Chapter 2 – Additional Notes

Immediate Operands

- Constant data specified in an instruction

  `addi x22, x22, 4`

- **Make the common case fast**
  - Small constants are common
  - Immediate operand avoids a load instruction

RISC-V R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

RV32I Architectural State

XLEN-1   0
x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15

XLEN-1   0
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

pc

RISC-V Register Usage Convention

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	—
x5–7	t0–2	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

[from page 100, The RISC-V Instruction Set Manual]

RV32I Arithmetic and Logical Operations

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

Load/Store Operations

- Both need two registers and a 12-bit immediate

imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD

Memory Operand Example

imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD

- C code:

  `A[12] = h + A[8];`

  - h in x21, base address of A in x22

- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

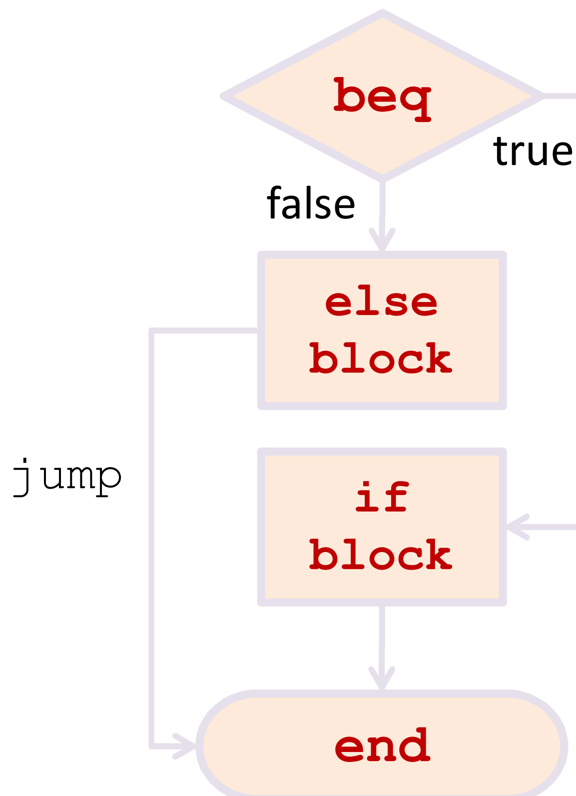
```
ld      x9, 64(x22) // load double word
add     x9, x21, x9
sd      x9, 96(x22)
```

Control flow in assembly

- Not all programs follow a linear set of instructions.
  - Some operations require the code to branch to one section of code or another (if/else).
  - Some require the code to jump back and repeat a section of code again (for/while).
- For this, we have labels on the left-hand side that indicate the points that the program flow might need to jump to.
  - References to these points in the assembly code are resolved at compile time to offset values for the program counter.

A trick with if statements

- Use flow charts to help you sort out the control flow of the code:

