



Computer Organisation & Program Execution 2021

Uwe R. Zimmer - The Australian National University

Computer Organisation & Program Execution 2021



Organization & Contents

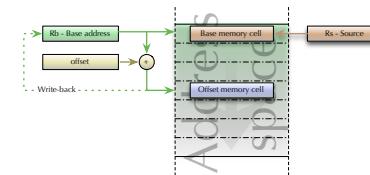
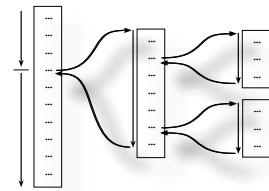
Uwe R. Zimmer - The Australian National University



Organization & Contents

what is offered here?

```
b      while_condition
while:
    mul   r1, r1
while_condition:
    cmp   r1, #100
    blt   while
```



Fundamentals, Overview & Hands-on Experience

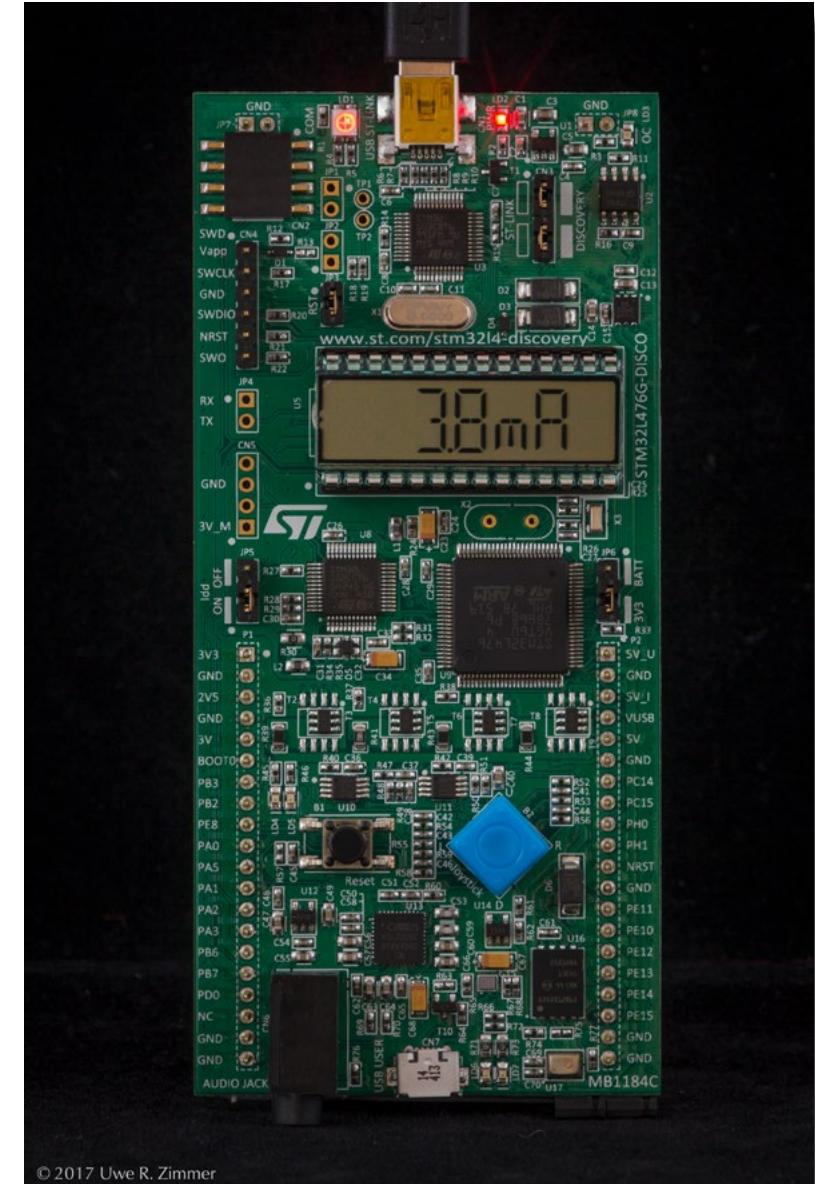
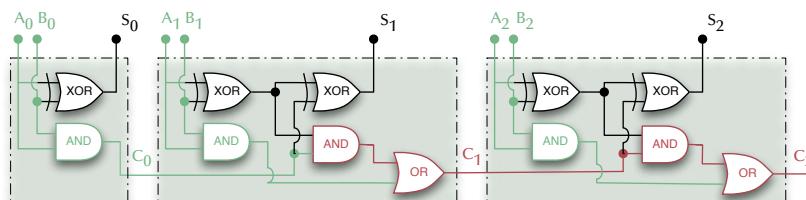
of

Computer Architecture

ADDs <Rd>, <Rn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rm	Rn	Rd						

Op Code Arguments



© 2017 Uwe R. Zimmer



Organization & Contents

who could be interested in this?

anybody who ...

... wants to know why and how computer science
immediately connects and translates to the physical world.

... would like to see **immediate real-world involvement** in their work.

... would like to understand what
really happens if you run a high level program.



Organization & Contents

who are these people? – introductions



Ben



Uwe

Ben Swift & Uwe R. Zimmer

Abigail (Abi) Thomas, Ashleigh Johannes, Ben Gray, Brent Schuetze,

Calum Snowdon, Chinmay Garg, Harrison Shoebridge,

Johannes (Johnny) Schmalz, Peter Baker, Ryan Stocks, Septian Razi,

Tom Willingham



Abi



Ashleigh



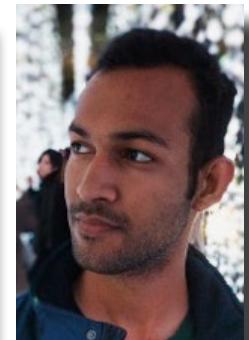
Ben



Brent



Calum



Chinmay



Peter



Ryan



Razi



Harrison



Organization & Contents

how will this all be done?

👉 Lectures:

- 2x 1.5 hours lectures per week ... all the nice stuff
Monday 13:30, Wednesday 11:30 (both on-line - which is: *here*)

👉 Laboratories:

- 3 hours per week ... all the rough stuff
time slots: on our web-site – on-campus in CSIT N.xxx or HN Lab.xx laboratories
-enrolment: <https://cs.anu.edu.au/streams/> (opened on Monday)

👉 Resources:

- Course site: <http://cs.anu.edu.au/student/comp2300/> ... as well as schedules, slides, sources, links to forums, etc. pp. ... keep an eye on this page!

👉 Assessment:

- *Hurdle lab* in week 4 (1%) – a pass here is a **hurdle** for the course
- *Mid-semester exam* (13%)
- *3 assignments* (12% each)
- *Final-exam* at the end of the course (50%) – 40/100 is a **hurdle** for the final exam



Organization & Contents

“Text book” for the course

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

ARM edition, Morgan Kaufmann 2017

☞ Many concepts in this course are in there – ***but not all!***

The [Patterson17] provides an excellent general background and a lot of in-depth studies into more specific fields.

☞ References for specific aspects of the course are provided during the course and are found on our web-site.

Computer Organisation & Program Execution 2021



1

Digital Logic

Uwe R. Zimmer - The Australian National University



Digital Logic

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

Appendix A “The Basics of Logic Design”

ARM edition, Morgan Kaufmann 2017

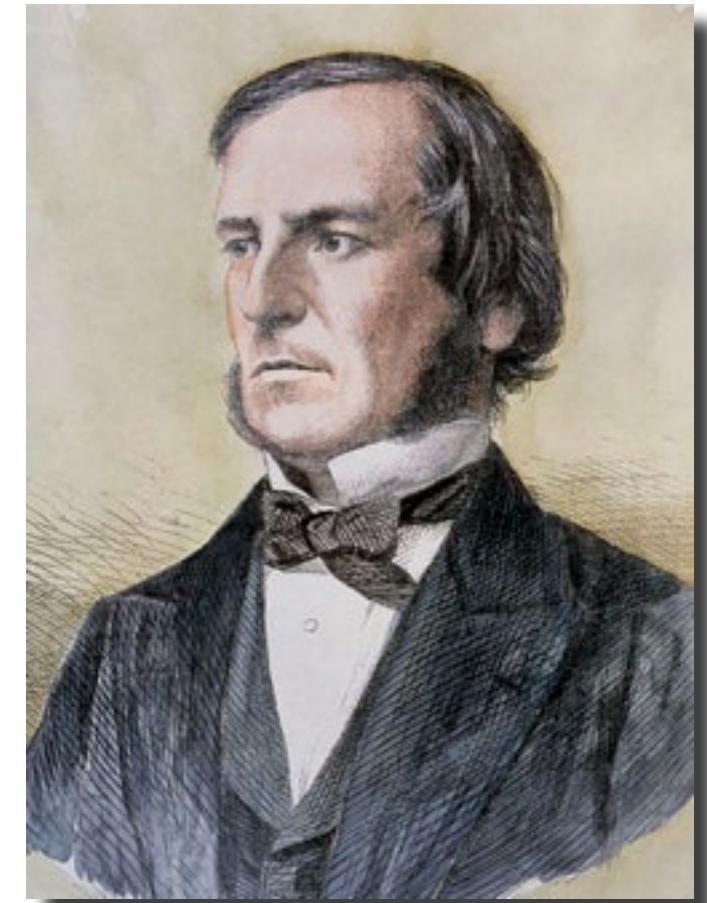


Digital Logic

It starts with a thought ...

An Investigation of the Laws of Thought
on Which are Founded the Mathematical
Theories of Logic and Probabilities

by **George Boole**, 1854



George Boole, 1815-1864



Digital Logic

Boolean Values & Operators

There are two *values*:

e.g. **True** and **False**. (aka “1” and “0”)

Two binary *operators* on expressions a, b :

$a \vee b$ (aka $a + b$ or “ a OR b ” or SUM)

$a \wedge b$ (aka $a \cdot b$ or “ a AND b ” or PRODUCT)

One unary *operator* on an expression a :

\bar{a} (aka $\neg a$ or a' or “NOT a ”)

Truth tables:

a	b	$a \vee b$	$a \wedge b$	\bar{a}
False	False	False	False	True
True	False	True	False	False
False	True	True	False	
True	True	True	True	



Digital Logic

Axiomatic Boolean Algebra (Whitehead 1898)

\vee -Laws

$$a \vee a = a$$

$$a \vee b = b \vee a$$

$$(a \vee b) \vee c = a \vee (b \vee c)$$

$$a \vee (a \wedge b) = a$$

$$a \vee \bar{a} = \text{True}$$

\wedge -Laws

$$a \wedge a = a$$

$$a \wedge b = b \wedge a$$

$$(a \wedge b) \wedge c = a \wedge (b \wedge c)$$

(redundant)

(commutative)

(associative)

(absorption)

(distribution)

(identity)

(constant)

$$a \wedge \text{True} = a$$

(inverse)

DeMorgan

(double not)

Algebras allow for easier reasoning than truth tables.



Digital Logic

Axiomatic Boolean Algebra (Huntington 1904)

\vee -Laws

\wedge -Laws

		(redundant)
$a \vee b = b \vee a$	$a \wedge b = b \wedge a$	(commutative)
		(associative)
		(absorption)
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	(distribution)
$a \vee \text{False} = a$	$a \wedge \text{True} = a$	(identity)
		(constant)
$a \vee \bar{a} = \text{True}$	$a \wedge \bar{a} = \text{False}$	(inverse)
		DeMorgan
		(double not)



Digital Logic

Axiomatic Boolean Algebra

... many other axiomatic formulations of Boolean algebra exist.



Digital Logic

Redundant Boolean Algebra

\vee -Laws

$$a \vee a = a$$

$$a \vee b = b \vee a$$

$$(a \vee b) \vee c = a \vee (b \vee c)$$

$$a \vee (a \wedge b) = a$$

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

$$a \vee \text{False} = a$$

$$a \vee \text{True} = \text{True}$$

$$a \vee \bar{a} = \text{True}$$

$$\overline{a \vee b} = \bar{a} \wedge \bar{b}$$

\wedge -Laws

$$a \wedge a = a$$

$$a \wedge b = b \wedge a$$

$$(a \wedge b) \wedge c = a \wedge (b \wedge c)$$

$$a \wedge (a \vee b) = a$$

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$

$$a \wedge \text{True} = a$$

$$a \wedge \text{False} = \text{False}$$

$$a \wedge \bar{a} = \text{False}$$

$$\overline{a \wedge b} = \bar{a} \vee \bar{b}$$

$$\overline{\bar{a}} = a$$

(double not)

... second nature for a computer scientist!

(redundant)

(commutative)

(associative)

(absorption)

(distribution)

(identity)

(constant)

(inverse)

DeMorgan



Digital Logic

Common Boolean operators

Commonly used *operators* on expressions a, b to define boolean algebras:

$a \vee b$	(aka $a + b$ or "a OR b" or SUM)
$a \wedge b$	(aka $a \cdot b$ or "a AND b" or PRODUCT)
\bar{a}	(aka $\neg a$ or a' or "not a")

Other handy operators:

$a \rightarrow b = (\bar{a} \vee b)$	(aka "a IMPLIES b")
$(a = b) = (a \wedge b) \vee (\bar{a} \wedge \bar{b})$	(aka "a EQUALS b")
$a \oplus b = (a \wedge \bar{b}) \vee (\bar{a} \wedge b)$	(aka "a EXCLUSIVE-OR b" or "a XOR b")
$\overline{a \wedge b} = (\bar{a} \vee \bar{b})$	(aka "a NOT-AND b" or "a NAND b")
$\overline{a \vee b} = (\bar{a} \wedge \bar{b})$	(aka "a NOT-OR b" or "a NOR b")

NAND and NOR are the only sole sufficient boolean operators,
i.e. you can reduce any boolean expression to only NAND or only NOR operators.



Digital Logic

All binary Boolean operators

	Inputs a, b		Function	Name	Sum of products	NAND build	Don't cares
a	F	F	T	T			
b	F	T	F	T			
q	F	F	F	F	False	Constant FALSE	
	F	F	F	T	$a \wedge b$	AND	$\overline{\overline{a \wedge b} \wedge \overline{a \wedge b}}$
	F	F	T	F	$\overline{a \rightarrow b}$	NOT-IMPLICATION	$(a \wedge \overline{b})$
	F	F	T	T	\overline{a}	IDENTITY a	
	F	T	F	F	$\overline{b \rightarrow a}$	NOT-IMPLICATION	$(\overline{a} \wedge b)$
	F	T	F	T	\overline{b}	IDENTITY b	
	F	T	T	F	$a \oplus b$	EXCLUSIVE-OR, XOR	$(a \wedge \overline{b}) \vee (\overline{a} \wedge b)$
	F	T	T	T	$a \vee b$	OR	$\overline{\overline{a \wedge a} \wedge \overline{b \wedge b}}$
	T	F	F	F	$\overline{a \vee b}$	NOT-OR, NOR	$(\overline{a} \wedge \overline{b})$
	T	F	F	T	$a = b$	EQUALITY, EQ	$(a \wedge b) \vee (\overline{a} \wedge \overline{b})$
	T	F	T	F	\overline{b}	INVERSE b	
	T	F	T	T	$b \rightarrow a$	IMPLICATION	$a \vee \overline{b}$
	T	T	F	F	\overline{a}	INVERSE a	
	T	T	F	T	$a \rightarrow b$	IMPLICATION	$\overline{a} \vee b$
	T	T	T	F	$\overline{a \wedge b}$	NOT-AND, NAND	$\overline{a} \vee \overline{b}$
	T	T	T	T	True	Constant True	
Output q							



Digital Logic

Combinational Logic Functions

☞ Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q
F	F	F	F
F	F	T	F
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	T
T	T	F	T
T	T	T	F



Digital Logic

Combinational Logic Functions

☞ Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q	minterms
F	F	F	F	
F	F	T	F	
F	T	F	T	$\bar{a} \wedge b \wedge \bar{c}$
F	T	T	F	
T	F	F	T	$a \wedge \bar{b} \wedge \bar{c}$
T	F	T	T	$a \wedge \bar{b} \wedge c$
T	T	F	T	$a \wedge b \wedge \bar{c}$
T	T	T	F	

Sum of minterms: $q = (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$



Digital Logic

Combinational Logic Functions

☞ Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q	minterms	Simplified minterms
F	F	F	F		
F	F	T	F		
F	T	F	T	$\bar{a} \wedge b \wedge \bar{c}$	
F	T	T	F		
T	F	F	T	$a \wedge \bar{b} \wedge \bar{c}$	
T	F	T	T	$a \wedge \bar{b} \wedge c$	$a \wedge \bar{b}$
T	T	F	T	$a \wedge b \wedge \bar{c}$	
T	T	T	F		

