_taxonomy&oldid=1096210749

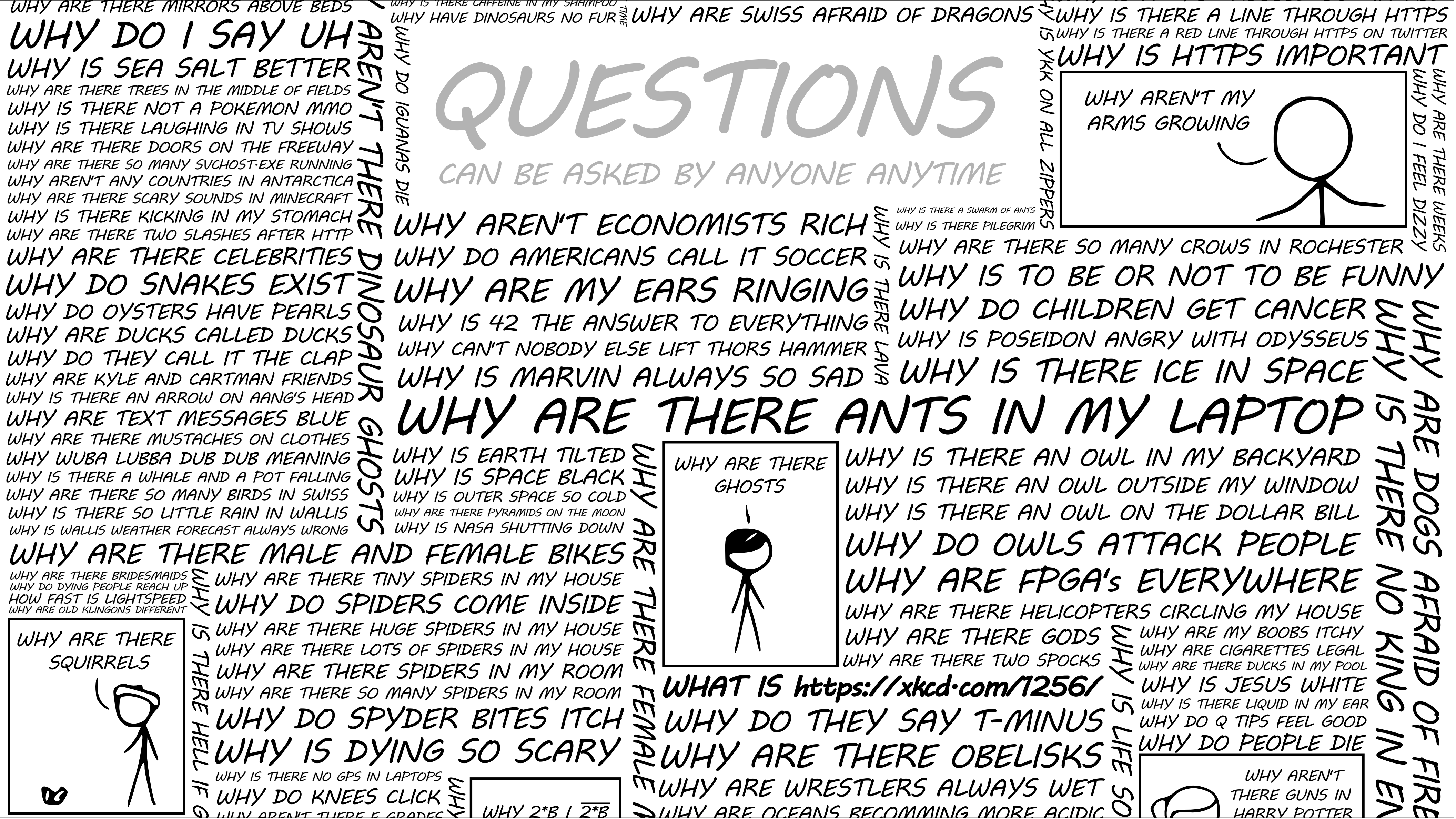
[4]
“Instruction cycle,” *Wikipedia*. Apr. 20, 2022. Accessed: May 29, 2022. [Online]. Available:
https://en.wikipedia.org/w/index.php?title=Instruction_cycle&oldid=1083738942

[5]
F. Embeddev, “ISA Resources,” *Five EmbedDev*. https://www.five-embeddev.com//riscv-isa-manual/ (accessed Jun. 04, 2022).

[6]
E. Roberts, “RISC vs. CISC.” https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/ (accessed Jun. 02, 2022).

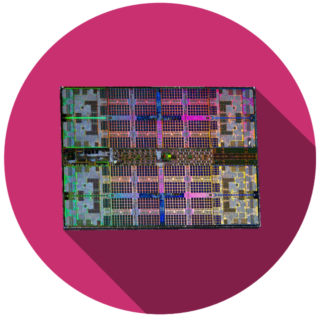
[7]
A. Waterman, K. Asanovic, and F. Embeddev, “RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA,” *Five EmbedDev*, 2019.
https://www.five-embeddev.com//riscv-isa-manual/latest/riscv-spec.html (accessed Jun. 04, 2022).

[8]
F. Embeddev, “RISC-V Quick Reference,” *Five EmbedDev*, 2022. https://www.five-embeddev.com//quickref/tools.html (accessed Jun. 04, 2022).