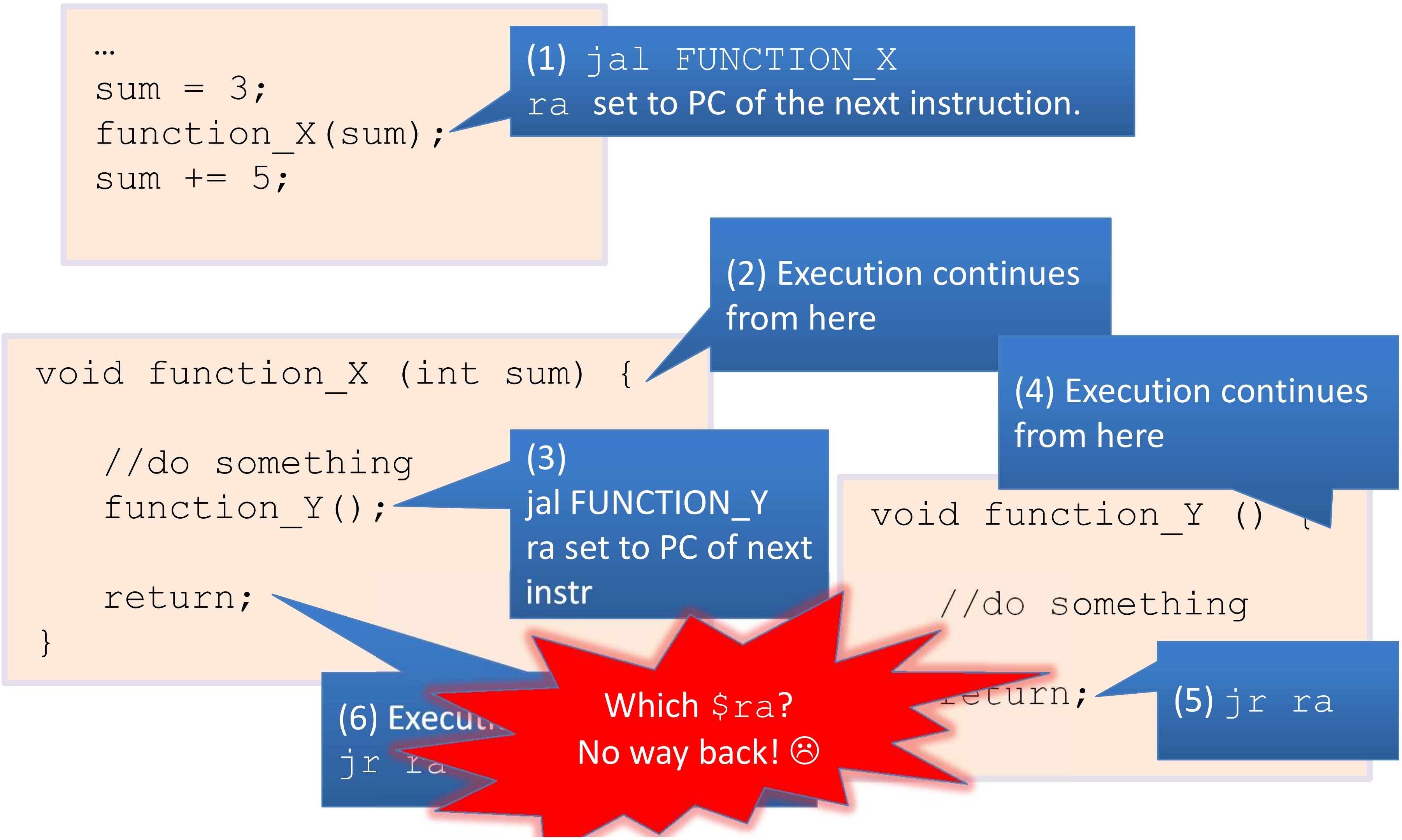


Lab D – C code

main.c

```c
/* C program to swap bytes/words of integer number.*/
#include <stdio.h>

int main()
{
    unsigned int data=0xAABB1234;
    printf("\ndata before swapping : %04X",data);