Sum of minterms: $q = (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$

Sum of simplified minterms: $q = (a \wedge \bar{b}) \vee (b \wedge \bar{c})$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others



Digital Logic

Combinational Logic Functions

☞ Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q	minterms	Simplified minterms
F	F	F	F		
F	F	T	F		
F	T	F	T	$\bar{a} \wedge b \wedge \bar{c}$	
F	T	T	F		
T	F	F	T	$a \wedge \bar{b} \wedge \bar{c}$	
T	F	T	T	$a \wedge \bar{b} \wedge c$	$a \wedge \bar{b}$
T	T	F	T	$a \wedge b \wedge \bar{c}$	
T	T	T	F		

Every combinational function can
be written as a **sum of products**!

$$\text{Sum of minterms: } q = (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$$

$$\text{Sum of simplified minterms: } q = (a \wedge \bar{b}) \vee (b \wedge \bar{c})$$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others



Digital Logic

Combinational Logic Functions

☞ Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q
F	F	F	F
F	F	T	F
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	T
T	T	F	T
T	T	T	F



Digital Logic

Combinational Logic Functions

☞ Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q	maxterms
F	F	F	F	$a \vee b \vee c$
F	F	T	F	$a \vee b \vee \bar{c}$
F	T	F	T	
F	T	T	F	$a \vee \bar{b} \vee \bar{c}$
T	F	F	T	
T	F	T	T	
T	T	F	T	
T	T	T	F	$\bar{a} \vee \bar{b} \vee \bar{c}$

$$\text{maxterms product } q = (a \vee b \vee c) \wedge (a \vee b \vee \bar{c}) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee \bar{c})$$



Digital Logic

Combinational Logic Functions

☞ Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q	maxterms	Simplified maxterms
F	F	F	F	$a \vee b \vee c$	
F	F	T	F	$a \vee b \vee \bar{c}$	$a \vee b$
F	T	F	T		
F	T	T	F	$a \vee \bar{b} \vee \bar{c}$	
T	F	F	T		
T	F	T	T		$\bar{b} \vee \bar{c}$
T	T	F	T		
T	T	T	F	$\bar{a} \vee \bar{b} \vee \bar{c}$	

$$\text{maxterms product } q = (a \vee b \vee c) \wedge (a \vee b \vee \bar{c}) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee \bar{c})$$
$$\text{simplified maxterms product } q = (a \vee b) \wedge (\bar{b} \vee \bar{c})$$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others



Digital Logic

Combinational Logic Functions

☞ Logic is reducible/equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q	maxterms	Simplified maxterms
F	F	F	F	$a \vee b \vee c$	
F	F	T	F	$a \vee b \vee \bar{c}$	
F	T	F	T		
F	T	T	F	$a \vee \bar{b} \vee \bar{c}$	
T	F	F	T		
T	F	T	T		
T	T	F	T		
T	T	T	F	$\bar{a} \vee \bar{b} \vee \bar{c}$	

Every combinational function can
be written as a **product of sums**!

$$\text{maxterms product } q = (a \vee b \vee c) \wedge (a \vee b \vee \bar{c}) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee \bar{c})$$
$$\text{simplified maxterms product } q = (a \vee b) \wedge (\bar{b} \vee \bar{c})$$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others

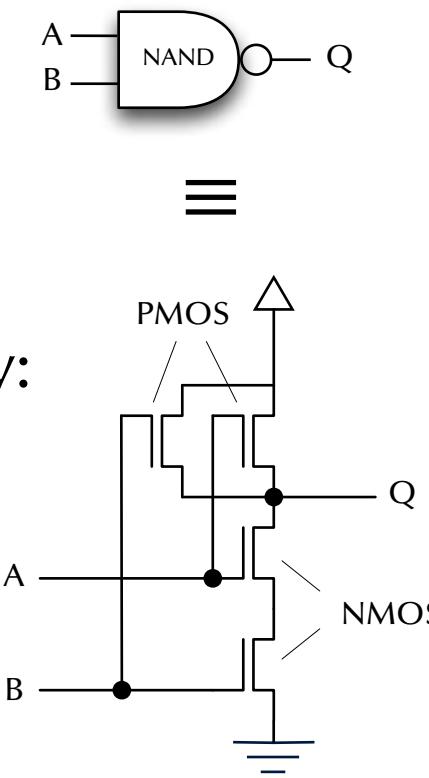


Digital Logic

Digital Electronics

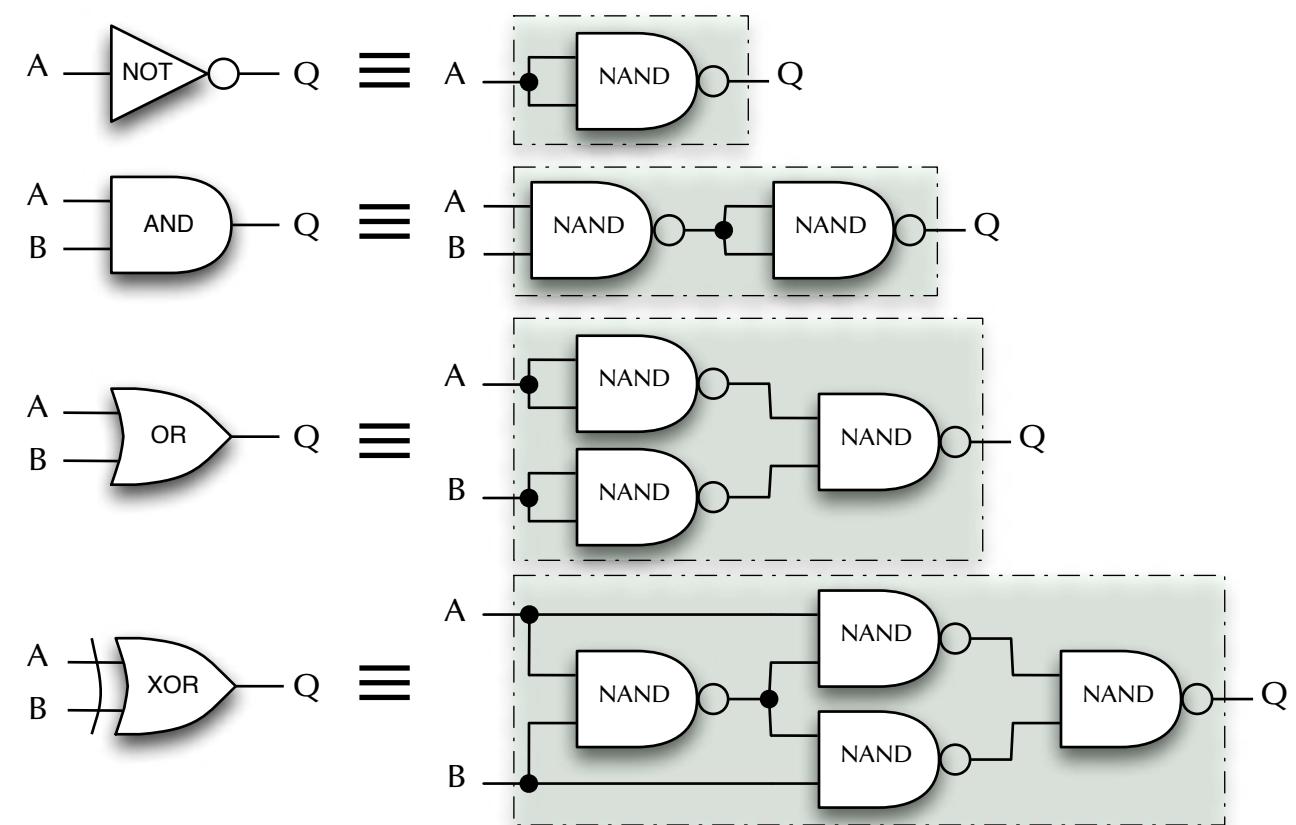
Symbolic: $Q = \overline{A} \wedge \overline{B}$

Diagram:



Technology:

Elementary logic gate symbols:





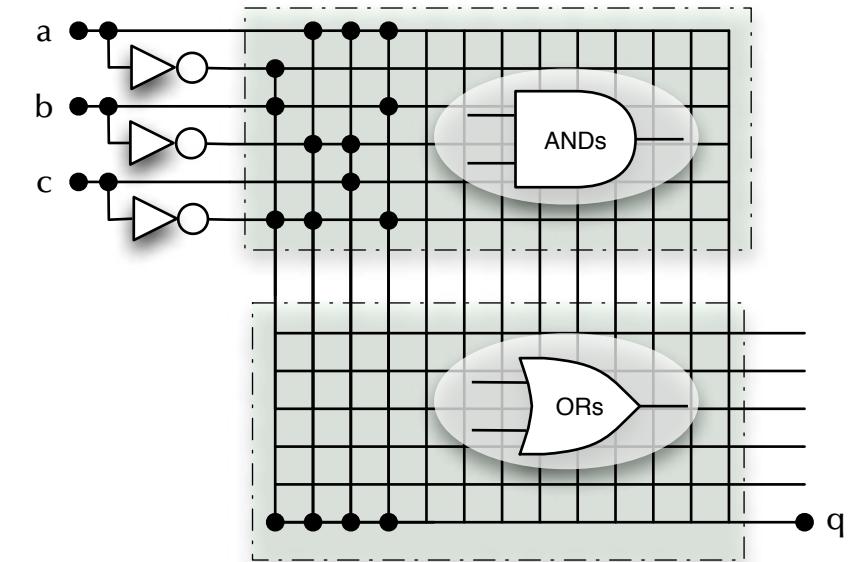
Digital Logic

Combinational Logic Functions

☞ The logic equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

a	b	c	Output q	Minterms	Simplified minterms
F	F	F	F		
F	F	T	F		
F	T	F	T	$\bar{a} \wedge b \wedge \bar{c}$	
F	T	T	F		
T	F	F	T	$a \wedge \bar{b} \wedge \bar{c}$	
T	F	T	T	$a \wedge \bar{b} \wedge c$	$a \wedge \bar{b}$
T	T	F	T	$a \wedge b \wedge \bar{c}$	
T	T	T	F		



$$\text{Sum of minterms: } q = (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$$

$$\text{Sum of simplified minterms: } q = (a \wedge \bar{b}) \vee (b \wedge \bar{c})$$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others



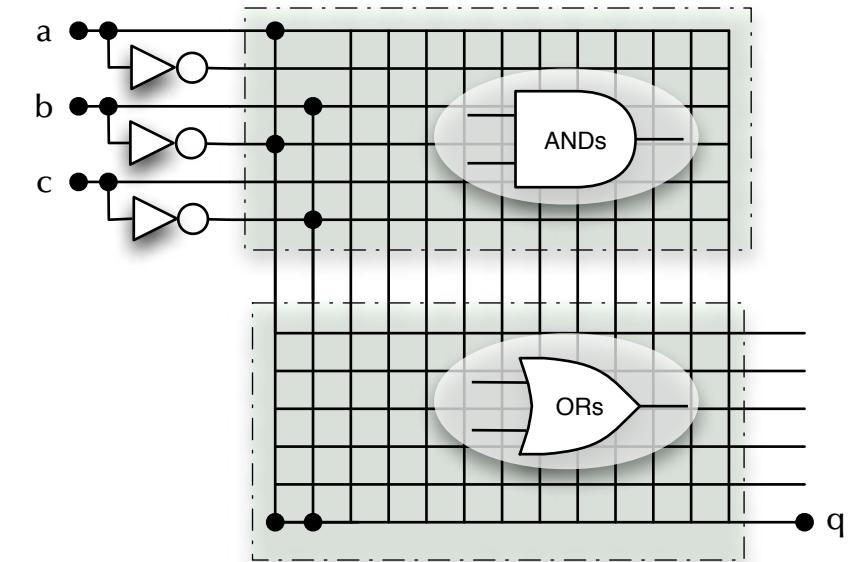
Digital Logic

Combinational Logic Functions

☞ The logic equivalent to pure functions: there are no states!

IF the function is combinational then there is only one output for any combination of inputs, e.g. the function can be written out as a truth table:

a	b	c	Output q	Minterms	Simplified minterms
F	F	F	F		
F	F	T	F		
F	T	F	T	$\bar{a} \wedge b \wedge \bar{c}$	
F	T	T	F		
T	F	F	T	$a \wedge \bar{b} \wedge \bar{c}$	
T	F	T	T	$a \wedge \bar{b} \wedge c$	$a \wedge \bar{b}$
T	T	F	T	$a \wedge b \wedge \bar{c}$	
T	T	T	F		



$$\text{Sum of minterms: } q = (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$$

$$\text{Sum of simplified minterms: } q = (a \wedge \bar{b}) \vee (b \wedge \bar{c})$$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others



Digital Logic

Combinational Logic Functions

The logic equivalent to pure functions: there are no states!

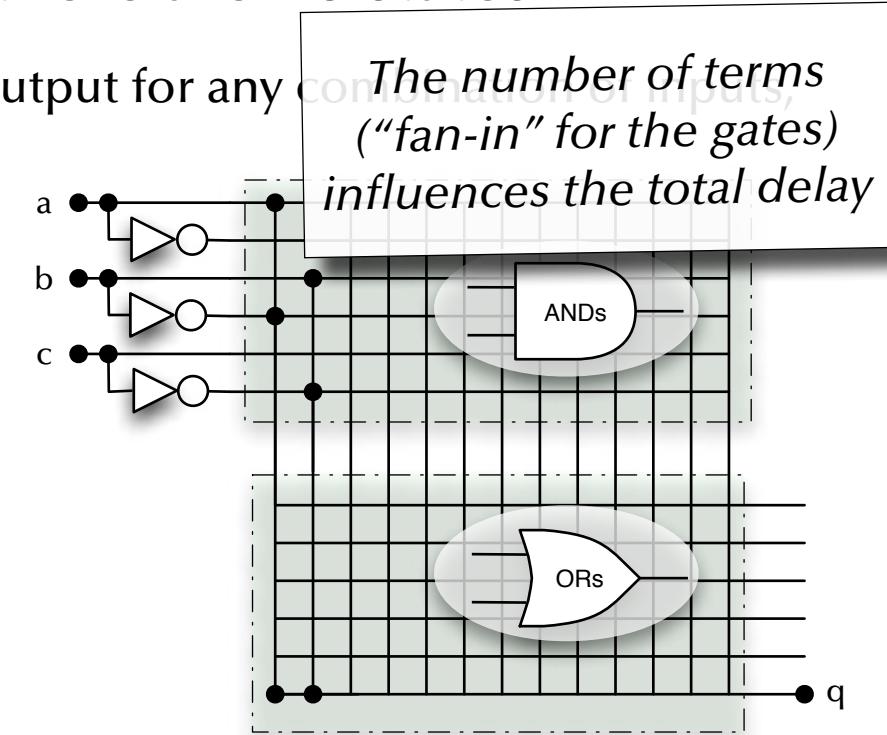
IF the function is combinational then there is only one output for any combination of inputs,
e.g. the function can be written out as a truth table:

a	b	c	Output q	Minterms	Simplified minterms
F	F	F	F		
F	F	T	F		
F	T	F	T	$\bar{a} \wedge b \wedge \bar{c}$	
F	T	T	F		
T	F	F	T	$a \wedge \bar{b} \wedge \bar{c}$	
T	F	T	T	$a \wedge \bar{b} \wedge c$	$a \wedge \bar{b}$
T	T	F	T	$a \wedge b \wedge \bar{c}$	
T	T	T	F		

$$\text{Sum of minterms: } q = (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$$

$$\text{Sum of simplified minterms: } q = (a \wedge \bar{b}) \vee (b \wedge \bar{c})$$

Simplifications can be done by (automated) algebraic transformations, Karnaugh maps or others





