Computer Organization and Networks

(INB.06000UF, INB.07001UF)

Chapter 6: Programming a RISC-V CPU

Winter 2020/2021

TU Graz

Stefan Mangard, www.iaik.tugraz.at

Software

The Software/Hardware Interface: Instruction Set Architecture (ISA):

- The ISA defines anything that is needed by programmers to correctly write a program for the hardware.
- In particular this includes defining, instructions, registers, data types, memory model, …

Hardware

Software

The Software/Hardware Interface: Instruction Set Architecture (ISA):
- The ISA defines anything that is needed by programmers to correctly write a program for the hardware.
- In particular this includes defining, instructions, registers, data types, memory model, …

- A **microarchitecture** defines how the instruction set is implemented in a concrete processor. This includes all details from realizing the register file and ALU up to pipelining, out-of-order execution, …

Hardware

- Note: the programmer does not need to care about the microarchitecture (i.e. the concrete realization of the ISA)

- The software tool chain maps program description in all kinds programming languages down to machine language (i.e. instructions that the CPU can execute)

Software

The Software/Hardware Interface: Instruction Set Architecture (ISA):

- The ISA defines anything that is needed by programmers to correctly write a program for the hardware.
- In particular this includes defining, instructions, registers, data types, memory model, …

- A **microarchitecture** defines how the instruction set is implemented in a concrete processor. This includes all details from realizing the register file and ALU up to pipelining, out-of-order execution, …

Hardware

- Note: the programmer does not need to care about the microarchitecture (i.e. the concrete realization of the ISA)

Micro RISC-V

Micro RISC-V

- Micro RISC-V is a very simple CPU that we use for our introductory programming examples
- Micro RISC-V implements a subset of R32I
- Tools and code for micro RISC-V
  - Code for Micro RISC-V and examples are available in the examples-2020 repo
  - Assembler: riscvasm.py
  - Simulator: riscvsim.py

Micro RISC-V Overview

Registers:

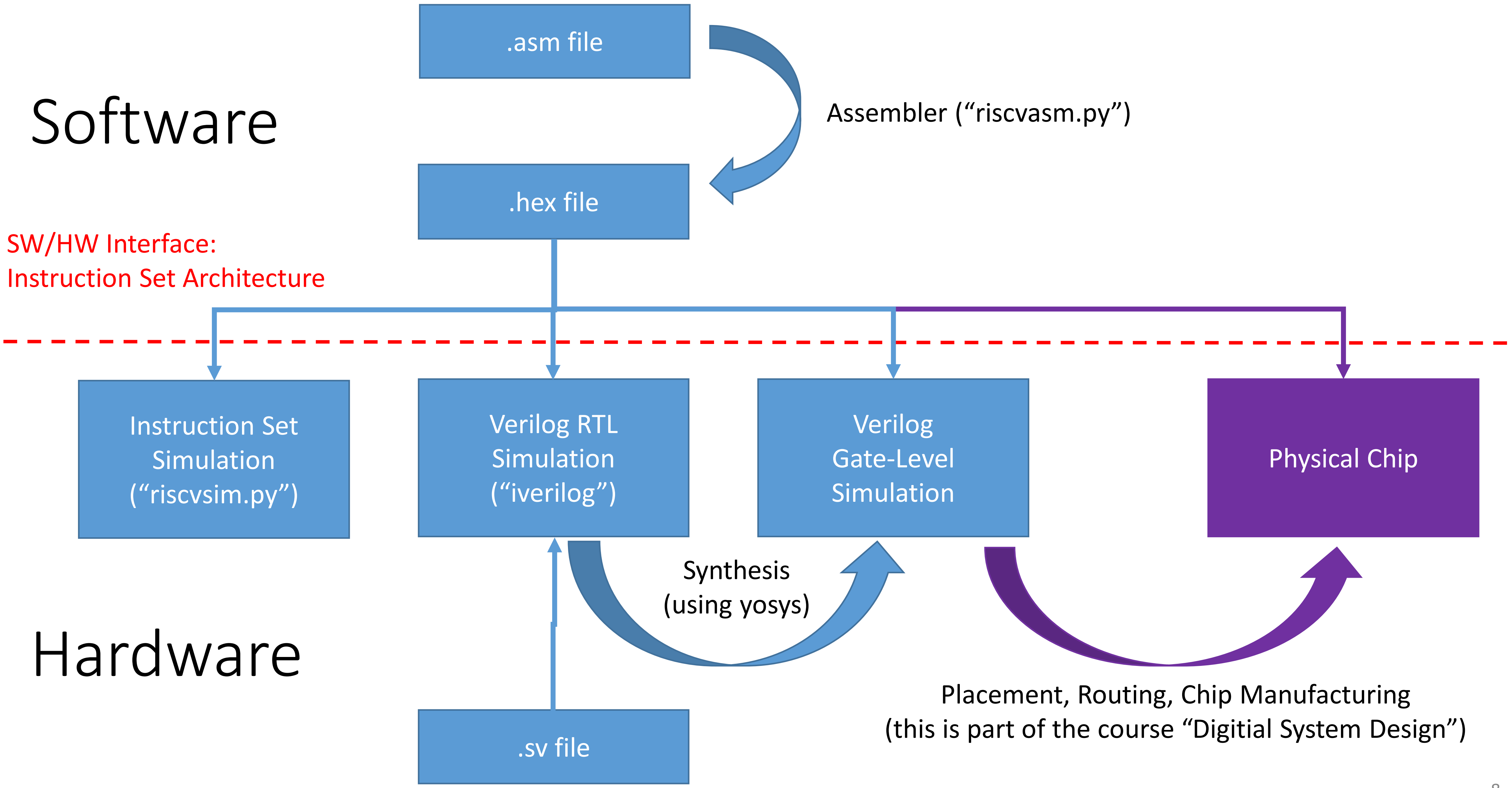
- Zero Register: `x0`
- General Purpose Registers: `x1` - `x31`

Memory:

- almost 2 KiB of Memory (`0x000` - `0x7fc`)
- memory-mapped I/O at address `0x7fc`

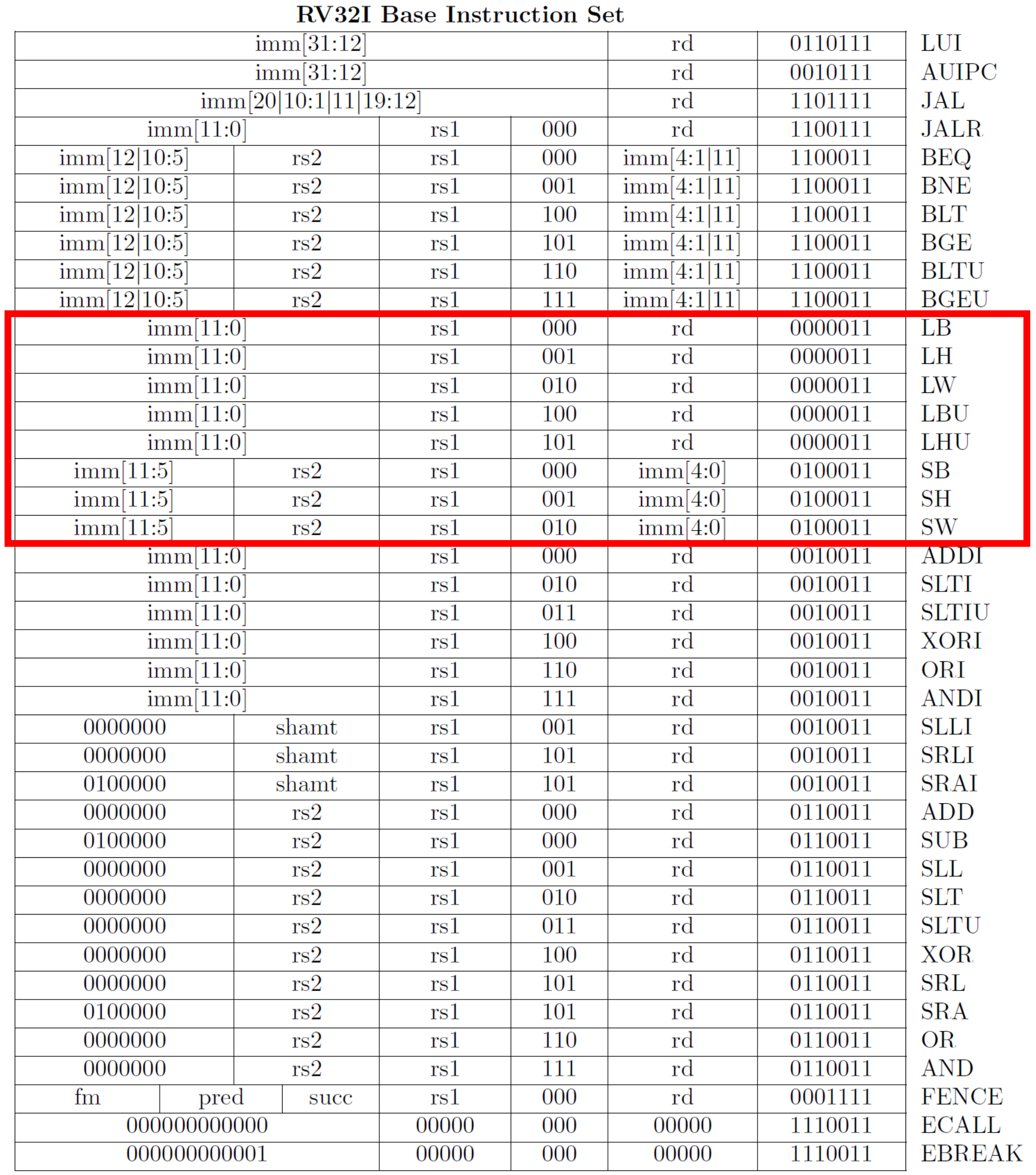
Instructions:

- ALU: `OP rd, rs1, rs2`
  - ADD, SUB, AND, OR, XOR, SLL, SRL, SRA
- Add immediate: `ADDI rd, rs1, value`
- Load upper immediate: `LUI rd, value`
- Branch: `OP rs1, rs2, offset`
  - BEQ, BNE, BLT, BGE
- Jump / Call: `JAL rd, offset`
- Jump / Call indirect: `JALR rd, offset(rs1)`
- Load: `LW rd, offset(rs1)`
- Store: `SW rs2, offset(rs1)`
- Halt: `EBREAK`


Software
.asm file
Assembler ("riscvasm.py")
.hex file
SW/HW Interface:
Instruction Set Architecture
Instruction Set Simulation ("riscvsim.py")
Verilog RTL Simulation ("iverilog")
Verilog Gate-Level Simulation
Physical Chip
Synthesis (using yosys)
Placement, Routing, Chip Manufacturing
(this is part of the course "Digitial System Design")
Hardware
.sv file

Input and Output Devices

RV32I Base Instruction Set

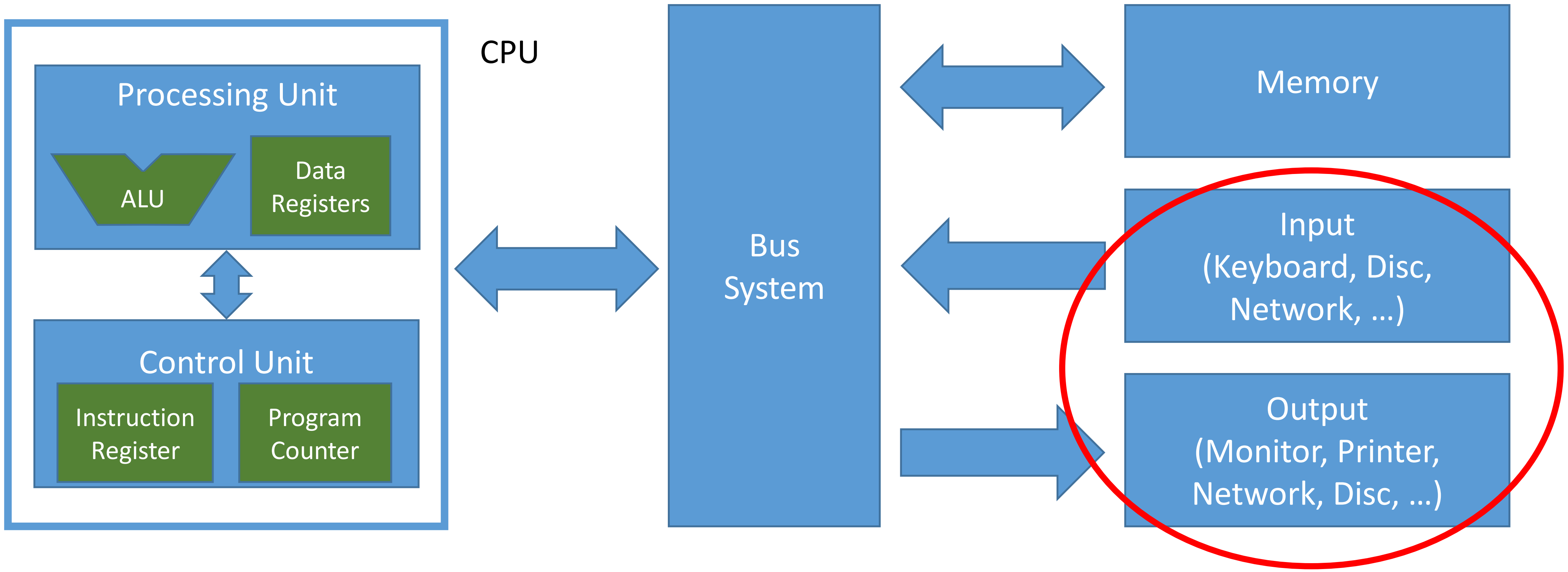


imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

How to Implement I/O?

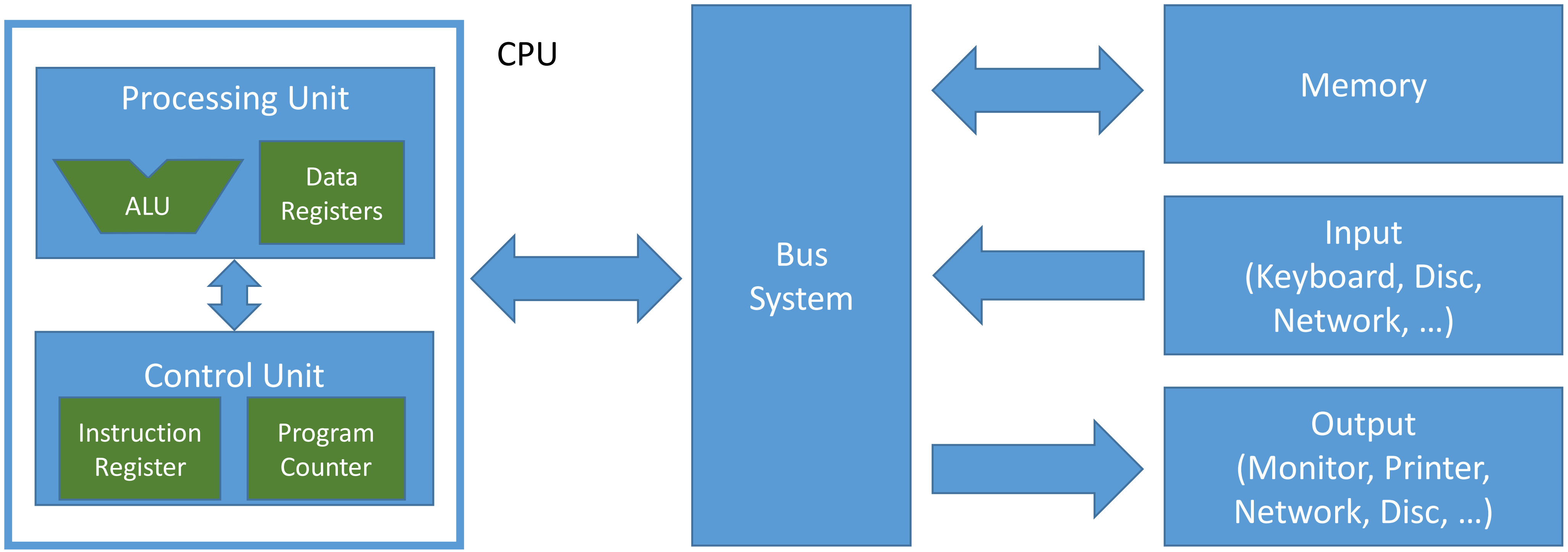
- We access I/O and other devices like memory

→ we build memory-mapped peripherals


CPU
Processing Unit
ALU
Data Registers
Control Unit
Instruction Register
Program Counter
Bus System
Memory
Input (Keyboard, Disc, Network, …)
Output (Monitor, Printer, Network, Disc, …)

Memory-Mapped Peripherals and I/O

- Store and load instructions allow addressing 32-bit of memory space
- Not all the memory space that is addressable is used for actual memory
- We can split the memory space in pieces and assign a certain range to actual memory and other ranges to peripherals or I/O:
  - Peripherals: load/store operations write to registers of state machines with additional functionality (Co-processors, sound, graphics, …)
  - I/O: We implement input and output as wires connecting to our CPU

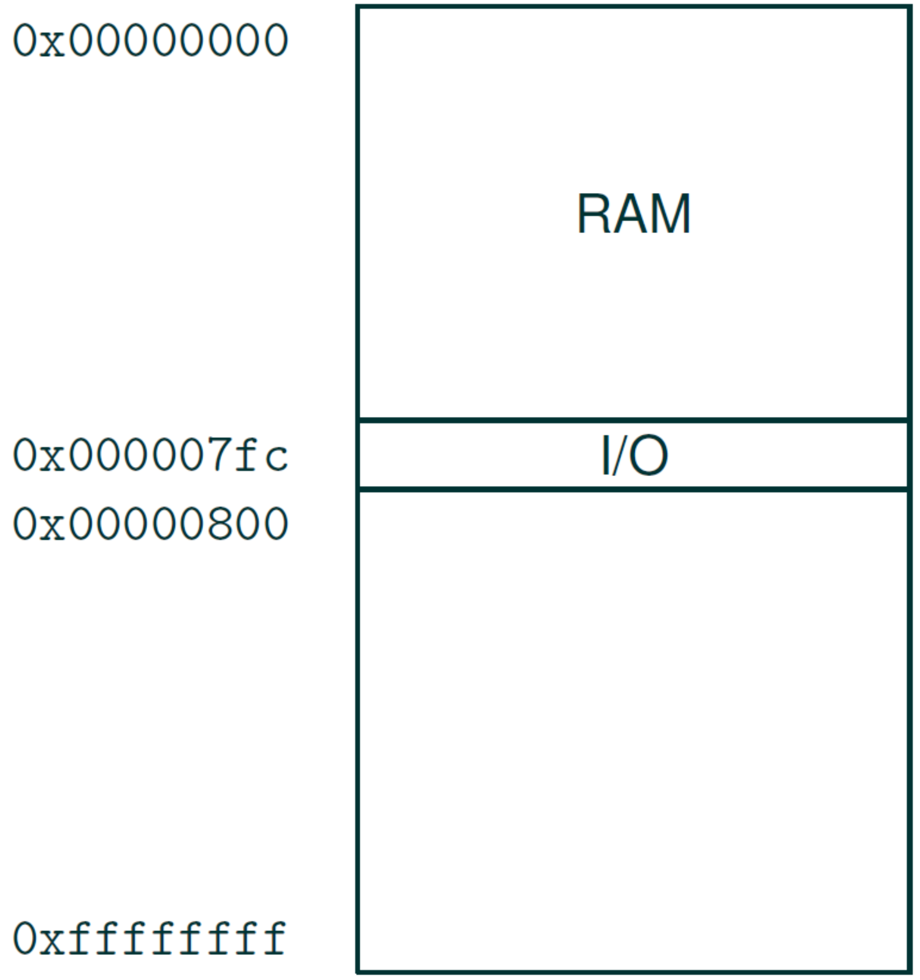


The bus system takes care of routing the load/store operations to the correct physical device as defined by the memory ranges