QUESTIONS
CAN BE ASKED BY ANYONE ANYTIME
WHY ARE THERE MIRRORS ABOVE BEDS
WHY DO I SAY UH
WHY IS SEA SALT BETTER
WHY ARE THERE TREES IN THE MIDDLE OF FIELDS
WHY IS THERE NOT A POKEMON MMO
WHY IS THERE LAUGHING IN TV SHOWS
WHY ARE THERE DOORS ON THE FREEWAY
WHY ARE THERE SO MANY SVCHOST.EXE RUNNING
WHY AREN'T ANY COUNTRIES IN ANTARCTICA
WHY ARE THERE SCARY SOUNDS IN MINECRAFT
WHY IS THERE KICKING IN MY STOMACH
WHY ARE THERE TWO SLASHES AFTER HTTP
WHY ARE THERE CELEBRITIES
WHY DO SNAKES EXIST
WHY DO OYSTERS HAVE PEARLS
WHY ARE DUCKS CALLED DUCKS
WHY DO THEY CALL IT THE CLAP
WHY ARE KYLE AND CARTMAN FRIENDS
WHY IS THERE AN ARROW ON AANG'S HEAD
WHY ARE TEXT MESSAGES BLUE
WHY ARE THERE MUSTACHES ON CLOTHES
WHY WUBA LUBBA DUB DUB MEANING
WHY IS THERE A WHALE AND A POT FALLING
WHY ARE THERE SO MANY BIRDS IN SWISS
WHY IS THERE SO LITTLE RAIN IN WALLIS
WHY IS WALLIS WEATHER FORECAST ALWAYS WRONG
WHY AREN'T THERE DINOSAUR GHOSTS
WHY HAVE DINOSAURS NO FUR
WHY DO IGUANAS DIE
TIME
WHY ARE SWISS AFRAID OF DRAGONS
WHY IS THERE A LINE THROUGH HTTPS
WHY IS THERE A RED LINE THROUGH HTTPS ON TWITTER
WHY IS HTTPS IMPORTANT
WHY IS YKK ON ALL ZIPPERS
WHY AREN'T MY ARMS GROWING
WHY ARE THERE WEEKS
WHY DO I FEEL DIZZY
WHY IS THERE A SWARM OF ANTS
WHY IS THERE PILEGRIM
WHY AREN'T ECONOMISTS RICH
WHY DO AMERICANS CALL IT SOCCER
WHY ARE MY EARS RINGING
WHY IS 42 THE ANSWER TO EVERYTHING
WHY CAN'T NOBODY ELSE LIFT THORS HAMMER
WHY IS MARVIN ALWAYS SO SAD
WHY IS THERE LAVA
WHY ARE THERE SO MANY CROWS IN ROCHESTER
WHY IS TO BE OR NOT TO BE FUNNY
WHY DO CHILDREN GET CANCER
WHY IS POSEIDON ANGRY WITH ODYSSEUS
WHY IS THERE ICE IN SPACE
WHY ARE THERE ANTS IN MY LAPTOP
WHY IS EARTH TILTED
WHY IS SPACE BLACK
WHY IS OUTER SPACE SO COLD
WHY ARE THERE PYRAMIDS ON THE MOON
WHY IS NASA SHUTTING DOWN
WHY ARE THERE GHOSTS
WHY IS THERE AN OWL IN MY BACKYARD
WHY IS THERE AN OWL OUTSIDE MY WINDOW
WHY IS THERE AN OWL ON THE DOLLAR BILL
WHY DO OWLS ATTACK PEOPLE
WHY ARE FPGA's EVERYWHERE
WHY ARE THERE HELICOPTERS CIRCLING MY HOUSE
WHY ARE THERE GODS
WHY ARE THERE TWO SPOCKS
WHY ARE MY BOOBS ITCHY
WHY ARE CIGARETTES LEGAL
WHY ARE THERE DUCKS IN MY POOL
WHY IS JESUS WHITE
WHY IS THERE LIQUID IN MY EAR
WHY DO Q TIPS FEEL GOOD
WHY DO PEOPLE DIE
WHY AREN'T THERE GUNS IN HARRY POTTER
WHY ARE THERE MALE AND FEMALE BIKES
WHY ARE THERE BRIDESMAIDS
WHY DO DYING PEOPLE REACH UP
HOW FAST IS LIGHTSPEED
WHY ARE OLD KLINGONS DIFFERENT
WHY ARE THERE SQUIRRELS
WHY ARE THERE TINY SPIDERS IN MY HOUSE
WHY DO SPIDERS COME INSIDE
WHY ARE THERE HUGE SPIDERS IN MY HOUSE
WHY ARE THERE LOTS OF SPIDERS IN MY HOUSE
WHY ARE THERE SPIDERS IN MY ROOM
WHY ARE THERE SO MANY SPIDERS IN MY ROOM
WHY DO SPYDER BITES ITCH
WHY IS DYING SO SCARY
WHY IS THERE NO GPS IN LAPTOPS
WHY DO KNEES CLICK
WHAT IS https://xkcd.com/1256/
WHY DO THEY SAY T-MINUS
WHY ARE THERE OBELISKS
WHY ARE WRESTLERS ALWAYS WET
WHY ARE OCEANS BECOMMING MORE ACIDIC

Hes·so // VALAIS WALLIS

Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften



Silvan Zahno silvan.zahno@hevs.ch