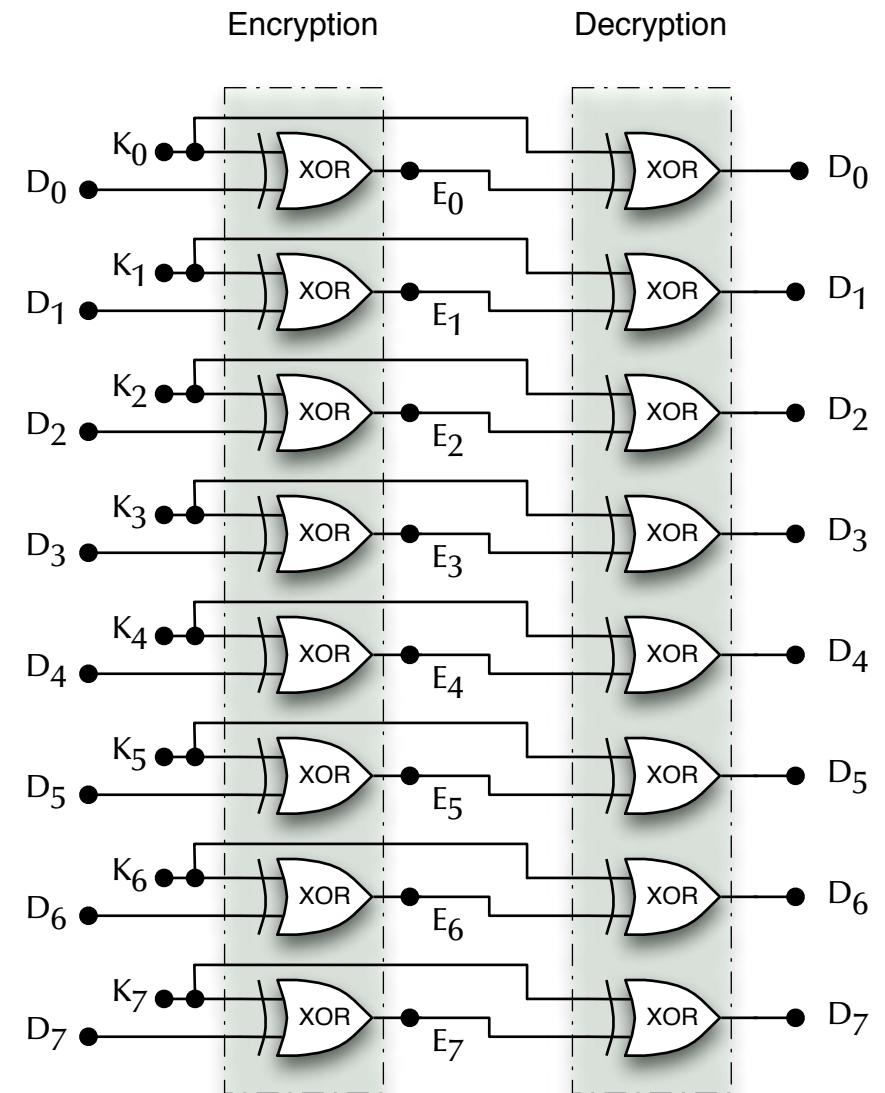
Digital Logic

Processing Data

Encrypting a bit vector
(whatever it represents)
with a secret key:

Assuming the key is random
and not used for anything else:

- ☞ This is surprisingly secure
- ☞ ... and extremely fast!





Digital Logic

Bit Vectors

Groups of bits could represent:

States, enumeration values, arrays of Booleans, numbers, etc. pp.
... or any grouping or combination of the above ↗ **Algebraic Types**

The form of encoding could be chosen to optimize for:

- **Performance** ↗ e.g. minimal decoding effort
- **Redundancy / error detection** ↗ e.g. large Hamming distance
- **Safe transitions** ↗ e.g. Gray codes
- **Physical mapping** ↗ e.g. maps on existing hardware interfaces
- **Compactness** ↗ e.g. holds the maximal number of values per memory cell



Digital Logic

Encoding

Assuming a type can have 7 different values, many forms of encoding are possible:

Enumeration type		Encoding						
Index	Value	Single bit	Gray code	1-bit error detecting	Even parity	Hamming (7,4)	Hamming (3,1)	Binary
1	Secured	0000001	000	0000	0000000	000000000	000	
2	Taxi	0000010	001	0011	0011	1110000	000000111	001
3	Take-off	0000100	011	0101	0101	1001100	000111000	010
4	Cruising	0001000	010	0110	0110	0111100	000111111	011
5	Gliding	0010000	110	1001	1001	0101010	111000000	100
6	Approach	0100000	111	1010	1010	1011010	111000111	101
7	Landing	1000000	101	1100	1100	1100110	111111000	110

☞ VHDL or Verilog gives you full control over the encoding.



Digital Logic

Binary encoding

- ☞ Encoding of choice if compactness is essential or you need to add values.

$$\begin{array}{cccccccc} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ \downarrow & & \downarrow & & \downarrow & & & \\ *2^7 & *2^6 & *2^5 & *2^4 & *2^3 & *2^2 & *2^1 & *2^0 \\ \downarrow & & \downarrow & & \downarrow & & & \\ 32 & + & 8 & + & 2 & & & = 42 \end{array}$$



Digital Logic

Binary encoding

- Encoding of choice if compactness is essential or you need to add values.

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

$$\begin{array}{ccccccc} \downarrow & \downarrow & \downarrow \\ *2^7 & *2^6 & *2^5 & *2^4 & *2^3 & *2^2 & *2^1 & *2^0 \\ \downarrow & \downarrow & \downarrow \\ 32 & + & 8 & + & 2 & & = 42 \end{array}$$

Decimal	Binary	Hexadecimal
0 =	0 0 0 0	= 0
1 =	0 0 0 1	= 1
2 =	0 0 1 0	= 2
3 =	0 0 1 1	= 3
4 =	0 1 0 0	= 4
5 =	0 1 0 1	= 5
6 =	0 1 1 0	= 6
7 =	0 1 1 1	= 7
8 =	1 0 0 0	= 8
9 =	1 0 0 1	= 9
10 =	1 0 1 0	= A
11 =	1 0 1 1	= B
12 =	1 1 0 0	= C
13 =	1 1 0 1	= D
14 =	1 1 1 0	= E
15 =	1 1 1 1	= F



Digital Logic

Binary encoding

☞ Encoding of choice if compactness is essential or you need to add values.

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

$\downarrow \quad \downarrow \quad \downarrow$

$*2^7 \quad *2^6 \quad *2^5 \quad *2^4 \quad *2^3 \quad *2^2 \quad *2^1 \quad *2^0$

$\downarrow \quad \downarrow \quad \downarrow$

32 + 8 + 2 = 42

2	A
---	---

$\downarrow \quad \downarrow$

$*16^1 \quad *16^0$

$\downarrow \quad \downarrow$

32 + 10 = 42

Decimal	Binary	Hexadecimal
0 =	0 0 0 0	= 0
1 =	0 0 0 1	= 1
2 =	0 0 1 0	= 2
3 =	0 0 1 1	= 3
4 =	0 1 0 0	= 4
5 =	0 1 0 1	= 5
6 =	0 1 1 0	= 6
7 =	0 1 1 1	= 7
8 =	1 0 0 0	= 8
9 =	1 0 0 1	= 9
10 =	1 0 1 0	= A
11 =	1 0 1 1	= B
12 =	1 1 0 0	= C
13 =	1 1 0 1	= D
14 =	1 1 1 0	= E
15 =	1 1 1 1	= F



Digital Logic

Half Adder

A	B	S	C	S minterms	C minterms
0	0	0	0		
0	1	1	0	$\bar{A} \wedge B$	
1	0	1	0	$A \wedge \bar{B}$	
1	1	0	1		$A \wedge B$



Digital Logic

Half Adder

A	B	S	C	S minterms	C minterms
0	0	0	0		
0	1	1	0	$\overline{A} \wedge B$	
1	0	1	0	$A \wedge \overline{B}$	
1	1	0	1		$A \wedge B$

☞ $S = (A \wedge \overline{B}) \vee (\overline{A} \wedge B)$

☞ $C = A \wedge B$



Digital Logic

Half Adder

A	B	S	C	S minterms	C minterms
0	0	0	0		
0	1	1	0	$\overline{A} \wedge B$	
1	0	1	0	$A \wedge \overline{B}$	
1	1	0	1		$A \wedge B$

☞ $S = (A \wedge \overline{B}) \vee (\overline{A} \wedge B) = A \oplus B$

☞ $C = A \wedge B$



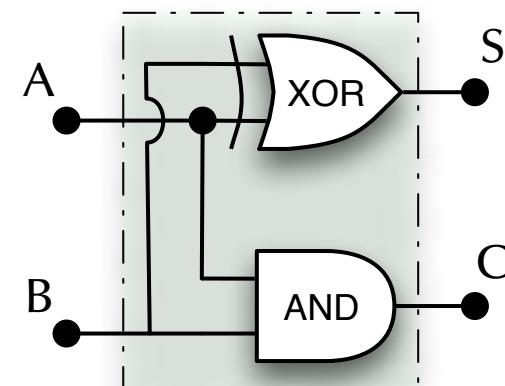
Digital Logic

Half Adder

A	B	S	C	S minterms	C minterms
0	0	0	0		
0	1	1	0	$\bar{A} \wedge B$	
1	0	1	0	$A \wedge \bar{B}$	
1	1	0	1		$A \wedge B$

☞ $S = (A \wedge \bar{B}) \vee (\bar{A} \wedge B) = A \oplus B$

☞ $C = A \wedge B$





Digital Logic

Full Adder

A_i	B_i	C_{i-1}	S_i	C_i	
0	0	0	0	0	
0	1	0	1	0	
1	0	0	1	0	
1	1	0	0	1	
0	0	1	1	0	
0	1	1	0	1	
1	0	1	0	1	
1	1	1	1	1	



Digital Logic

Full Adder

A_i	B_i	C_{i-1}	S_i	C_i	S_i minterms	C_i minterms
0	0	0	0	0		
0	1	0	1	0	$\overline{A}_i \wedge B_i \wedge \overline{C}_{i-1}$	
1	0	0	1	0	$A_i \wedge \overline{B}_i \wedge \overline{C}_{i-1}$	
1	1	0	0	1		$A_i \wedge B_i \wedge \overline{C}_{i-1}$
0	0	1	1	0	$\overline{A}_i \wedge \overline{B}_i \wedge C_{i-1}$	
0	1	1	0	1		$\overline{A}_i \wedge B_i \wedge C_{i-1}$
1	0	1	0	1		$A_i \wedge \overline{B}_i \wedge C_{i-1}$
1	1	1	1	1	$A_i \wedge B_i \wedge C_{i-1}$	$A_i \wedge B_i \wedge C_{i-1}$

☞ $S_i = (A_i \wedge \overline{B}_i \wedge \overline{C}_{i-1}) \vee (\overline{A}_i \wedge B_i \wedge \overline{C}_{i-1}) \vee (\overline{A}_i \wedge \overline{B}_i \wedge C_{i-1}) \vee (A_i \wedge B_i \wedge C_{i-1})$



Digital Logic

Full Adder

A_i	B_i	C_{i-1}	S_i	C_i	S_i minterms	C_i minterms
0	0	0	0	0		
0	1	0	1	0	$\overline{A}_i \wedge B_i \wedge \overline{C}_{i-1}$	
1	0	0	1	0	$A_i \wedge \overline{B}_i \wedge \overline{C}_{i-1}$	
1	1	0	0	1		$A_i \wedge B_i \wedge \overline{C}_{i-1}$
0	0	1	1	0	$\overline{A}_i \wedge \overline{B}_i \wedge C_{i-1}$	
0	1	1	0	1		$\overline{A}_i \wedge B_i \wedge C_{i-1}$
1	0	1	0	1		$A_i \wedge \overline{B}_i \wedge C_{i-1}$
1	1	1	1	1	$A_i \wedge B_i \wedge C_{i-1}$	$A_i \wedge B_i \wedge C_{i-1}$

$$\begin{aligned} \Rightarrow S_i &= (A_i \wedge \overline{B}_i \wedge \overline{C}_{i-1}) \vee (\overline{A}_i \wedge B_i \wedge \overline{C}_{i-1}) \vee (\overline{A}_i \wedge \overline{B}_i \wedge C_{i-1}) \vee (A_i \wedge B_i \wedge C_{i-1}) \\ &= (((A_i \wedge \overline{B}_i) \vee (\overline{A}_i \wedge B_i)) \wedge \overline{C}_{i-1}) \vee (((\overline{A}_i \wedge \overline{B}_i) \vee (A_i \wedge B_i)) \wedge C_{i-1}) \\ &= ((A_i \oplus B_i) \wedge \overline{C}_{i-1}) \vee ((A_i = B_i) \wedge C_{i-1}) = ((A_i \oplus B_i) \wedge \overline{C}_{i-1}) \vee ((\overline{A}_i \oplus \overline{B}_i) \wedge C_{i-1}) \\ &= (A_i \oplus B_i) \oplus C_{i-1} \end{aligned}$$



Digital Logic

Full Adder

A_i	B_i	C_{i-1}	S_i	C_i	S_i minterms	C_i minterms
0	0	0	0	0		
0	1	0	1	0	$\overline{A}_i \wedge B_i \wedge \overline{C}_{i-1}$	
1	0	0	1	0	$A_i \wedge \overline{B}_i \wedge \overline{C}_{i-1}$	
1	1	0	0	1		$A_i \wedge B_i \wedge \overline{C}_{i-1}$
0	0	1	1	0	$\overline{A}_i \wedge \overline{B}_i \wedge C_{i-1}$	
0	1	1	0	1		$\overline{A}_i \wedge B_i \wedge C_{i-1}$
1	0	1	0	1		$A_i \wedge \overline{B}_i \wedge C_{i-1}$
1	1	1	1	1	$A_i \wedge B_i \wedge C_{i-1}$	$A_i \wedge B_i \wedge C_{i-1}$

☞ $S_i = (A_i \oplus B_i) \oplus C_{i-1}$

☞ $C_i = (A_i \wedge B_i \wedge \overline{C}_{i-1}) \vee (\overline{A}_i \wedge B_i \wedge C_{i-1}) \vee (A_i \wedge \overline{B}_i \wedge C_{i-1}) \vee (A_i \wedge B_i \wedge C_{i-1})$



Digital Logic

Full Adder

A_i	B_i	C_{i-1}	S_i	C_i	S_i minterms	C_i minterms
0	0	0	0	0		
0	1	0	1	0	$\overline{A}_i \wedge B_i \wedge \overline{C}_{i-1}$	
1	0	0	1	0	$A_i \wedge \overline{B}_i \wedge \overline{C}_{i-1}$	
1	1	0	0	1		$A_i \wedge B_i \wedge \overline{C}_{i-1}$
0	0	1	1	0	$\overline{A}_i \wedge \overline{B}_i \wedge C_{i-1}$	
0	1	1	0	1		$\overline{A}_i \wedge B_i \wedge C_{i-1}$
1	0	1	0	1		$A_i \wedge \overline{B}_i \wedge C_{i-1}$
1	1	1	1	1	$A_i \wedge B_i \wedge C_{i-1}$	$A_i \wedge B_i \wedge C_{i-1}$

☞ $S_i = (A_i \oplus B_i) \oplus C_{i-1}$

$$\begin{aligned} \Rightarrow C_i &= (A_i \wedge B_i \wedge \overline{C}_{i-1}) \vee (\overline{A}_i \wedge B_i \wedge C_{i-1}) \vee (A_i \wedge \overline{B}_i \wedge C_{i-1}) \vee (A_i \wedge B_i \wedge C_{i-1}) \\ &= (A_i \wedge B_i \wedge \overline{C}_{i-1}) \vee (A_i \wedge B_i \wedge C_{i-1}) \vee (\overline{A}_i \wedge B_i \wedge C_{i-1}) \vee (A_i \wedge \overline{B}_i \wedge C_{i-1}) \\ &= (A_i \wedge B_i) \vee (((\overline{A}_i \wedge B_i) \vee (A_i \wedge \overline{B}_i)) \wedge C_{i-1}) \\ &= (A_i \wedge B_i) \vee ((A_i \oplus B_i) \wedge C_{i-1}) \end{aligned}$$