Memory Map in Micro RISC-V

- In Micro RISC-V, the physical memory map is as follows:
  - RAM is located from 0x00000000 to 0x000007FB
  - I/O is located at address 0x000007FC
  - The remaining memory range is not connected (write has no effect; read returns 0)

- The physical memory map is defined for each device depending on size of memory, peripherals, etc.



Programming in Assembly

Note on ASM Examples

- Run “make” to generate .hex files

- Run “make run” to assemble and run the .asm file in the current working directory with the RTL simulator (micro-RISCV)

- Run “make sim” to simulate the .asm file in the current working directory with the python asmlib RISC-V simulator

If there are more than one asm files in the current working directory, you need to specify the target explicitly using “make run=the_asm_file_without_file_extension_suffix” (and accordingly for “make sim”).

Read/Write from Memory vs. Read Write from I/O

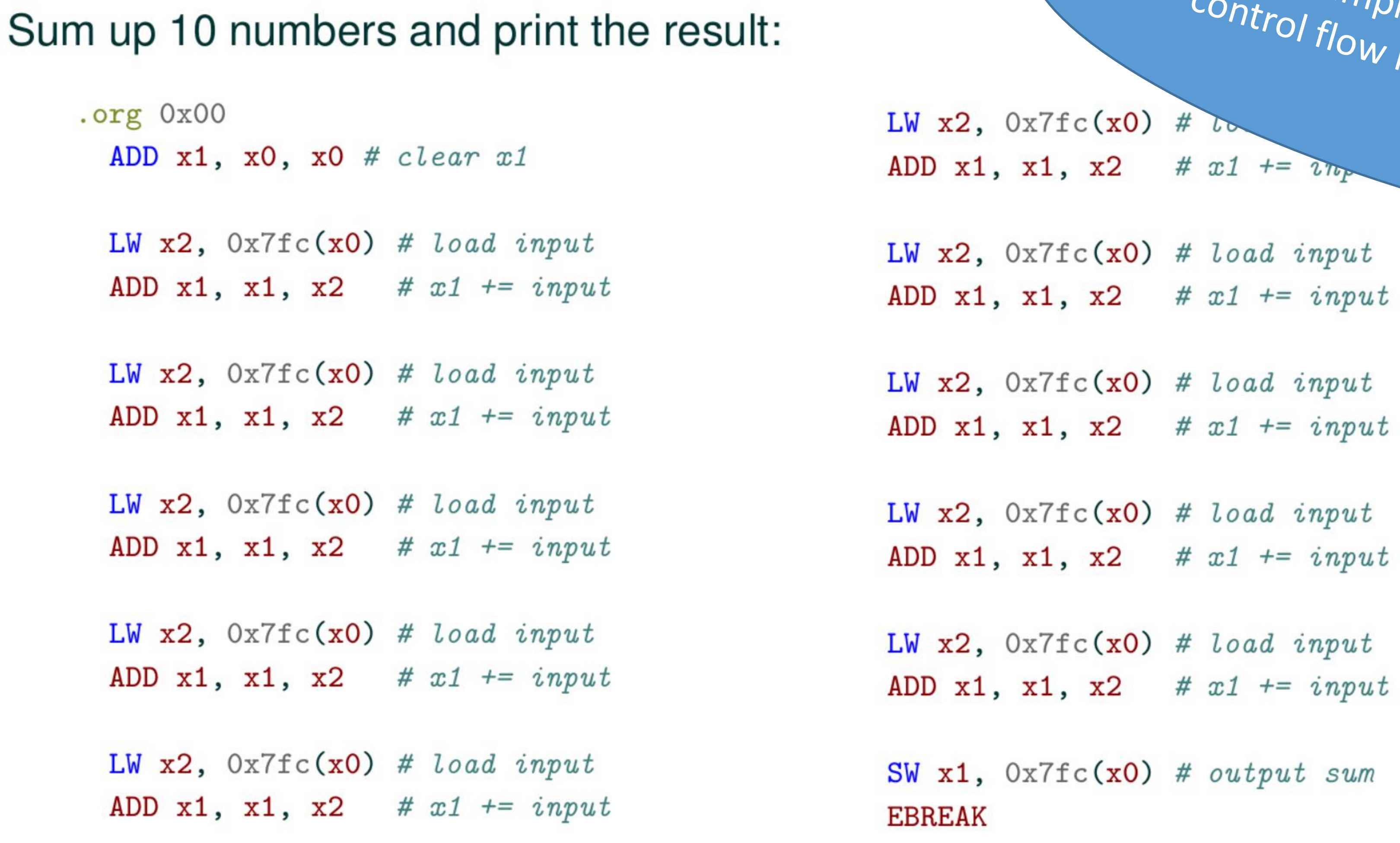
con04_adding-two-constants

```
.org 0x00
  # read from memory
  LW x1, 0x20(x0)
  LW x2, 0x24(x0)
  ADD x3, x1, x2
  # write to memory
  SW x3, 0x24(x0)
  EBREAK
```

con06_adding-stdin-numbers

```
.org 0x00
  # read from I/O
  LW x1, 0x7fc(x0)
  LW x2, 0x7fc(x0)
  ADD x3, x1, x2
  # write to I/O
  SW x3, 0x7fc(x0)
  EBREAK
```

Summing Up 10 Input Values



Sum up 10 numbers and print the result:

```
.org 0x00
  ADD x1, x0, x0 # clear x1

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

LW x2, 0x7fc(x0) # lo
ADD x1, x1, x2   # x1 += inp

LW x2, 0x7fc(x0) # load input
ADD x1, x1, x2   # x1 += input

LW x2, 0x7fc(x0) # load input
ADD x1, x1, x2   # x1 += input

LW x2, 0x7fc(x0) # load input
ADD x1, x1, x2   # x1 += input

LW x2, 0x7fc(x0) # load input
ADD x1, x1, x2   # x1 += input

SW x1, 0x7fc(x0) # output sum
EBREAK
```