    data= ((data<<8)&0xff00)|((data>>8)&0x00ff);

    printf("\ndata after swapping  : %04X",data);

    return 0;
}

```

But what if I have nested function calls?



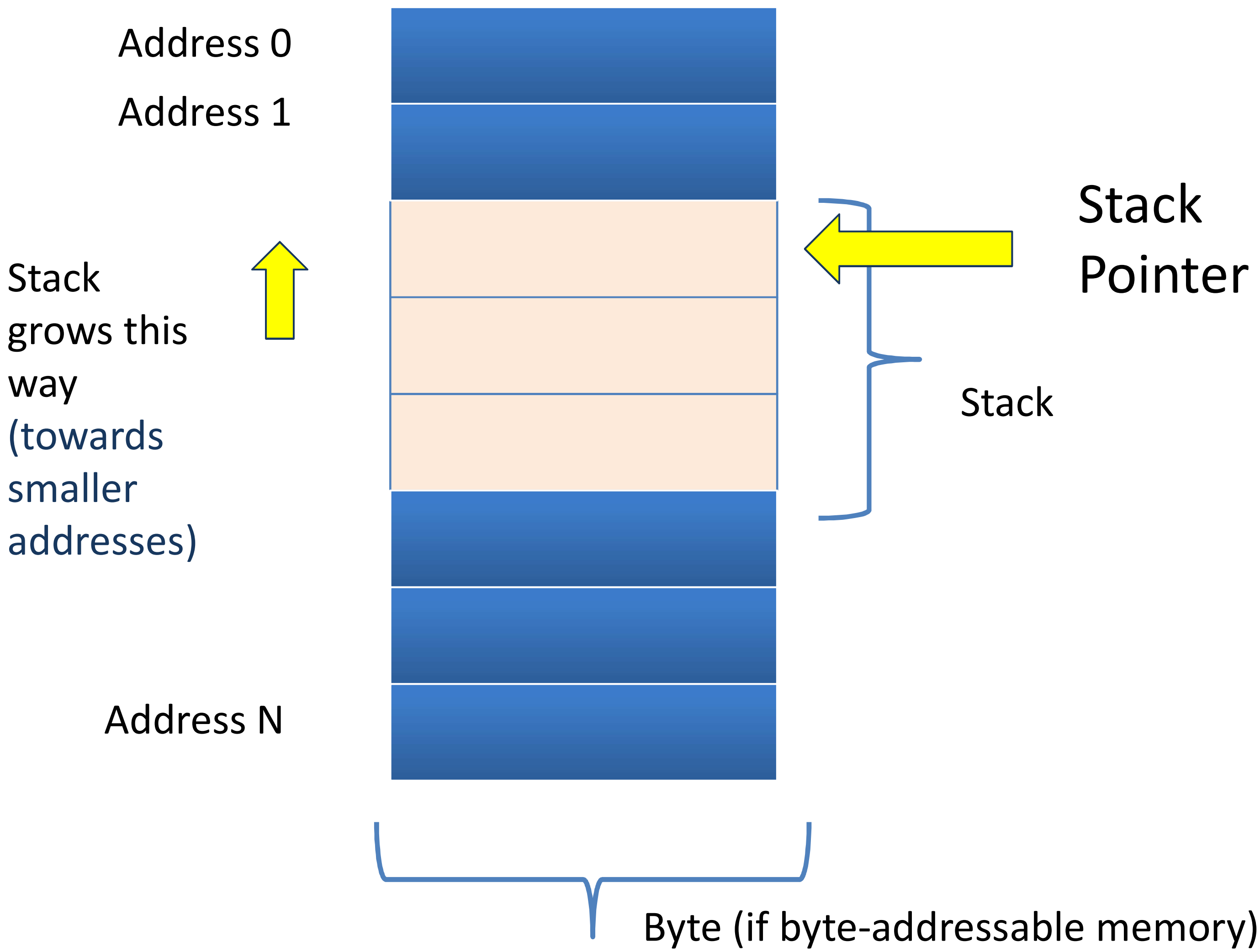
The stack to the rescue!