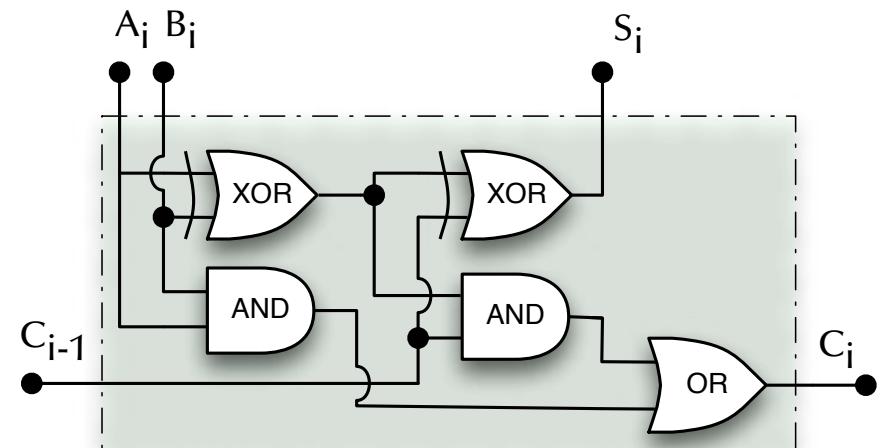
Digital Logic

Full Adder

A_i	B_i	C_{i-1}	S_i	C_i	S_i minterms	C_i minterms
0	0	0	0	0		
0	1	0	1	0	$\overline{A}_i \wedge B_i \wedge \overline{C}_{i-1}$	
1	0	0	1	0	$A_i \wedge \overline{B}_i \wedge \overline{C}_{i-1}$	
1	1	0	0	1		$A_i \wedge B_i \wedge \overline{C}_{i-1}$
0	0	1	1	0	$\overline{A}_i \wedge \overline{B}_i \wedge C_{i-1}$	
0	1	1	0	1		$\overline{A}_i \wedge B_i \wedge C_{i-1}$
1	0	1	0	1		$A_i \wedge \overline{B}_i \wedge C_{i-1}$
1	1	1	1	1	$A_i \wedge B_i \wedge C_{i-1}$	$A_i \wedge B_i \wedge C_{i-1}$

☞ $S_i = (A_i \oplus B_i) \oplus C_{i-1}$

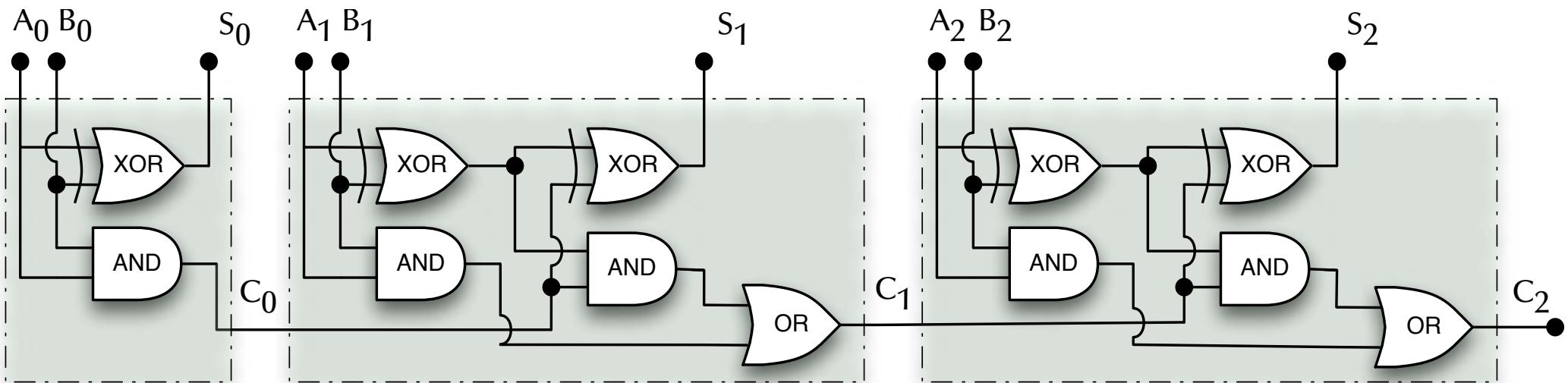
☞ $C_i = (A_i \wedge B_i) \vee ((A_i \oplus B_i) \wedge C_{i-1})$





Digital Logic

Ripple Carry Adder

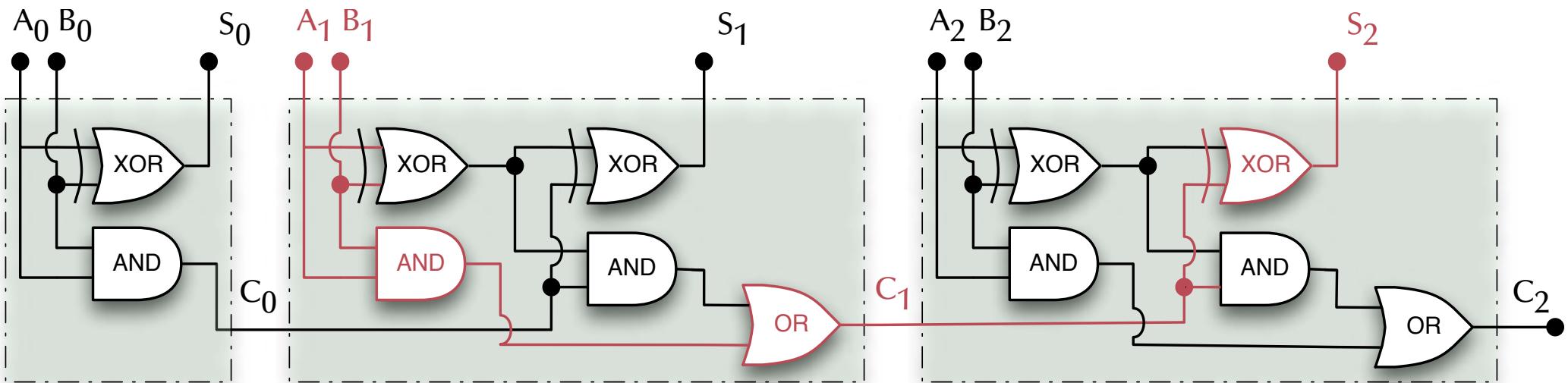


$$2 + 2 = 4 ?$$



Digital Logic

Ripple Carry Adder

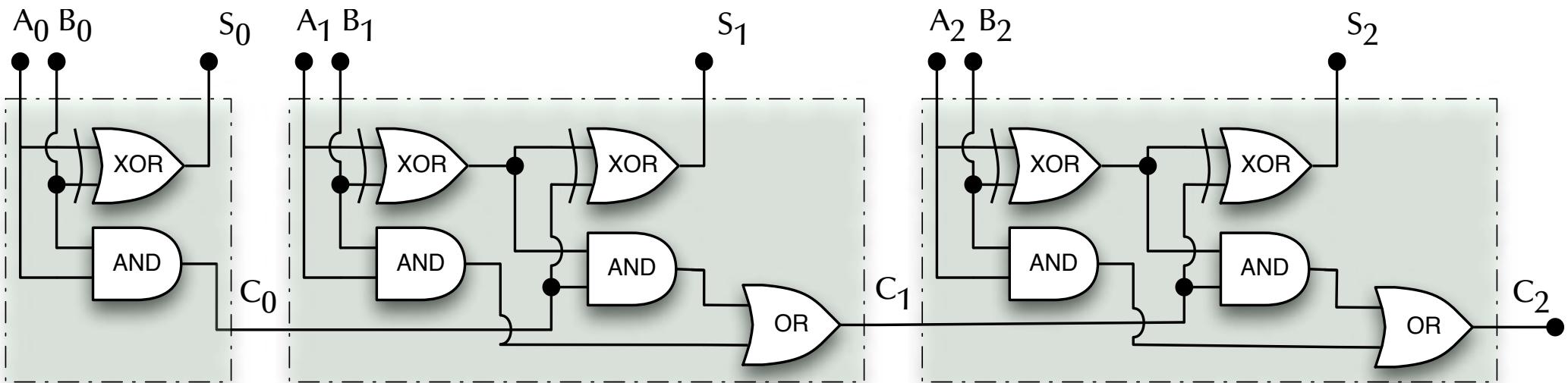


$$2 + 2 = 4 !$$



Digital Logic

Ripple Carry Adder



$$2 - 1 = 1 ?$$



Digital Logic

Radix complements

☞ Can we define negative numbers such that our adder still works?

$$x - x = 0$$

Or: what can you add to 42 in an 8 bit binary representation such that the result will be 2^8 (and hence 0 in 8 bits)?

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

42

+

?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---

-42

=

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

256



Digital Logic

Radix complements

☞ Can we define negative numbers such that our adder still works?

$$x - x = 0$$

Or: what can you add to 42 in an 8 bit binary representation such that the result will be 2^8 (and hence 0 in 8 bits)?

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

42

+

1	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---

-42

=

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

256



Digital Logic

Radix complements

☞ Can we define negative numbers such that our adder still works?

$$x - x = 0$$

Or: what can you add to 42 in an 8 bit binary representation such that the result will be 2^8 (and hence 0 in 8 bits)?

"Invert all bits and add 1"
2's-complement
(as the radix/base is 2)

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

 42

+

1	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---

 -42

=

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 256



Digital Logic

2's complements

The 2's complement encoding interprets the natural binary range $2^{n-1} \dots 2^n - 1$ as negative numbers $-2^{n-1} \dots -1$

Natural binary numbers

0

$2^n - 1$

0	0	0	0	0	0	0	0	...	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	-----	---	---	---	---	---	---	---	---

2's complement binary numbers

-2^{n-1}

0

$2^{n-1} - 1$

1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	...	0	1	1	1	1	1	1	1



Digital Logic

2's complements

The 2's complement encoding interprets
the natural binary range $2^{n-1} \dots 2^n - 1$
as negative numbers $-2^{n-1} \dots -1$

It's all in your mind!

Natural binary numbers

0

$2^n - 1$

0	0	0	0	0	0	0	0	...	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	-----	---	---	---	---	---	---	---	---

2's complement binary numbers

-2^{n-1}

0

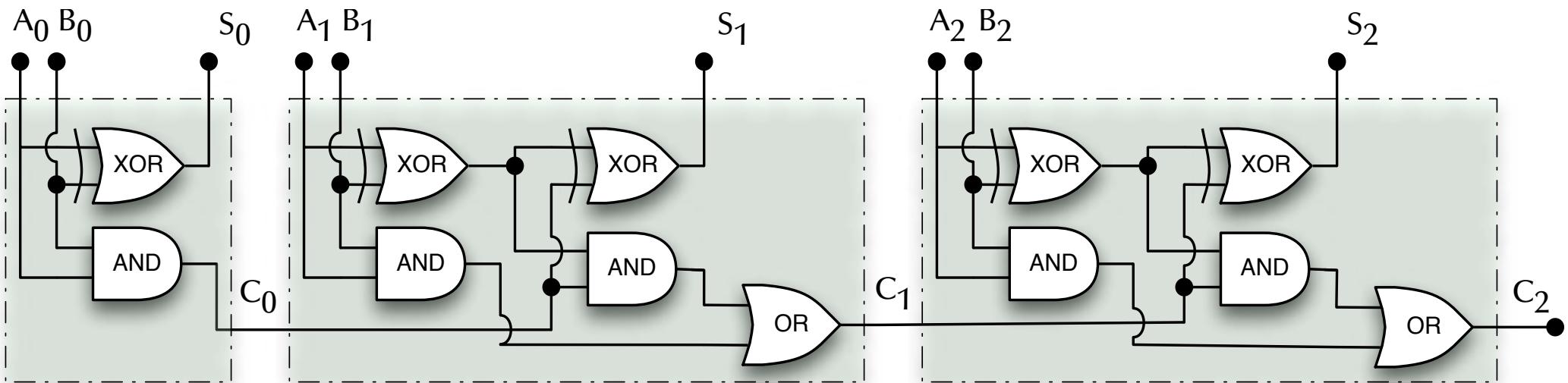
$2^{n-1} - 1$

1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	...	0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	-----	---	---	---	---	---	---	---	---	-----	---	---	---	---	---	---	---	---



Digital Logic

Ripple Carry Adder

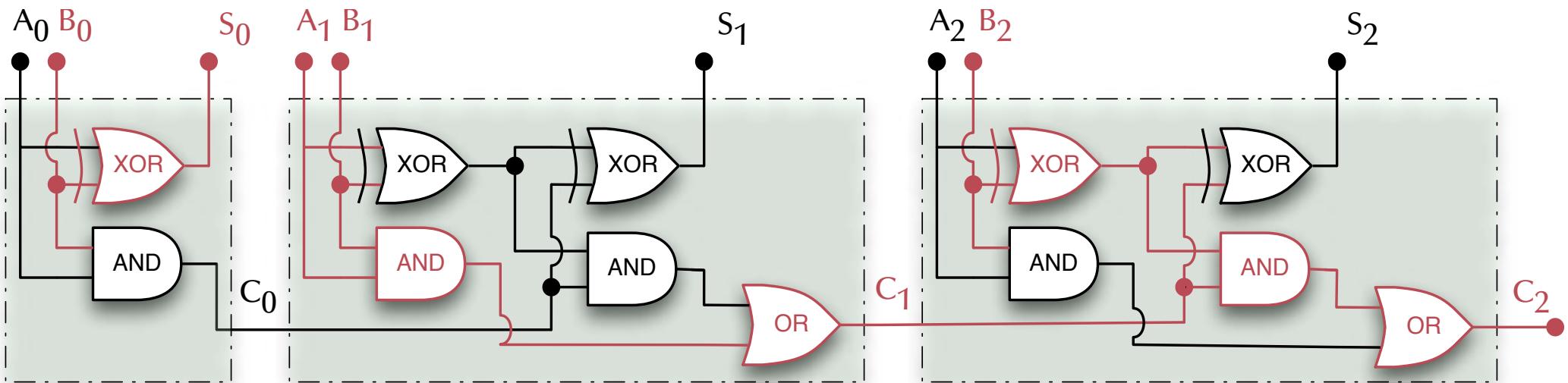


$$2 - 1 = 1 ?$$



Digital Logic

Ripple Carry Adder



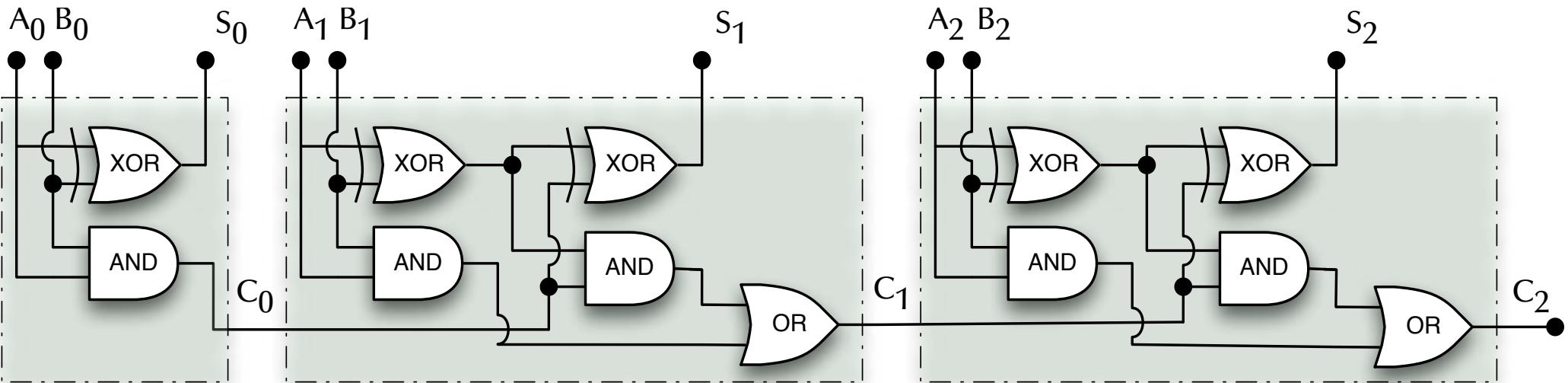
$$2 - 1 = 1 !$$

... with an overall carry-flag indicated.



Digital Logic

Ripple Carry Adder

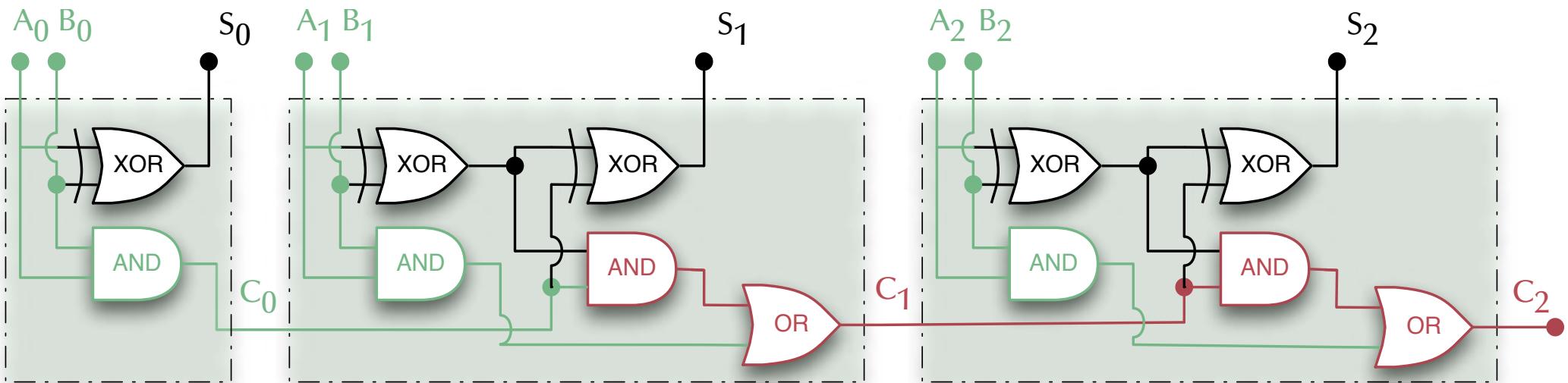


How long does it take until the last carry flag stabilizes ?