Let's improve this code using control flow instructions to build a loop

Loops

- start with the code for one iteration

```
.org 0x00
  ADD x1, x0, x0    # clear x1



  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input



  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Loops

- start with the code for one iteration
- add loop variables

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input




  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Loops

- start with the code for one iteration
- add loop variables
- increment the counter

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input

  ADDI x3, x3, 1    # counter++


  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Loops

- start with the code for one iteration
- add loop variables
- increment the counter
- branch to the start of the loop

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input

  ADDI x3, x3, 1    # counter++
  BLT x3, x4, ???   # if (counter < 10) loop

  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Loops

Counting offsets is not a nice job for a programmer

→ Let the compiler do it

- start with the code for one iteration
- add loop variables
- increment the counter
- branch to the start of the loop

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input

  ADDI x3, x3, 1    # counter++
  BLT x3, x4, -12   # if (counter < 10) loop

  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Symbols

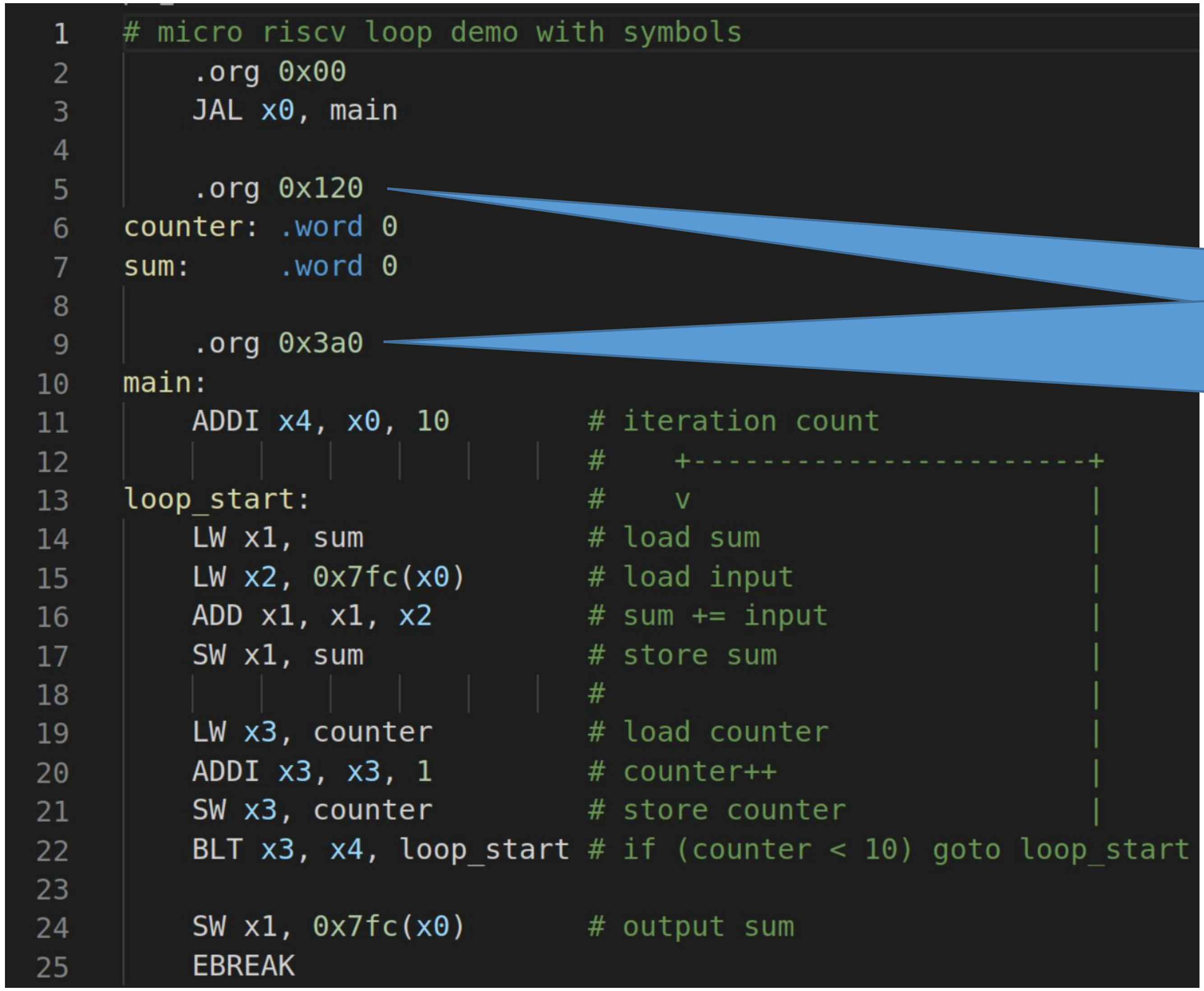
- Basic idea:
  - We label memory addresses
  - Each address we label is assigned a symbol (“a name”)

- When programming, we can replace memory addresses by symbols to simplify the complexity of programming

Loop Using a Label

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count
loop:
  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input

  ADDI x3, x3, 1    # counter++
  BLT x3, x4, loop # if (counter < 10) loop

  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

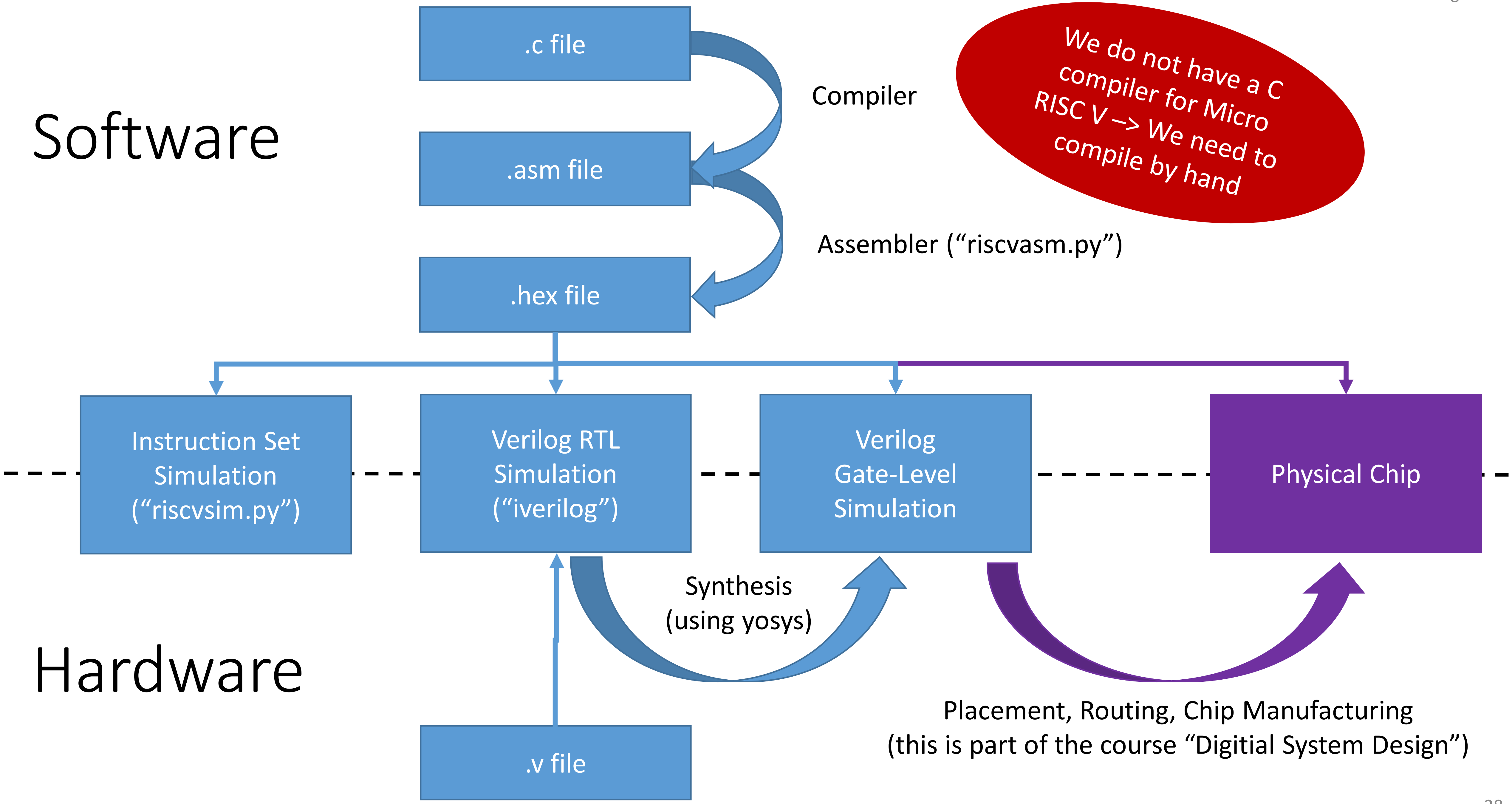
Variables, Having Fun With the Memory Layout



```
# micro riscv loop demo with symbols
    .org 0x00
    JAL x0, main

    .org 0x120
counter: .word 0
sum:     .word 0

    .org 0x3a0
main:
    ADDI x4, x0, 10        # iteration count
                           #    +-----------------------+
loop_start:                #    v                       |
    LW x1, sum             # load sum                   |
    LW x2, 0x7fc(x0)       # load input                 |
    ADD x1, x1, x2         # sum += input               |
    SW x1, sum             # store sum                  |
                           #                            |
    LW x3, counter         # load counter               |
    ADDI x3, x3, 1         # counter++                  |
    SW x3, counter         # store counter              |
    BLT x3, x4, loop_start # if (counter < 10) goto loop_start

    SW x1, 0x7fc(x0)       # output sum
    EBREAK
```

- We can choose the memory layout as we like
- We can mix data and code
- Try it out with your own code

Programming in C


Software
.c file
Compiler
.asm file
Assembler ("riscvasm.py")
.hex file
We do not have a C compiler for Micro RISC V –> We need to compile by hand
Instruction Set Simulation ("riscvsim.py")
Verilog RTL Simulation ("iverilog")
Verilog Gate-Level Simulation
Physical Chip
Synthesis (using yosys)
Placement, Routing, Chip Manufacturing (this is part of the course "Digitial System Design")
Hardware
.v file

Program in C

```
while (1) {
        scanf(“%x”, &a);
        if (a==0) break;
        printf(“%x”, a);
}
```

“Simplification”: While → If, goto

```
while (1) {
        scanf(“%x”, &a);
        if (a==0) break;
        printf(“%x”, a);
}
```

```
L0:   scanf(“%x”, &a);
            if (a == 0) goto L1;
            printf(“%x”,a);
            goto L0;
L1:   ;
```

From C to RISCV assembly language

Labels

```
L0:    scanf("%x", &a);
              if (a == 0) goto L1;
              printf("%x",a);
              goto L0;
L1:    ;
```

From C to RISCV assembly language

Labels

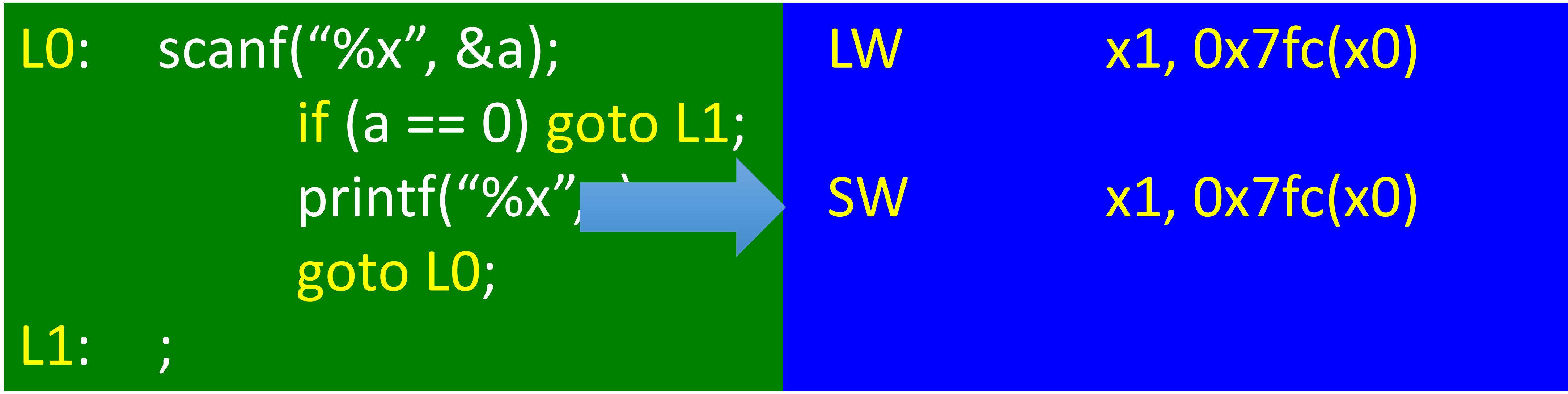
Copy value from
location 0x7fc
to CPU register x1.