- Store `ra` in the stack.
  - Different versions of `ra` will exist in the stack
- We can also use the stack to store*:
  - Function arguments
  - Function return values
  - And also to maintain register values (more on this later).

* As mentioned before there are some predefined registers used for the function arguments and return values; the stack is used if this number is exceeded.

- E.g., if there are more than 4 arguments.

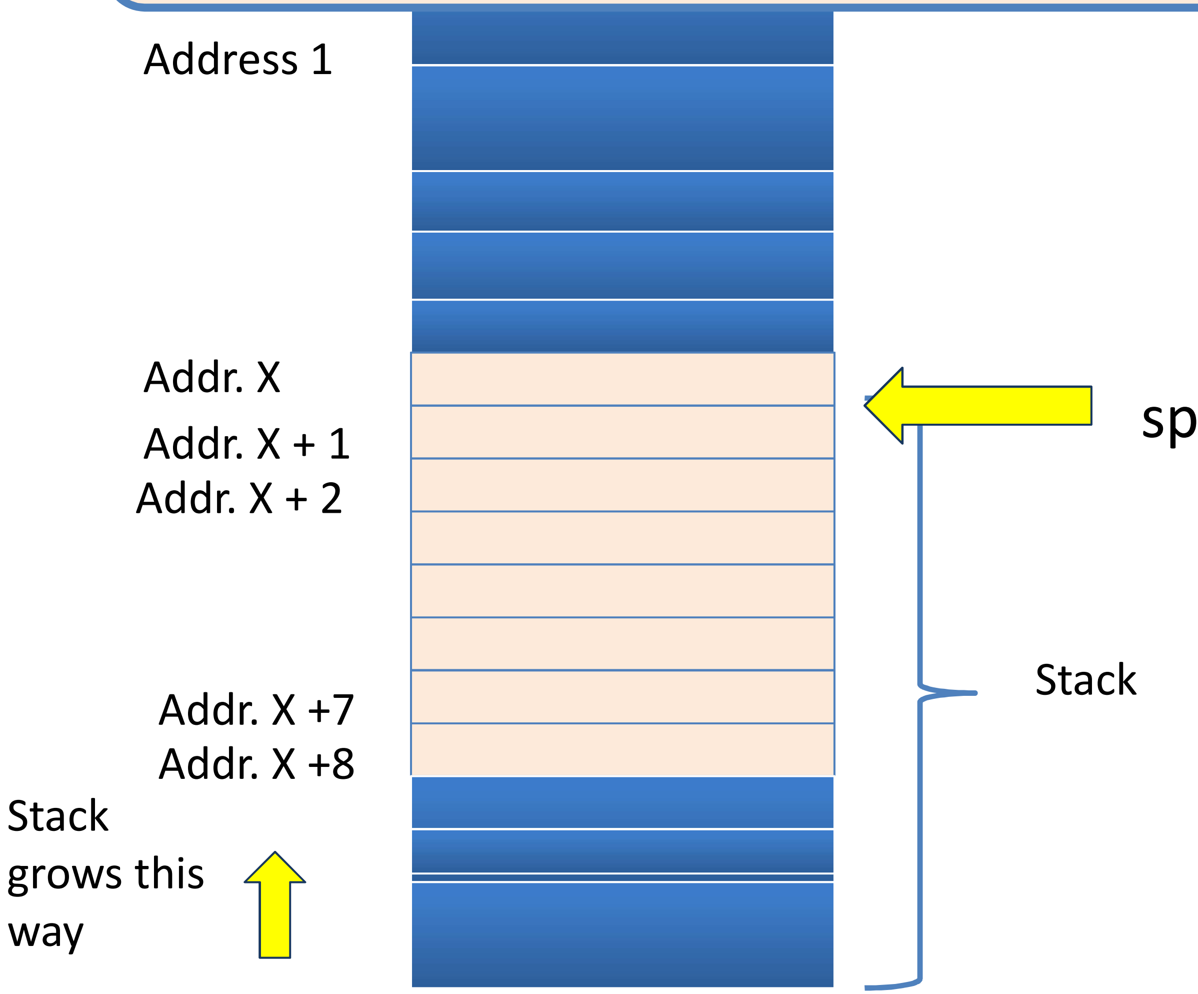
The Stack



Address 0
Address 1
Stack grows this way (towards smaller addresses)
Stack Pointer
Stack
Address N
Byte (if byte-addressable memory)

Pushing Values to the stack - After

```
addi   sp, sp, -8    # move stack pointer one word
sd     x5, 0(sp)     # push a word onto the stack
```




```
Just pushed
a 64-bit value
(8 bytes)
to the stack
```

Stack Usage

- Pushing something onto the stack
  - Allocate space by <u>decrementing</u> the stack pointer by the appropriate number of bytes.
  - Do a store (or multiple stores as needed).

- Popping something from the stack:
  - Do a load (or multiple loads as needed)
  - De-allocate space by incrementing the stack pointer by the appropriate number of bytes.

More advice on using the stack

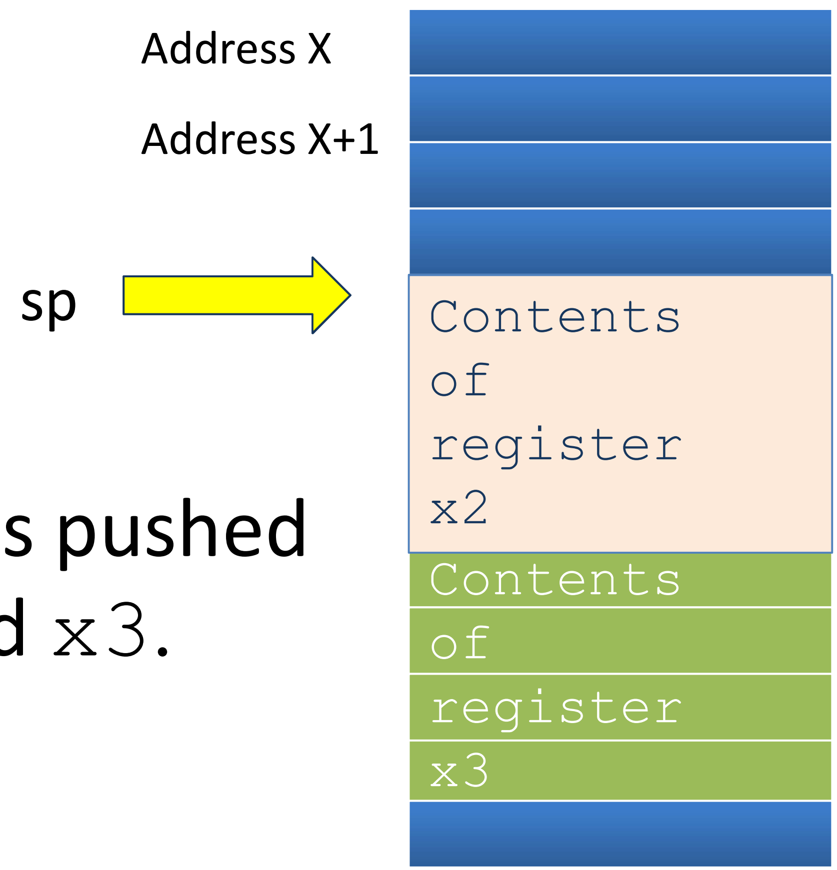
- Any space you allocate on the stack, you should later de-allocate.
- You should pop the items in the same order as you push them.
  - It might help to draw out an image of how your stack will look like.
- When pushing more than one item onto the stack, you can :
  - Either allocate all the space in the beginning
  - Or allocate space as you go.
  - Same for popping.

Let's do an example.

Figure shows stack *after* the push.

- Push contents of registers x2 and x3 onto the stack.

```
addi sp, sp, -16
sd x2, 0(sp)
sd x3, 8(sp)
```



- Restore stack values pushed to registers x2 and x3.

```
ld x2, 0(sp)
ld x3, 8(sp)
addi sp, sp, 16
```