Digital Logic

Ripple Carry Adder



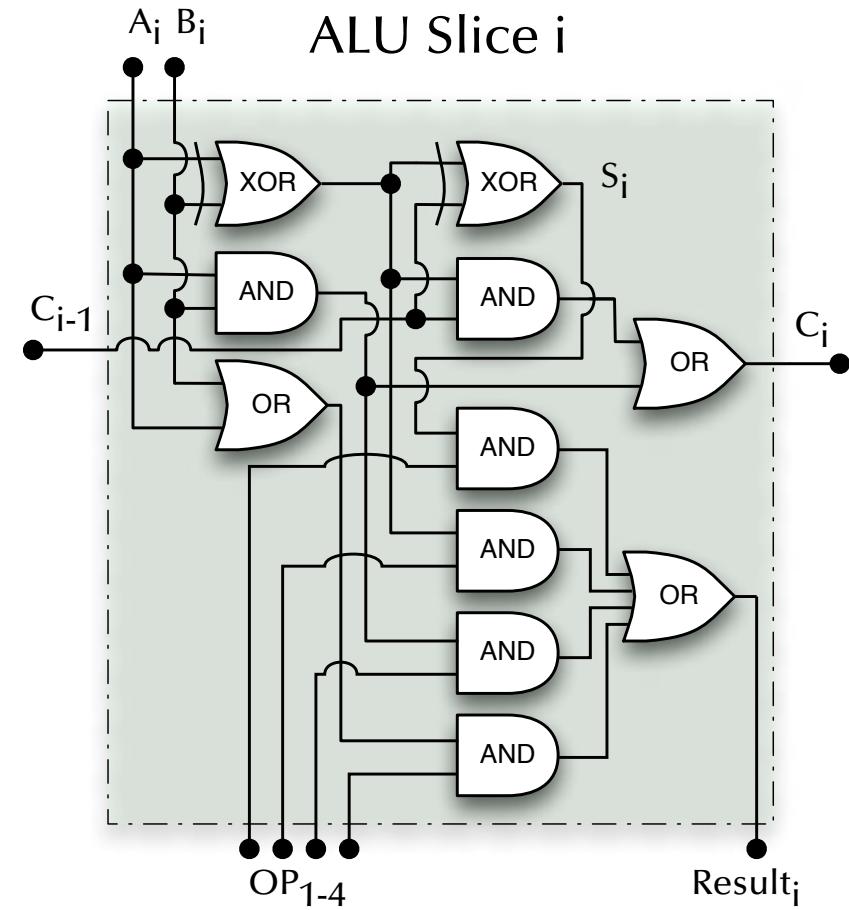
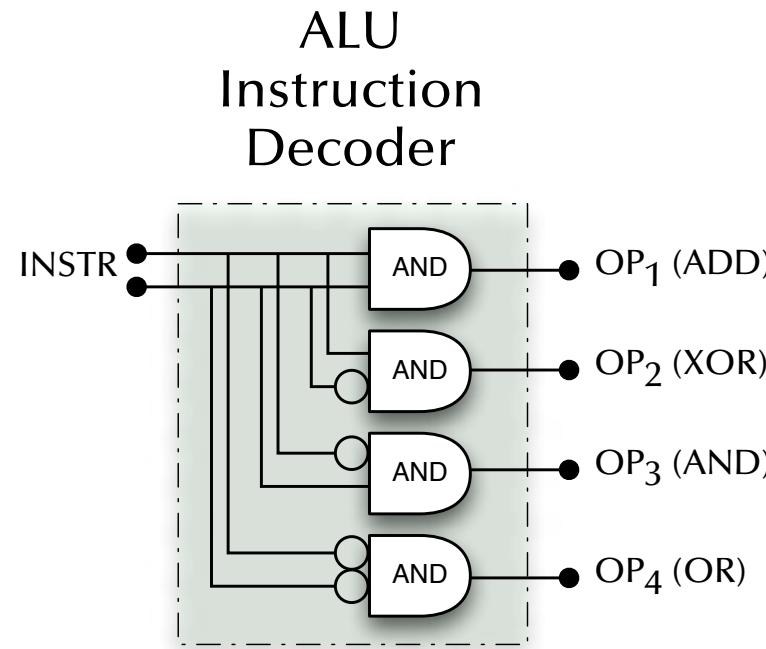
What distinguishes the red from the green gates ?

☞ Carry-lookahead circuitry



Digital Logic

Arithmetic Logic Unit (ALU)



A simple ALU which can ADD, XOR, AND, OR two arguments.



Digital Logic

Towards States

(everything up to here was combinational logic)

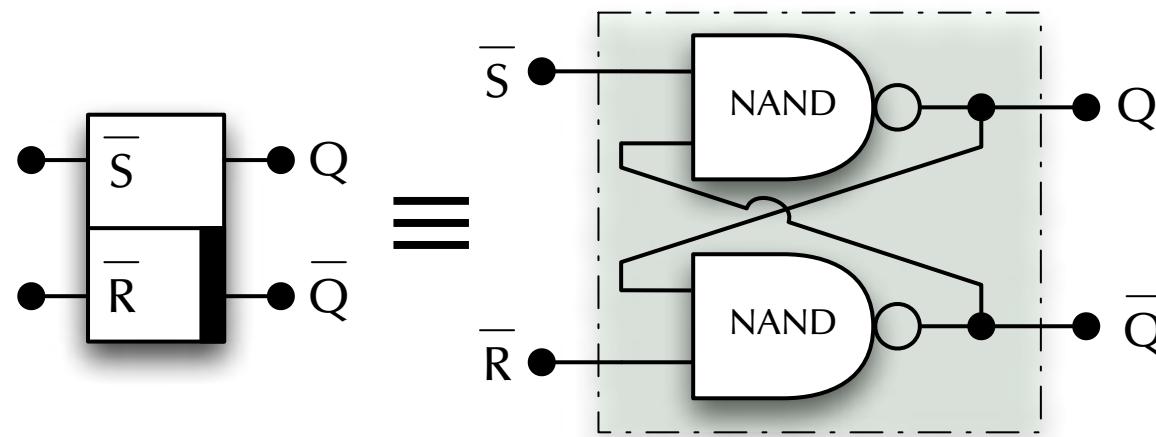
How do we make operations depends on:

- ☞ ... an overflow in the previous operation?
- ☞ ... the state of the CPU?
- ☞ ... a counter having reached zero?
- ☞ ... two arguments having been equal?
- ☞ ... etc. pp.
- ☞ We need to hold on to some states!



Digital Logic

States

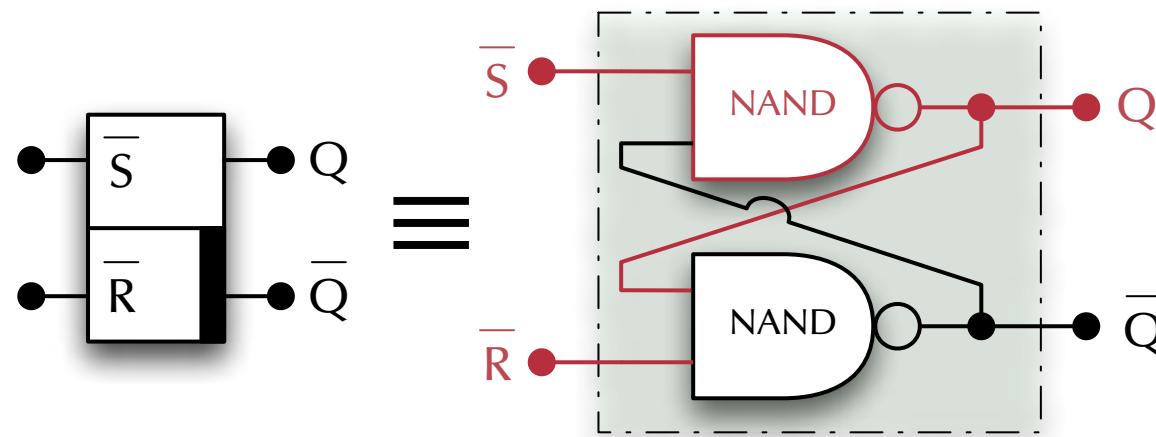


\bar{S}	\bar{R}	Q	\bar{Q}
0	0	?	?
0	1	?	?
1	0	?	?
1	1	?	?



Digital Logic

States

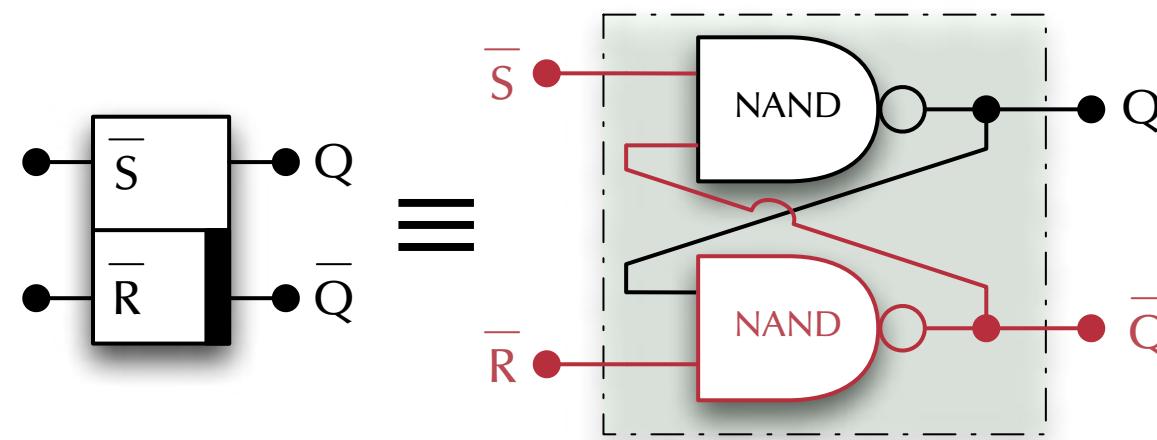


\bar{S}	\bar{R}	Q	\bar{Q}
0	0	?	?
0	1	?	?
1	0	?	?
1	1	Q	\bar{Q}



Digital Logic

States

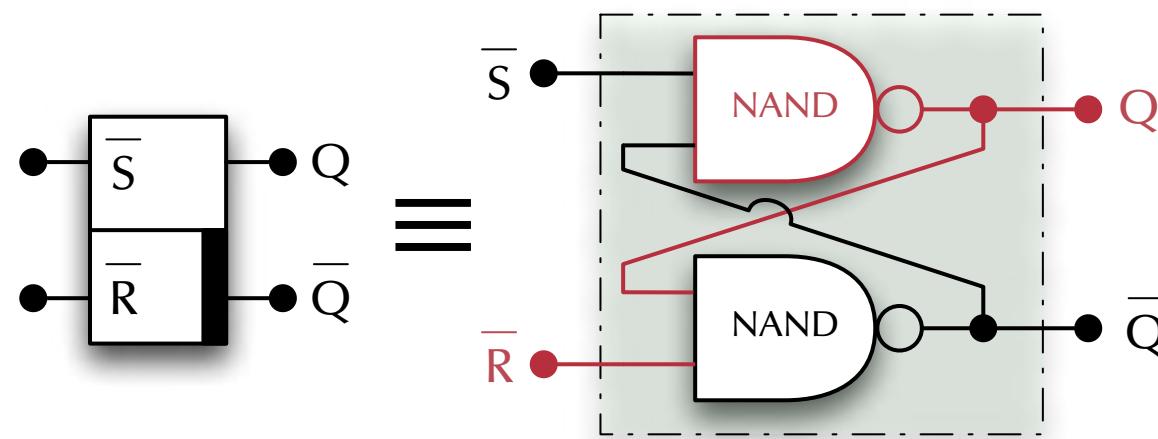


\bar{S}	\bar{R}	Q	\bar{Q}
0	0	?	?
0	1	?	?
1	0	?	?
1	1	Q	\bar{Q}



Digital Logic

States

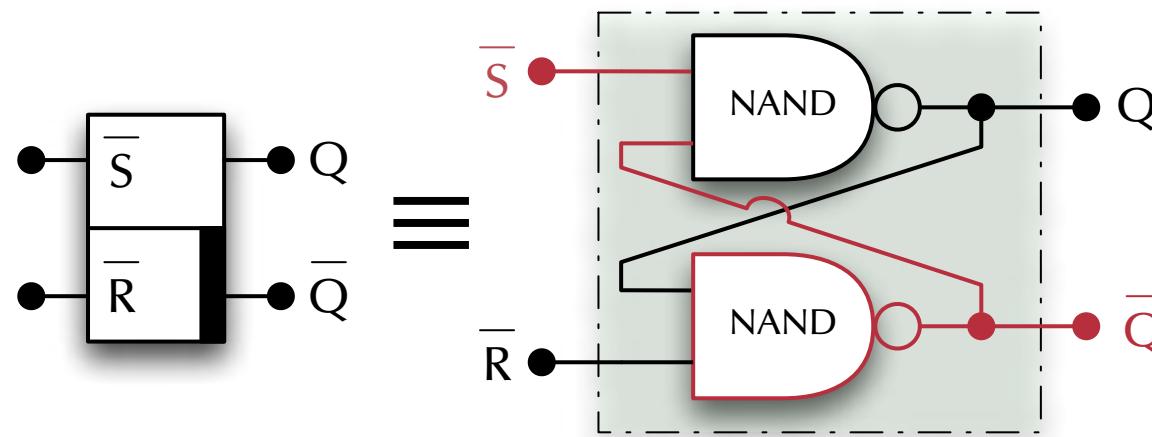


\bar{S}	\bar{R}	Q	\bar{Q}
0	0	?	?
0	1	1	0
1	0	?	?
1	1	Q	\bar{Q}



Digital Logic

States

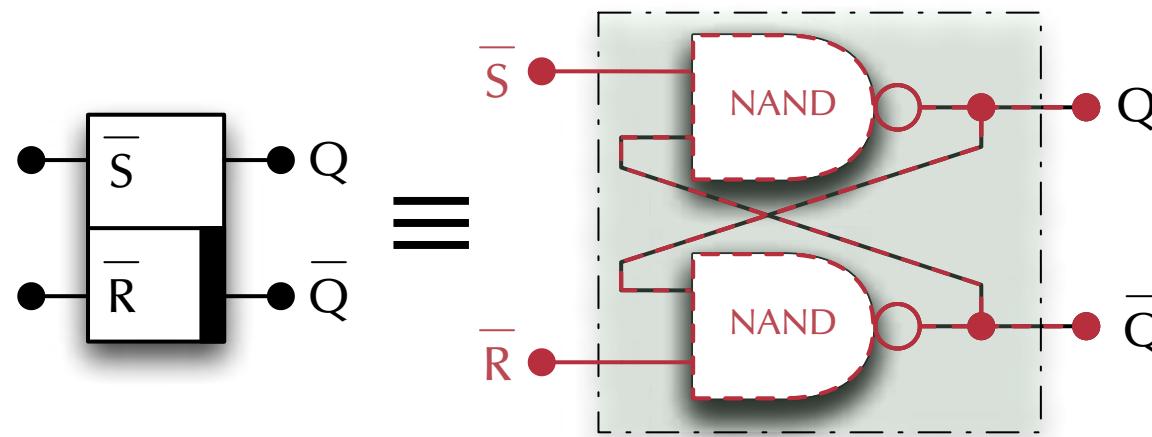


\bar{S}	\bar{R}	Q	\bar{Q}
0	0	?	?
0	1	1	0
1	0	0	1
1	1	Q	\bar{Q}



Digital Logic

States



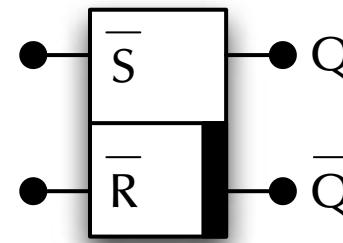
\bar{S}	\bar{R}	Q	\bar{Q}
0	0	$\frac{1}{2}$	$\frac{1}{2}$
0	1	1	0
1	0	0	1
1	1	Q	\bar{Q}

Assuming Q as well as \bar{Q} to be active simultaneously may lead to instability.