```
L0:    scanf("%x", &a);
             if (a == 0) goto L1;
             printf("%x",a);
             goto L0;
L1:    ;
```

```
LW          x1, 0x7fc(x0)
```

From C to RISCV assembly language

Labels



```
L0:     scanf("%x", &a);
              if (a == 0) goto L1;
              printf("%x", 
              goto L0;
L1:     ;
```

```
LW          x1, 0x7fc(x0)

SW          x1, 0x7fc(x0)
```

Store (= copy) value
in CPU register x1
to address 0x7fc

From C to RISCV assembly language

Labels

```
L0:    scanf(“%x”, &a);
              if (a == 0) g
              printf(“%x”,a);
              goto L0;
L1:    ;
```

```
LW        x1, 0x7fc(x0)
BEQ       x1, x0, L1
SW        x1, 0x7fc(x0)
JAL       x0,L0
```

If value in CPU register x1 is equal to 0,
Then goto label L1. Else continue with
the statement after the if-statement.

From C to RISCV assembly language

Labels

```
L0:   scanf("%x", &a);
          if (a == 0) goto L1;
          printf("%x",a);
          goto L0;
L1:   ;
```

```
LW      x1, 0x7fc(x0)
BEQ     x1, x0, L1
SW      x1, 0x7fc(x0)
JAL     x0,L0
```

This statement stands for
a unconditional "goto".

From C to RISCV assembly language

Labels

```
L0:    scanf("%x", &a);
           if (a == 0) goto L1;
           printf("%x",a);
           goto L0;
L1:    ;
```

```
LW        x1, 0x7fc(x0)
BEQ       x1, x0, L1
SW        x1, 0x7fc(x0)
JAL       x0,L0
EBREAK
```

The execution of the instruction EBREAK halts the CPU simulation.

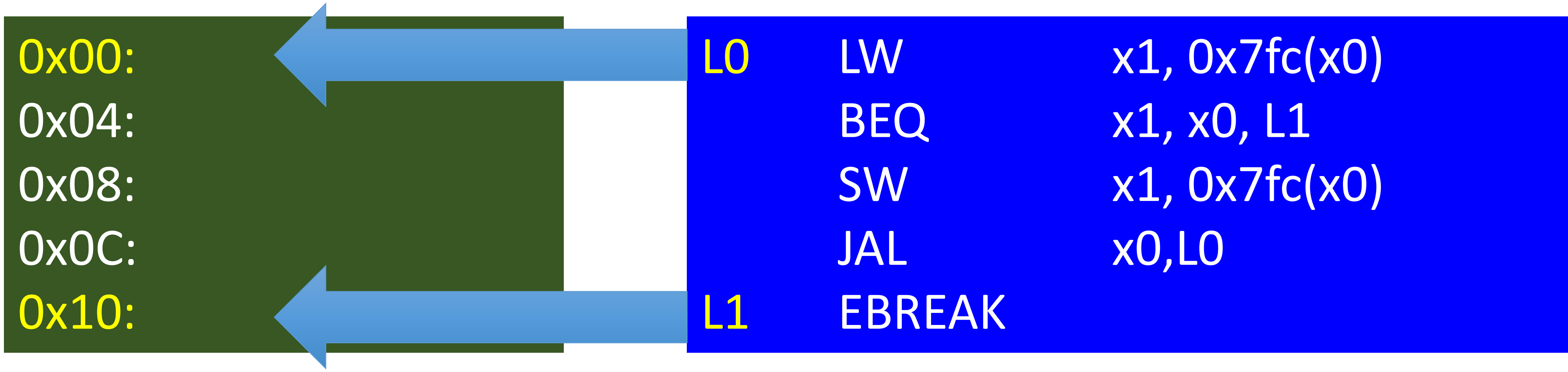
From assembly language to machine language

```
0x00:
0x04:
0x08:
0x0C:
0x10:
```

```
L0      LW          x1, 0x7fc(x0)
        BEQ         x1, x0, L1
        SW          x1, 0x7fc(x0)
        JAL         x0,L0
L1      EBREAK
```

TOY starts executing code at address 0x00.
Every machine instruction needs one word in memory.

Labels are “symbolic addresses”



```
L0    LW        x1, 0x7fc(x0)
      BEQ       x1, x0, L1
      SW        x1, 0x7fc(x0)
      JAL       x0,L0
L1    EBREAK
```

The label “L0” is a symbolic name for the memory location with address 0x00.
Likewise, the label “L1” is a symbolic name for the memory location with address 0x10.

```
0x00:    0x7F C0 20 83
0x04:
0x08:
0x0C:
0x10:
```

```
        LW        x1, 0x7fc(x0)
        BEQ       x1, x0, L1
        SW        x1, 0x7fc(x0)
        JAL       x0,L0
L1      EBREAK
```

```
0x00:    0x 7F C0 20 83
0x04:    0x 00 00 86 63
0x08:
0x0C:
0x10:
```

```
L0    LW        x1, 0x7fc(x0)
      BEQ       x1, x0, L1
      SW        x1, 0x7fc(x0)
      JAL       x0,L0
L1    EBREAK
```

```
0x00:    0x 7F C0 20 83
0x04:    0x 00 00 86 63
0x08:    0x 7E 10 2E 23
0x0C:
0x10:
```

```
L0    LW        x1, 0x7fc(x0)
      BEQ       x1, x0, L1
      SW        x1, 0x7fc(x0)
      JAL       x0,L0
L1    EBREAK
```

```
0x00:    0x 7F C0 20 83
0x04:    0x 00 00 86 63
0x08:    0x 7E 10 2E 23
0x0C:    0x FF 5F F0 6F
0x10:
```

```
L0    LW        x1, 0x7fc(x0)
      BEQ       x1, x0, L1
      SW        x1, 0x7fc(x0)
      JAL       x0,L0
L1    EBREAK
```

```
0x00:    0x 7F C0 20 83
0x04:    0x 00 00 86 63
0x08:    0x 7E 10 2E 23
0x0C:    0x FF 5F F0 6F
0x10:    0x 00 10 00 73
```

```
L0    LW        x1, 0x7fc(x0)
      BEQ       x1, x0, L1
      SW        x1, 0x7fc(x0)
      JAL       x0,L0
      EBREAK
```

The Machine Program

```
0x00:    0x 7F C0 20 83
0x04:    0x 00 00 86 63
0x08:    0x 7E 10 2E 23
0x0C:    0x FF 5F F0 6F
0x10:    0x 00 10 00 73
```

The Machine Program in Binary Notation

```
0x00:     0x 7F C0 20 83
0x04:     0x 00 00 86 63
0x08:     0x 7E 10 2E 23
0x0C:     0x FF 5F F0 6F
0x10:     0x 00 10 00 73
```

```
0x00: 0111_1111_1100_0000_0010_0000_1000_0011
0x14: 0000_0000_0000_0000_1000_0110_0110_0011
0x08: 0111_1110_0001_0000_0010_1110_0010_0011
0x0C: 1111_1111_0101_1111_1111_0000_0110_1111
0x10: 0000_0000_0001_0000_0000_0000_0111_0011
```

For reasons of readability, we use hexadecimal notation.

In memory we always only have binary patterns.

Let’s do a More Complex Example

```c
/** Task
 * Write an ASM program that adds all array elements
 * and writes the sum to stdout.
 *
 ** Approach
 * Write a C program (see below).
 * Then modify the C source code in a way such that
 * each code line can be directly translated into
 * RISC-V assembly language.
 */
#include <stdio.h>

int n        = 4;
int array[4] = {3, 4, 5, 6};
int sum      = 0;

int main() {
  for (int i=0; i<n; i++)
    sum = sum + array[i];

  printf("%d\n", sum);
}
```

- The program sums up 4 numbers and write the sum to stdout
- We translate the program from C to ASM step by step
- See examples repo for each step

Important Steps for the Transformation from C to ASM

- Transform all For/While loops into conditional goto statements (if + goto label)
- Resolve complex conditional statements and computational statements by using additional temporary variables → ASM instructions can only handle two operands
- Ensure the correct handling of the else branch when resolving if statements to (if + goto label) statements
- Make pointer arithmetic of e.g. arrays explicit and only use char* (because we have byte-level addressing in RISC-V ASM)

Function Calls, Calling Convention

Motivation

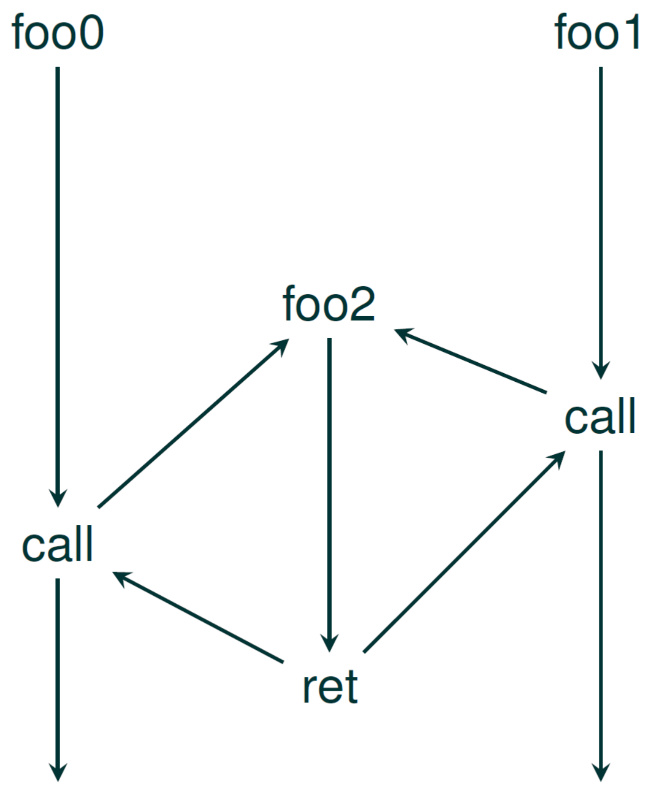
- The C to ASM translation we have done so far was limited
  - No function calls
  - Only global variables – no local variables in functions

- For real-world programs we want to partition our program into functions with local variables

Functions Calls

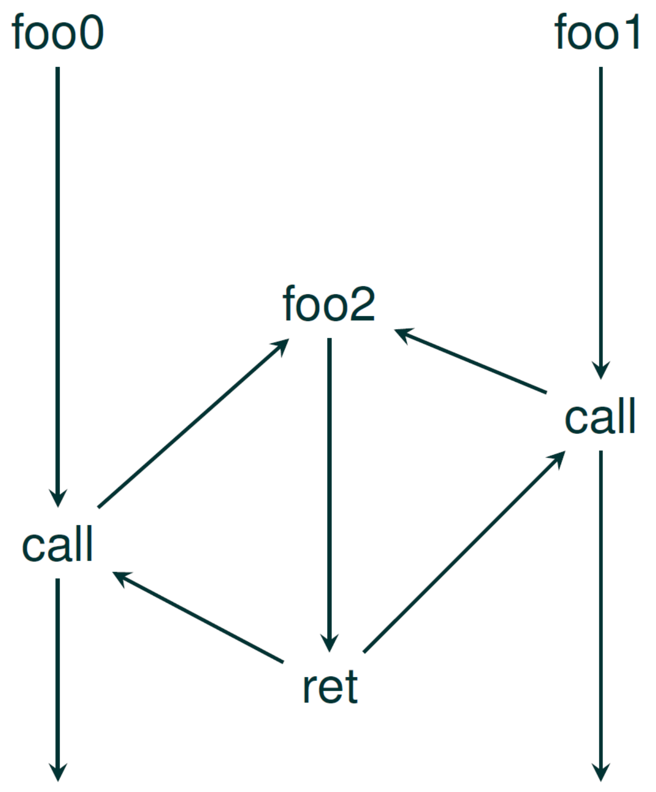
- Basic Idea:
  - partitioning of code into reusable functions
  - functions can call other functions arbitrarily (nested function calls, recursive function calls)

- Interface:
  - the function takes input arguments
  - the function provides a return value as output



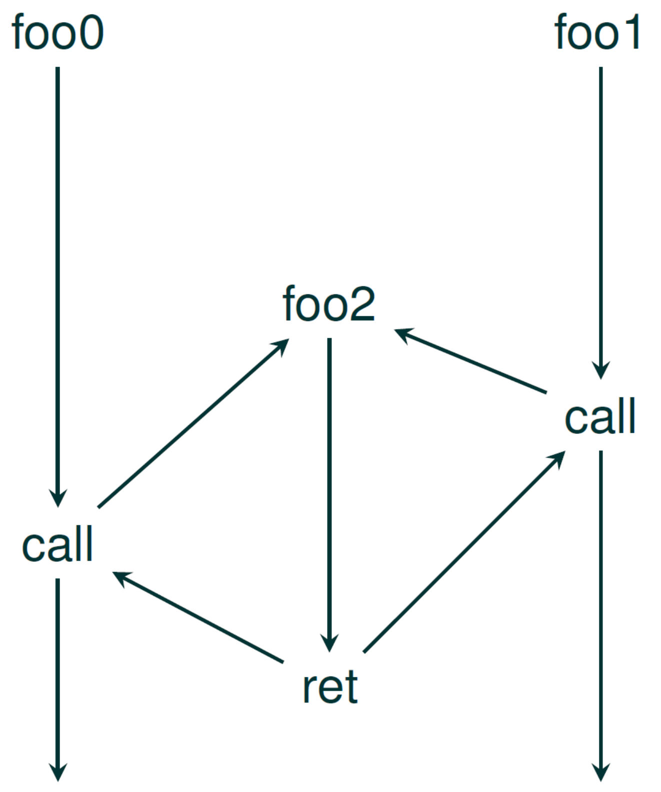
Realizing Function Calls and Returns

- A function call is not a simple branch instruction
- Whenever there is a function call, we also need to store the return address
  - foo2 needs to know whether to return to foo0 or foo1
  - The return address is a mandatory parameter to every function



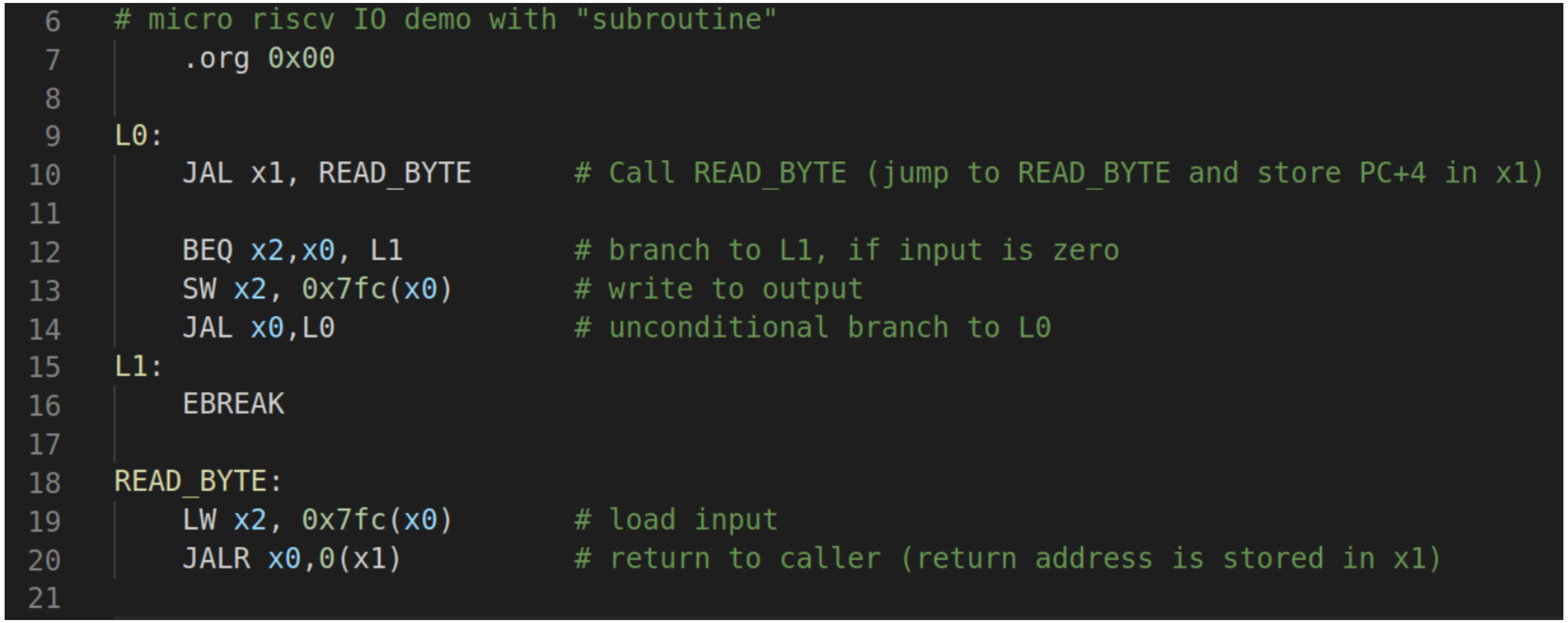
Realizing Function Calls and Returns on RISC-V

- RISC-V has two instructions to perform a “jump and link”
  - **JAL (Jump and Link):** JAL rd, offset
    - Jump relative to current PC
    - The jump destination is PC+offset
    - Upon the jump (PC+4) is stored in register rd
  - **JALR (Jump and Link Register):** JALR rd, rs, offset
    - Jump to address (register content from rs) + offset
    - Upon the jump (PC+4) is stored in register rd



Example

- See con06_function_call



```
# micro riscv IO demo with "subroutine"
    .org 0x00

L0:
    JAL x1, READ_BYTE       # Call READ_BYTE (jump to READ_BYTE and store PC+4 in x1)

    BEQ x2,x0, L1           # branch to L1, if input is zero
    SW x2, 0x7fc(x0)        # write to output
    JAL x0,L0               # unconditional branch to L0
L1:
    EBREAK