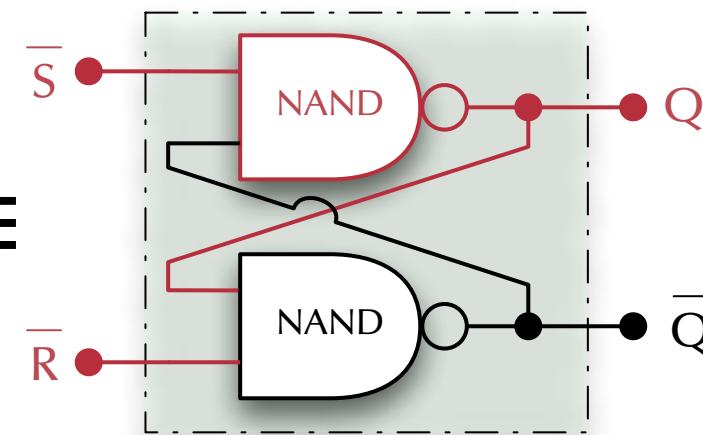


Digital Logic

“S-R Flip-Flop”



States



\bar{S}	\bar{R}	Q	\bar{Q}
0	0	Forbidden	
0	1	1	0
1	0	0	1
1	1	Q	\bar{Q}



Digital Logic

Deriving SR Flip Flops

\bar{S}	\bar{R}	Q	Q		
0	0	0	*		
0	0	1	*		
0	1	0	1		
0	1	1	1		
1	0	0	0		
1	0	1	0		
1	1	0	0		
1	1	1	1		



Digital Logic

Deriving SR Flip Flops

\bar{S}	\bar{R}	Q	Q	Q minterms	
0	0	0	*	$S \wedge R \wedge \bar{Q}$	
0	0	1	*	$S \wedge R \wedge Q$	
0	1	0	1	$S \wedge \bar{R} \wedge \bar{Q}$	
0	1	1	1	$S \wedge \bar{R} \wedge Q$	
1	0	0	0		
1	0	1	0		
1	1	0	0		
1	1	1	1	$\bar{S} \wedge \bar{R} \wedge Q$	



Digital Logic

Deriving SR Flip Flops

\bar{S}	\bar{R}	Q	Q	Q minterms	Simplified
0	0	0	*	$S \wedge R \wedge \bar{Q}$	
0	0	1	*	$S \wedge R \wedge Q$	
0	1	0	1	$S \wedge \bar{R} \wedge \bar{Q}$	S
0	1	1	1	$S \wedge \bar{R} \wedge Q$	
1	0	0	0		
1	0	1	0		
1	1	0	0		$\bar{R} \wedge Q$
1	1	1	1	$\bar{S} \wedge \bar{R} \wedge Q$	

👉 $Q = S \vee (\bar{R} \wedge Q)$



Digital Logic

Deriving SR Flip Flops

\bar{S}	\bar{R}	Q	Q	Q minterms	Simplified
0	0	0	*	$S \wedge R \wedge \bar{Q}$	
0	0	1	*	$S \wedge R \wedge Q$	
0	1	0	1	$S \wedge \bar{R} \wedge \bar{Q}$	S
0	1	1	1	$S \wedge \bar{R} \wedge Q$	
1	0	0	0		
1	0	1	0		
1	1	0	0		$\bar{R} \wedge Q$
1	1	1	1	$\bar{S} \wedge \bar{R} \wedge Q$	

👉 $Q = S \vee (\bar{R} \wedge Q) = \overline{\bar{S} \wedge \bar{R} \wedge \bar{Q}}$

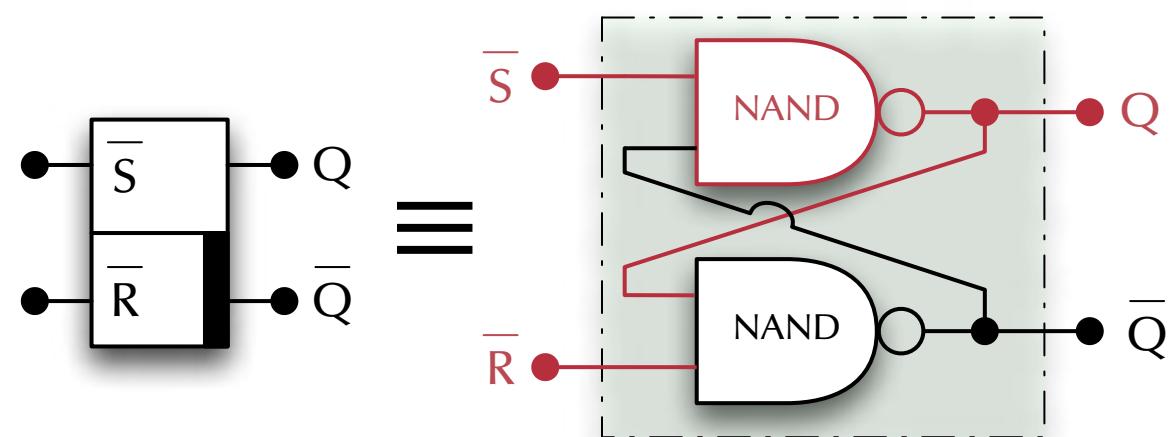


Digital Logic

Deriving SR Flip Flops

\bar{S}	\bar{R}	Q	Q	Q minterms	Simplified
0	0	0	*	$S \wedge R \wedge \bar{Q}$	
0	0	1	*	$S \wedge R \wedge Q$	
0	1	0	1	$S \wedge \bar{R} \wedge \bar{Q}$	S
0	1	1	1	$S \wedge \bar{R} \wedge Q$	$S \wedge \bar{R} \wedge Q$
1	0	0	0		
1	0	1	0		
1	1	0	0		$\bar{R} \wedge Q$
1	1	1	1	$\bar{S} \wedge \bar{R} \wedge Q$	

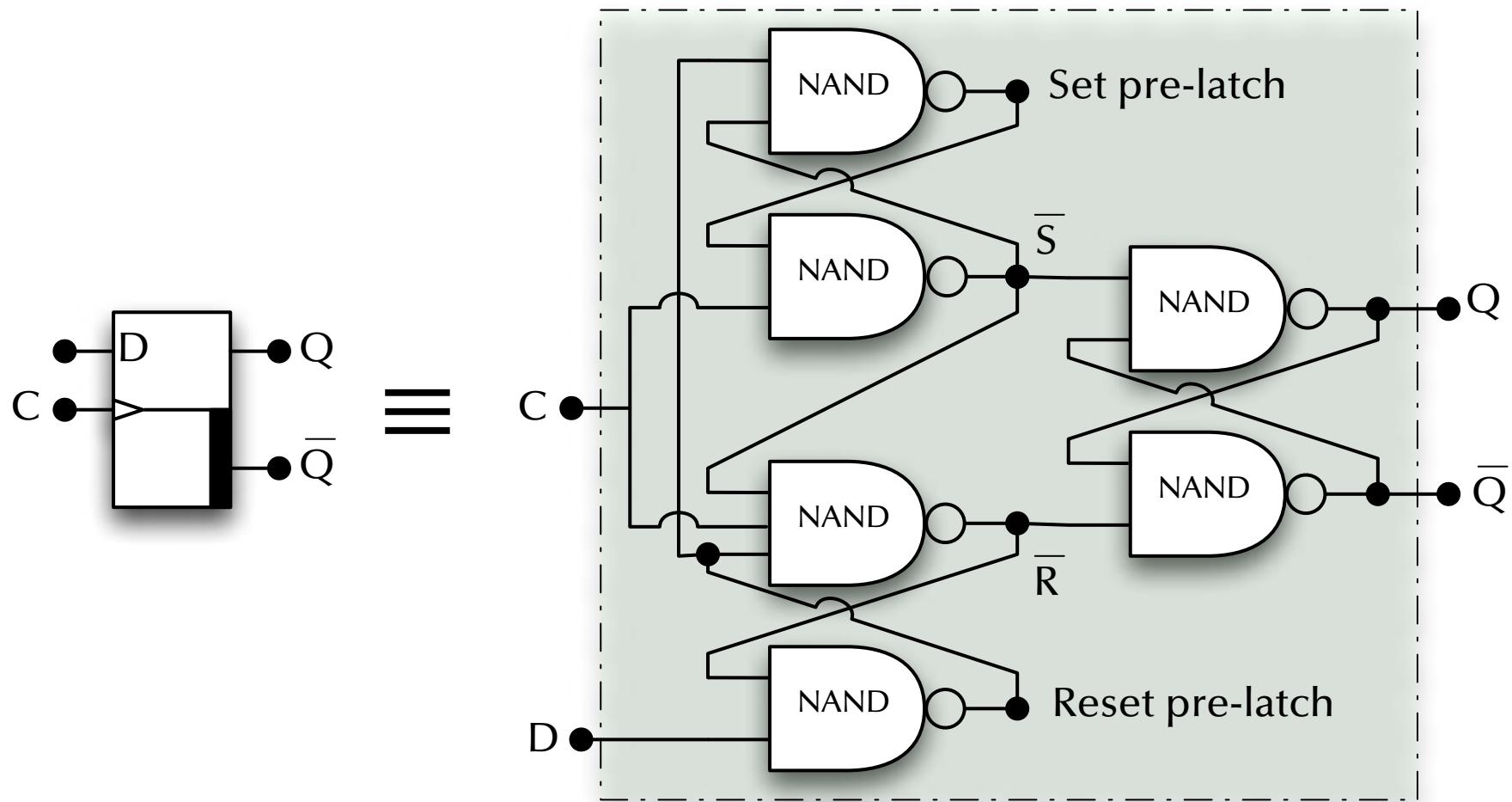
👉 $Q = S \vee (\bar{R} \wedge Q) = \overline{\bar{S} \wedge \bar{R} \wedge \bar{Q}}$