READ_BYTE:
    LW x2, 0x7fc(x0)        # load input
    JALR x0,0(x1)           # return to caller (return address is stored in x1)

```

Problem: Nested Subroutine Calls

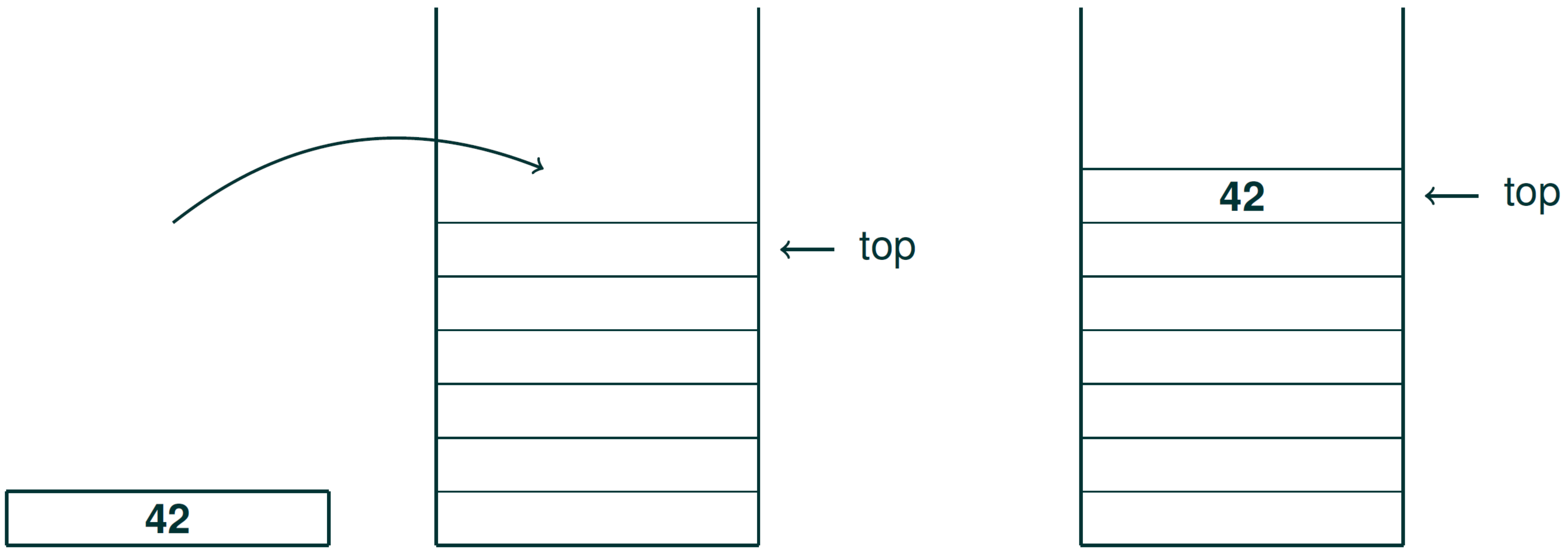
- JAL and JALR need a register for storing the return address
- We could use a different register for each function call. However, we would quickly run out of registers

→ We need a data structure in memory to take care of this.

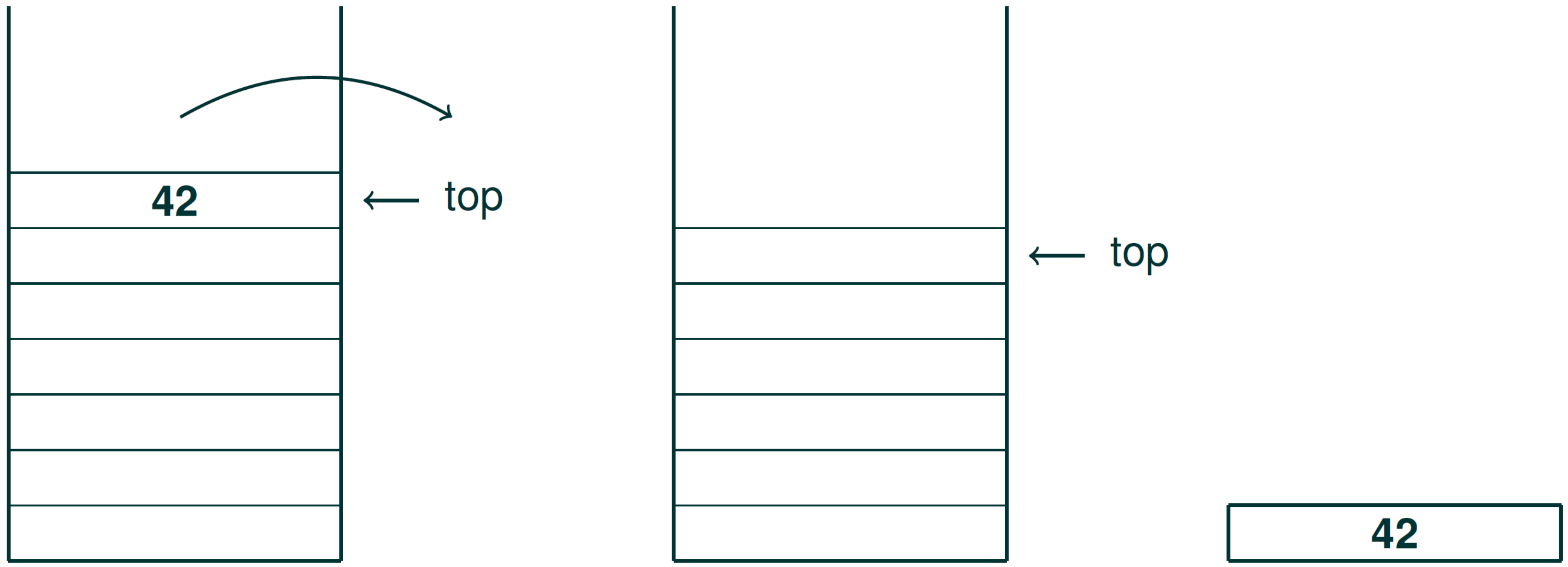
A Stack

- Stack characteristics:
  - Two operations:
    - “PUSH”: places an element on the stack
    - “POP”: receives an element from the stack
  - The stack is a FILO (first in, last out) data structure
  - The stack typically “grows” from high to low addresses
  - The stack is a contiguous section in memory
  - The “stack pointer” (sp) “points” to the “top of the stack” (TOS)

Push Value 42



Pop Value from Top of Stack



Implementing a Stack with RISC-V

- Initialize a stack pointer
  - Set starting point

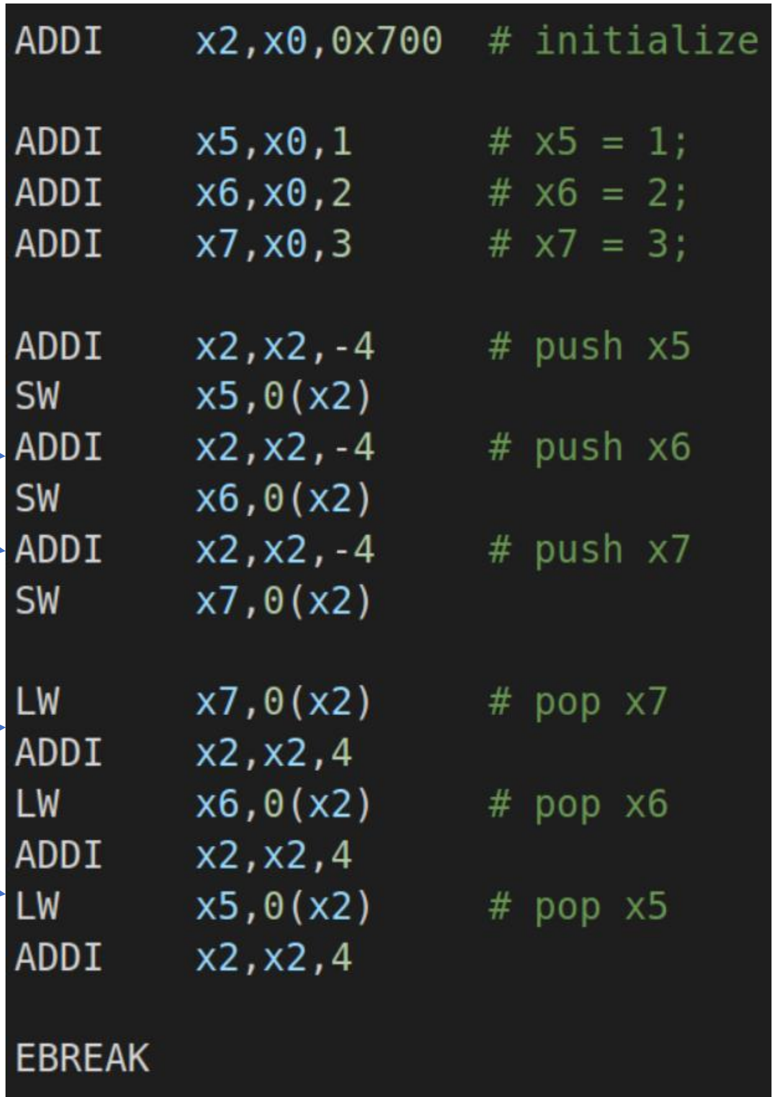
- Push value
  - Expand stack by 4
  - Copy value from register to top of stack

- Pop value
  - Copy value from top of stack to destination register
  - decrease stack by 4

Implementing a Stack with RISC-V

- Initialize a stack pointer
  - Set starting point
- Push value
  - Expand stack by 4
  - Copy value from register to top of stack
- Pop value
  - Copy value from top of stack to destination register
  - decrease stack by 4

push_pop.asm