Digital Logic

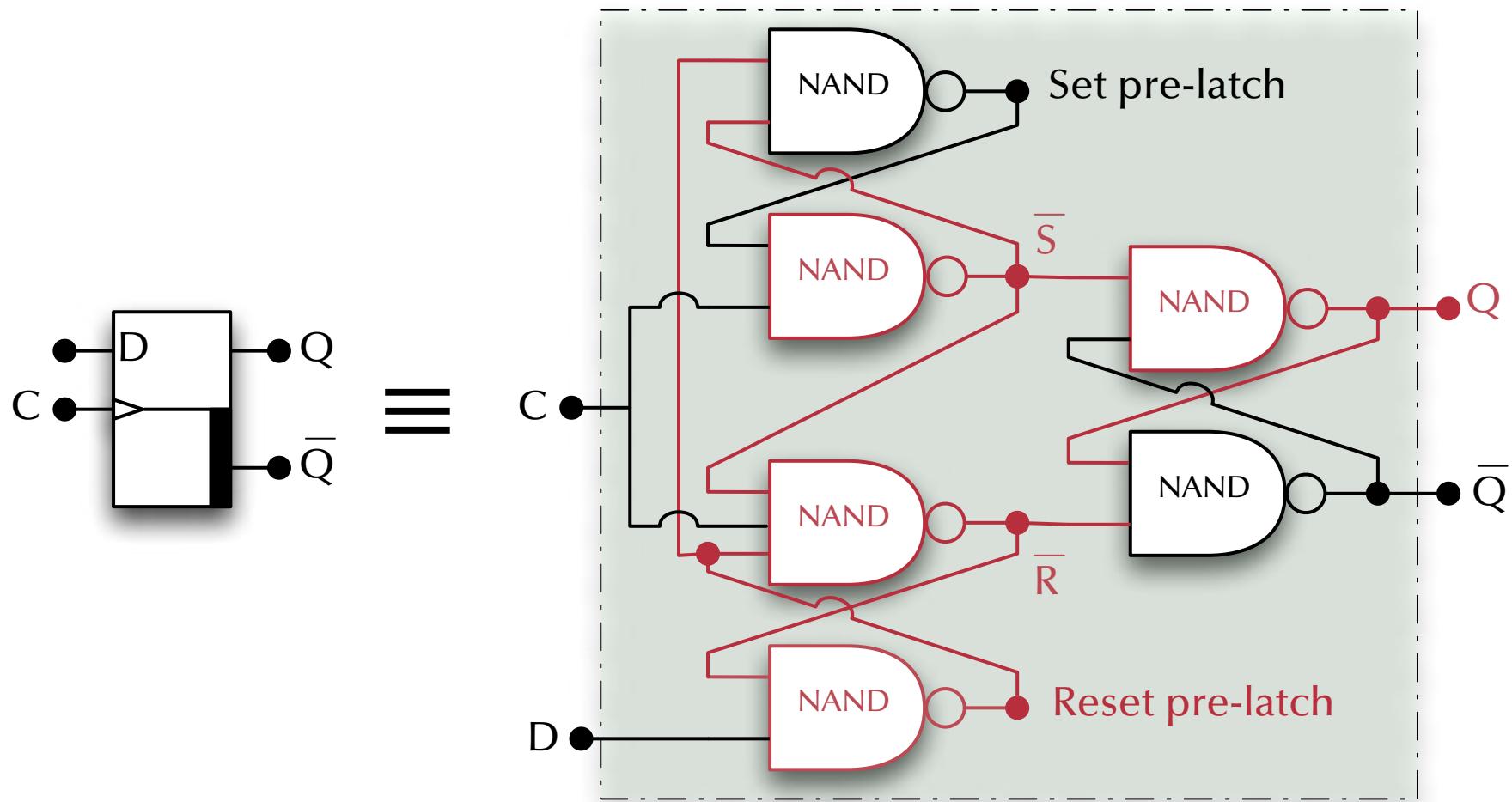
D Flip-Flop





Digital Logic

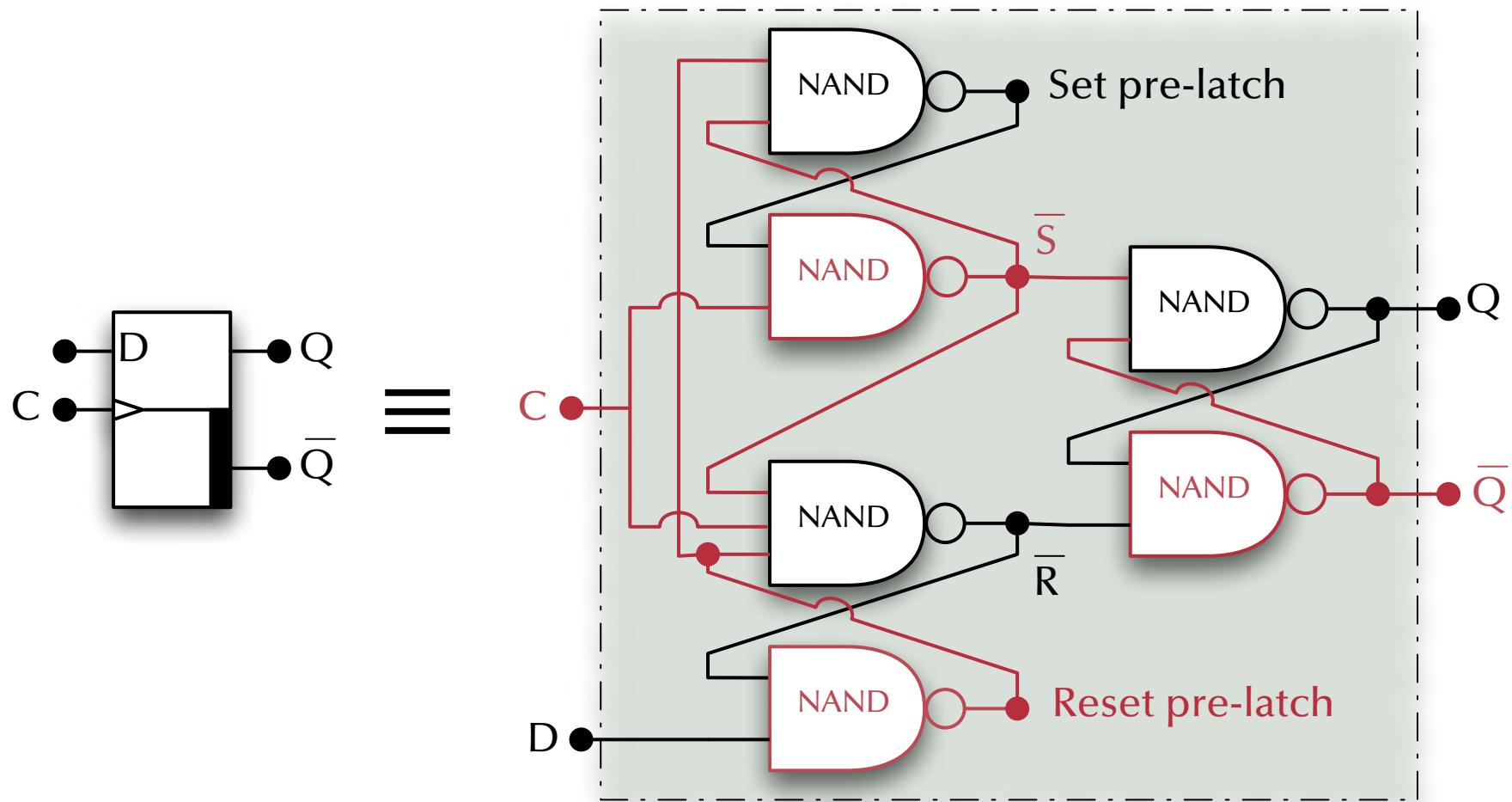
D Flip-Flop





Digital Logic

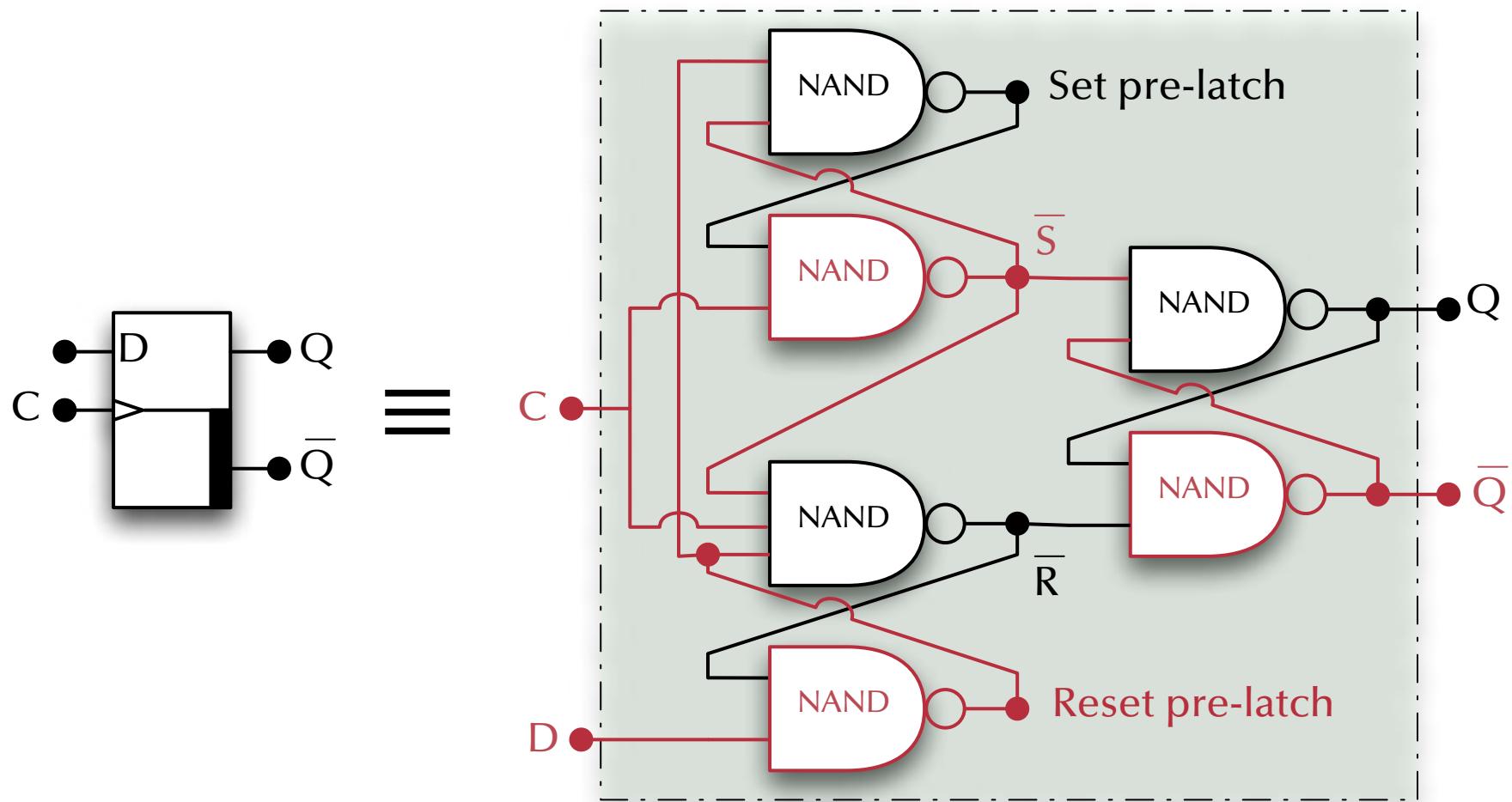
D Flip-Flop





Digital Logic

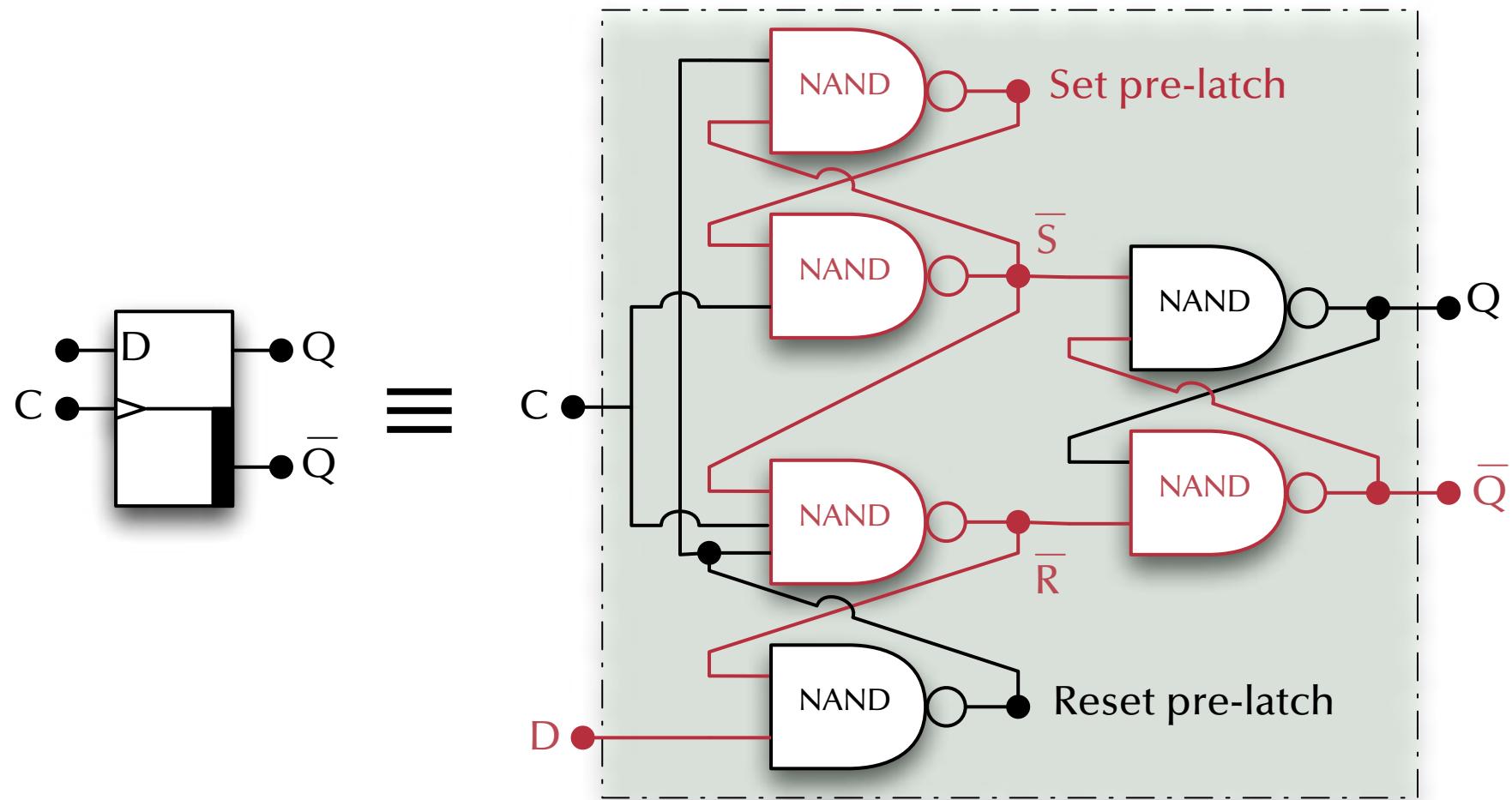
D Flip-Flop





Digital Logic

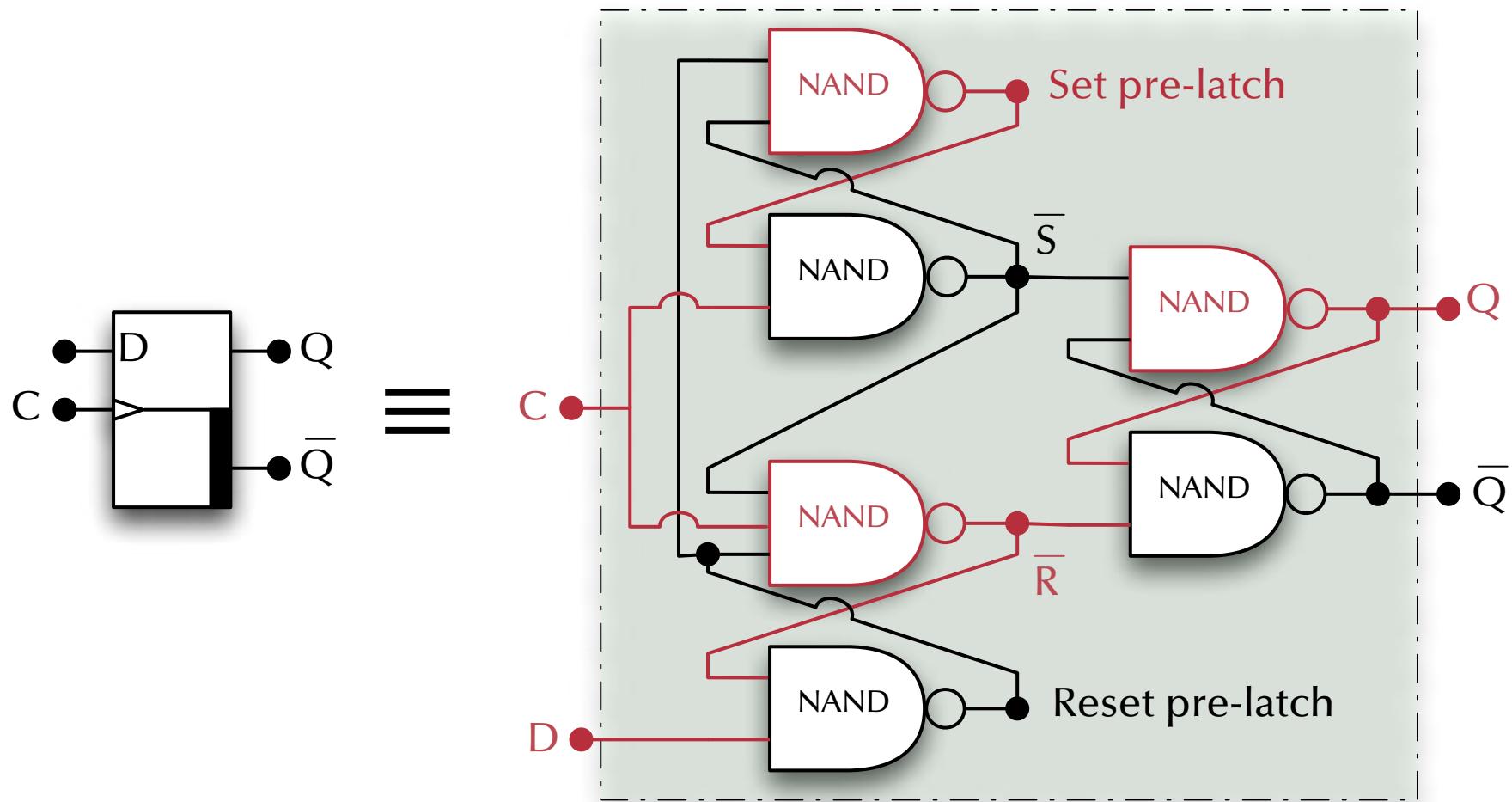
D Flip-Flop





Digital Logic

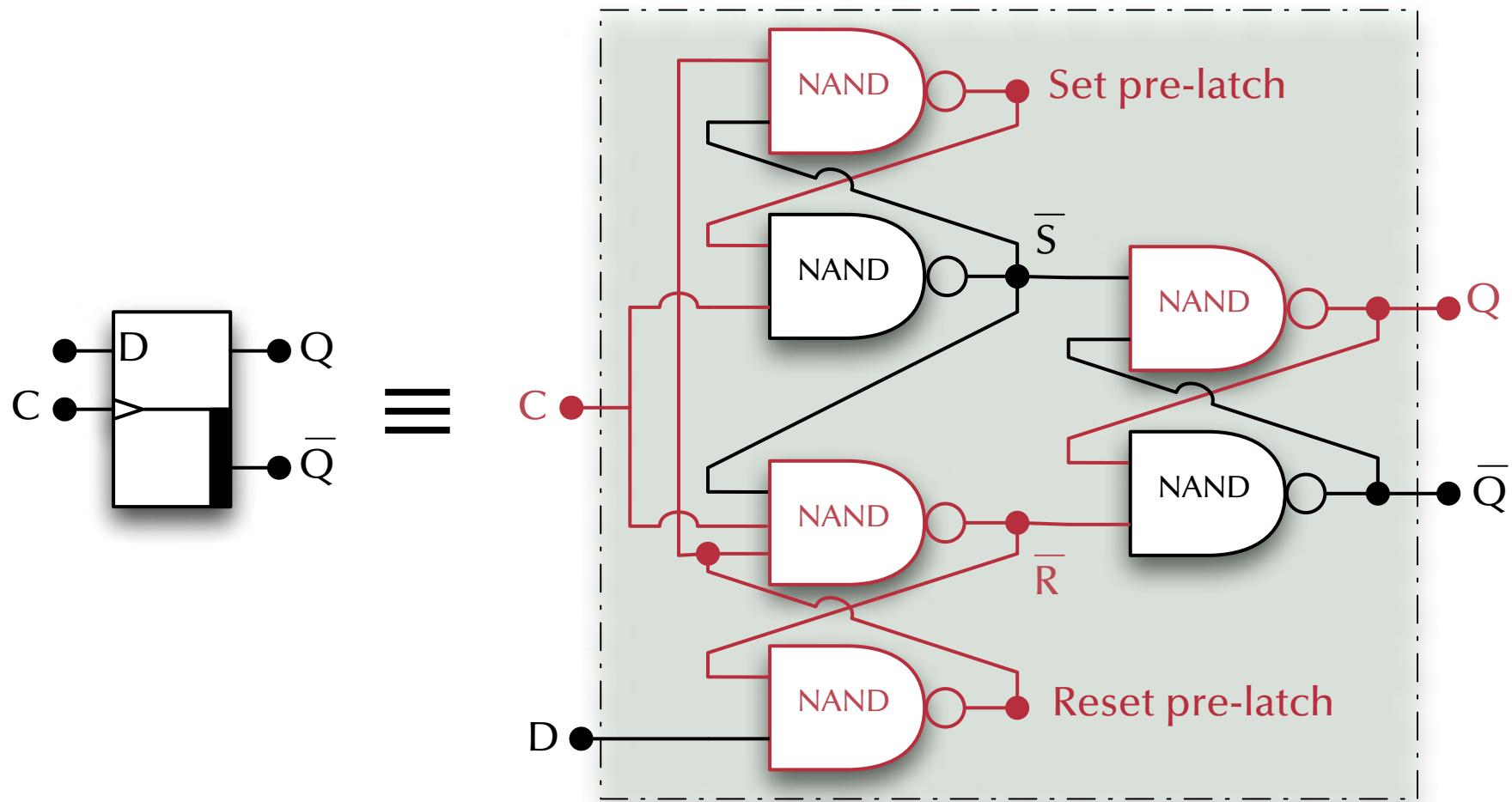
D Flip-Flop





Digital Logic