```
ADDI    x2,x0,0x700  # initialize

ADDI    x5,x0,1      # x5 = 1;
ADDI    x6,x0,2      # x6 = 2;
ADDI    x7,x0,3      # x7 = 3;

ADDI    x2,x2,-4     # push x5
SW      x5,0(x2)
ADDI    x2,x2,-4     # push x6
SW      x6,0(x2)
ADDI    x2,x2,-4     # push x7
SW      x7,0(x2)

LW      x7,0(x2)     # pop x7
ADDI    x2,x2,4
LW      x6,0(x2)     # pop x6
ADDI    x2,x2,4
LW      x5,0(x2)     # pop x5
ADDI    x2,x2,4

EBREAK
```

Register Usage in Subroutines

- We can use a **stack to store return addresses**
- In fact, the stack can be used as a storage for any register
- Assume you want to use register x1, but it currently stores another value that is needed later on
  - Push x1 to the stack
  - Use x1
  - Restore x1 by popping the content from the stack

  →This is called "register spilling"

Idea:

→We can use the stack to store and restore register states when entering/exiting function calls

→Every function can use the CPU registers as needed

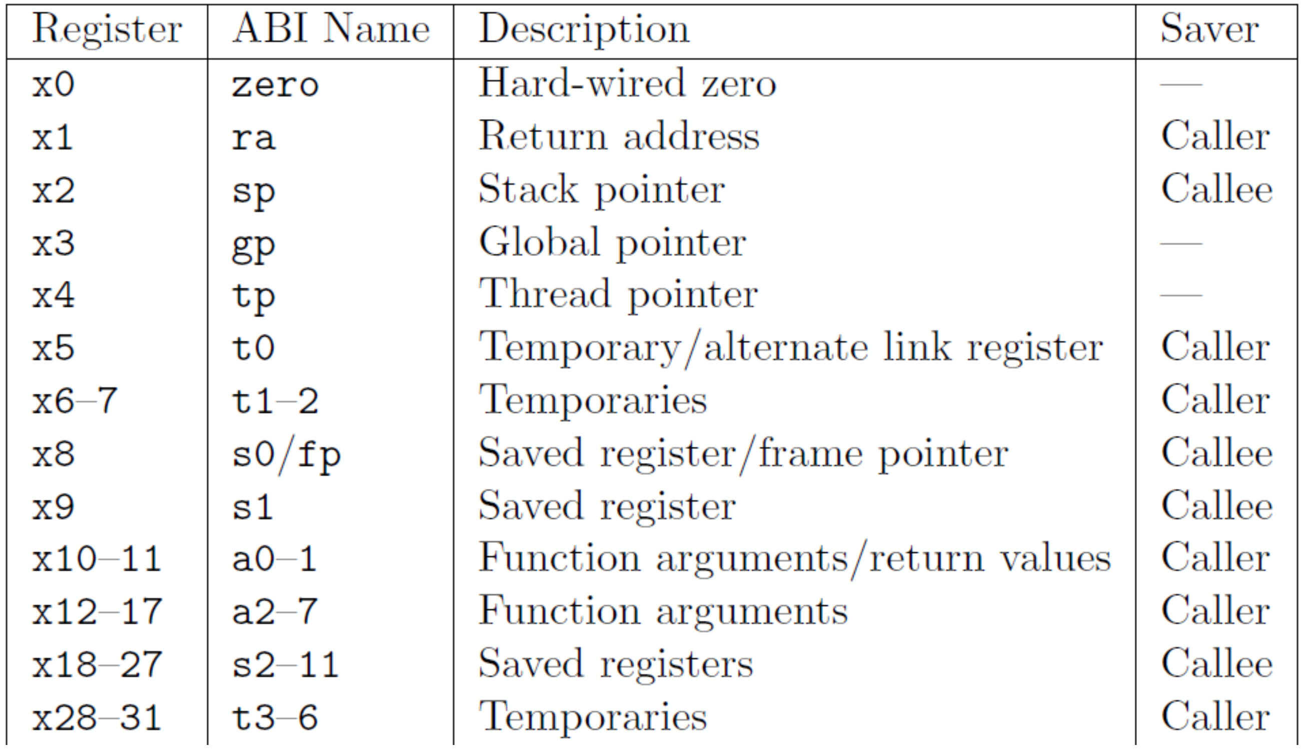
Calling Convention

- There are many different ways how to handle the stacking of registers when calling a subroutine

- There is a calling convention for each platform that defines the relationship between the caller (the part of the programming doing a call to a subroutine) and the callee (the subroutine that is called). It defines:
    - How arguments are passed between caller and callee
    - How values are returned from the callee to the caller
    - Who takes care of the stacking of which registers

RISC-V Registers

From the RISC-V Instruction Set Manual (riscv.org):



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Summary

- **Saved by Caller:**
  - ra (return address)
  - a0 - a1 (arguments/return values)
  - a2 – a7 (arguments)
  - t0 - t6 (temp. registers)
- **Saved by Callee:**
  - fp (frame pointer)
  - sp (stack pointer)
  - s1 – s11 (saved registers)

In this lecture we do not use gp and tp

The View of the Caller

Dear Callee,

Use these registers however you like – I do not care about the content. Your arguments are in a0 – a7. Give me your return value in a0 (32 bit case) or in a0 and a1 (64 bit value)

Dear Callee,

I want these registers back with exactly the same content as I passed them to you. In case you need them, these are registers are to be saved and restored by you.

Summary

- **Saved by Caller:**
  - ra (return address)
  - a0 - a1 (arguments/return values)
  - a2 – a7 (arguments)
  - t0 - t6 (temp. registers)
- **Saved by Callee:**
  - fp (frame pointer)
  - sp (stack pointer)
  - s1 – s11 (saved registers)

Switching from HW to SW View

- All subsequent assembler examples will be written using the software ABI conventions → we use no x.. registers any more
- In hardware this does not change anything – it is just the naming

Saved by Caller:
- ra (return address)
- a0 - a7 (arguments)
- t0 - t6 (temp. registers)

Saved by Callee:
- fp (frame pointer)
- sp (stack pointer)
- s1 – s11 (saved registers)

Code Parts of a Subroutine

- Important code parts for the handling of registers, local variables and arguments are
  - **Function Prolog** ("Set up") – the first instructions of a subroutine
  - **Neighborhood of a Nested Call** (before and after call)
  - **Epilog** ("Clean up") – the last instructions of a subroutine

Saved by Caller:
- ra (return address)
- a0 - a7 (arguments)
- t0 - t6 (temp. registers)

Saved by Callee:
- fp (frame pointer)
- sp (stack pointer)
- s1 – s11 (saved registers)

Examples

- Check the examples repo and look at the code in the directory **stack_according_to_abi**

- Compile and understand the following examples
  - **01_direct_return.asm**
  - **02_nested_function_call.asm**
  - **03_nested_call_with_argument.asm**
  - **04_recursive_call_with_arguments.asm**

Frame Pointer

- If there are too many arguments to fit them into the registers, the additional parameters are passed via the stack
- In order to facilitate the access to these arguments, we introduce the framepointer
- The framepointer stores the value of the stack pointer upon function entry
- This allows to have a fixed offset for variables throughout the function, e.g.
    - FP: points to the first argument on the stack
    - FP + 4: points to the second argument on the stack
    - FP - 4: points to the return address pushed by the callee (if needed)
- See example **05_call_with_many_arguments.asm**

Local Variables

- Whenever a function requires local variables, these variables are also stored on the stack
- See example **06_local_variables_and_call_by_reference.asm**

Call by Value vs. Call by Reference

- There are two important ways of passing arguments to a function

- **Call by Value**
  - The values of the arguments are provided in the registers a0-a7 and the stack

- **Call by Reference**
  - Instead of values, pointers are passed to the function (they point for example to variables of the stack frame of the caller)

  - See example **06_local_variables_and_call_by_reference.asm**

Full Stack Frame

- In case a function receives arguments via the stack, uses local variables and performs calls, the full stack frame looks as follows (addressed relative to the framepointer (fp)):
  - ….
  - FP + 8: third argument passed via stack
  - FP + 4: second argument passed via stack
  - FP: first argument
  - FP - 4: Return address
  - FP - 8: Frame pointer of caller
  - FP - 12: First local variable
  - FP - 16: Second local variable
  - …

Buffer Overflow

More on this?

Take the course "Secure Software Development"

- A computer performs one instruction after the other
- If return addresses on the stack are overwritten by user input, the computer will jump to a target defined by the user input
- Simple buffer overflows are detected on today's computer systems. However, there are many more options of how a user can attack a computer system.
- See example **07_stack_buffer_overflow.asm**

Summary on Code Parts of a Subroutine

- Prolog ("Set up") – the first instructions of a subroutine
  - Stacking the return address (in case needed)
  - Stacking of frame pointer of caller and initialization of FP for callee (in case needed)
  - Stacking of s1-s11 (in case these registers are needed)
  - Allocation of stack for local variables
- Neighborhood of a Nested Call (before and after call)
  - Preparation of arguments in registers and on stack (if needed) for the subroutine
  - Stacking and restoring of registers a0-a7, t0-t7 (in case these registers are still needed in the subroutine after returning from the call)
- Epilog ("Clean up") – the last instructions of a subroutine
  - Restore frame pointer
  - Restore return address
  - Restore stack pointer
  - Jump to return address

Saved by Caller:
- ra (return address)
- a0 - a7 (arguments)
- t0 - t6 (temp. registers)

Saved by Callee:
- fp (frame pointer)
- sp (stack pointer)
- s1 – s11 (saved registers)