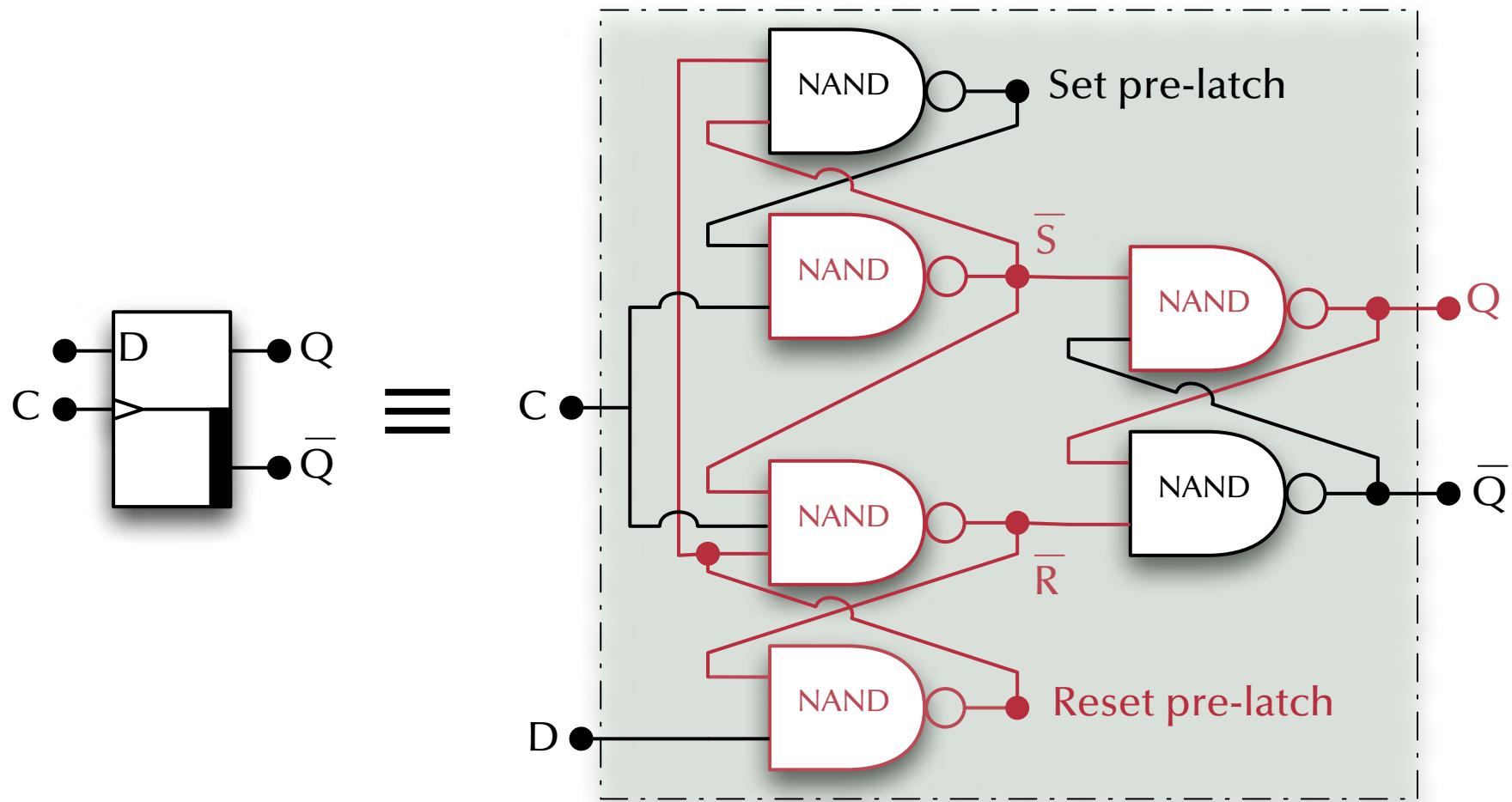
D Flip-Flop





Digital Logic

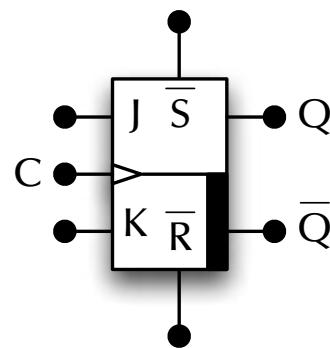
D Flip-Flop



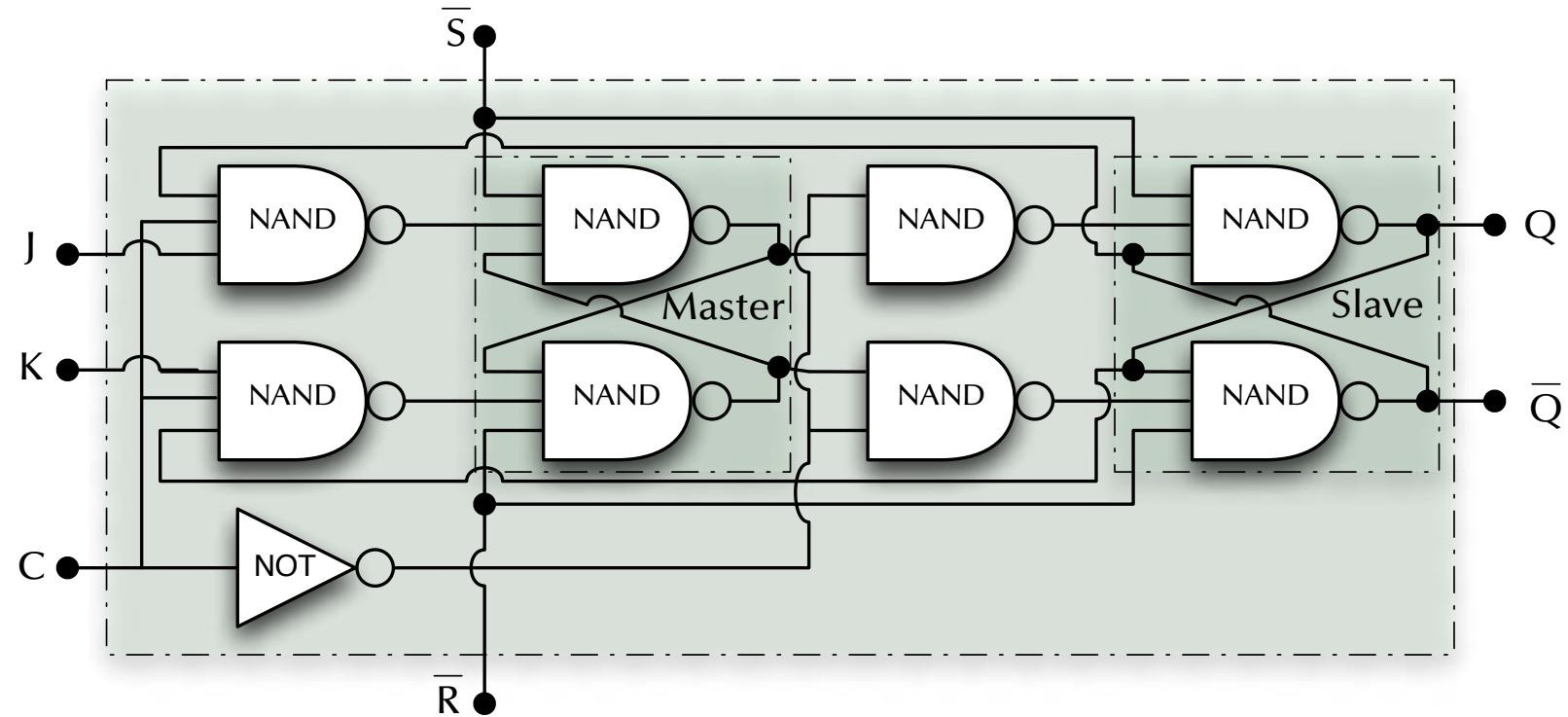


Digital Logic

Master-Slave JK Flip-Flop



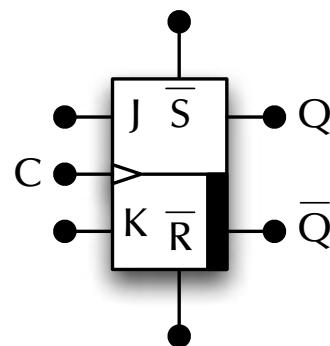
≡



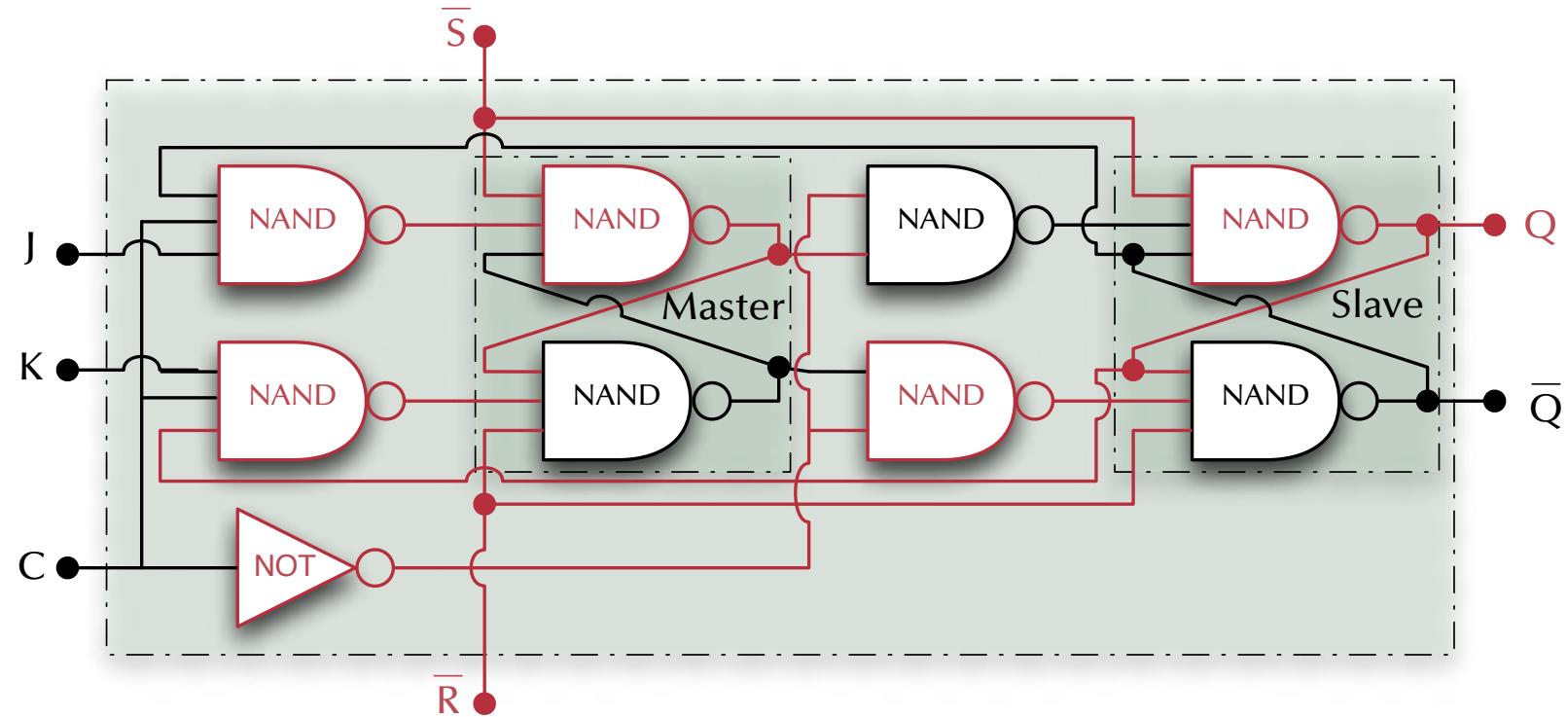


Digital Logic

Master-Slave JK Flip-Flop



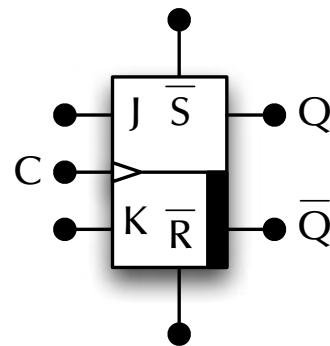
≡



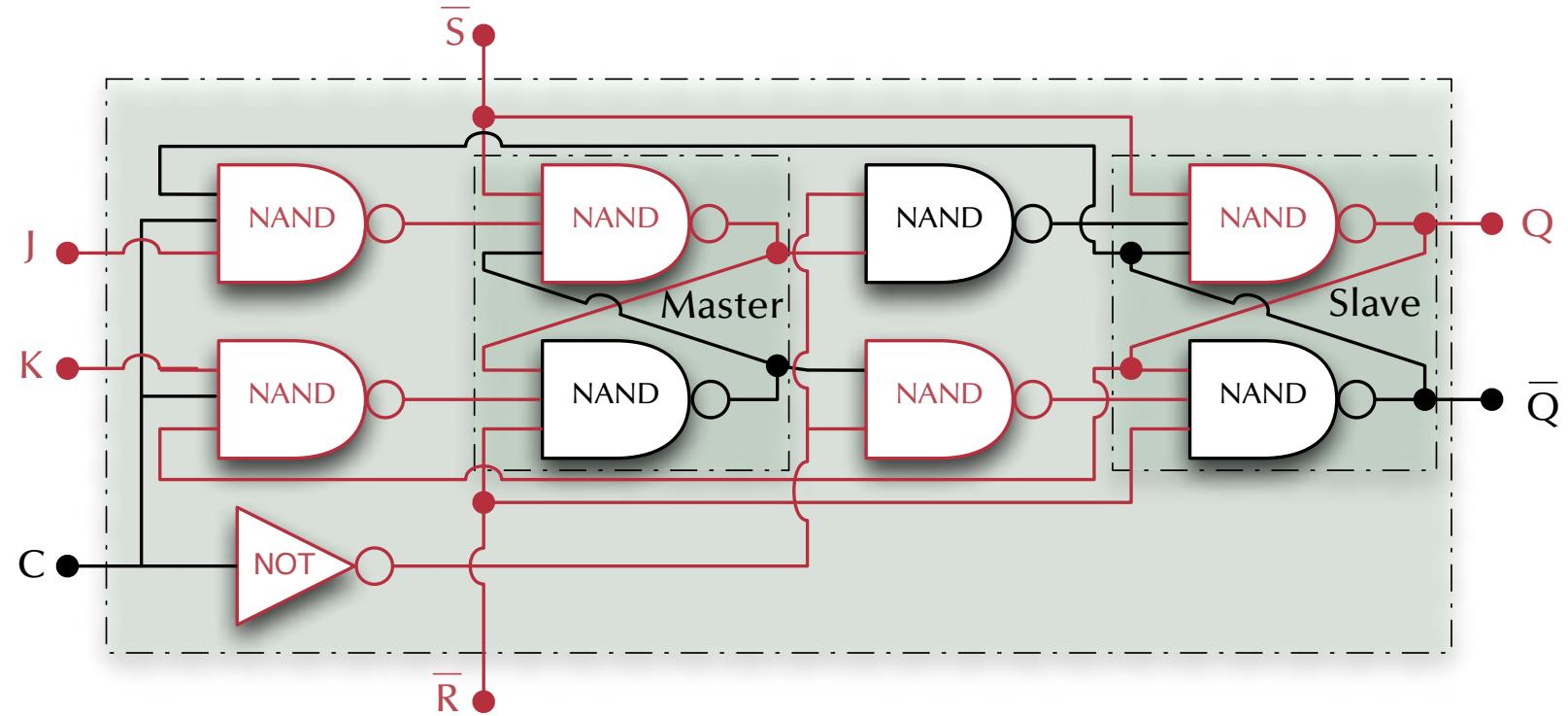


Digital Logic

Master-Slave JK Flip-Flop



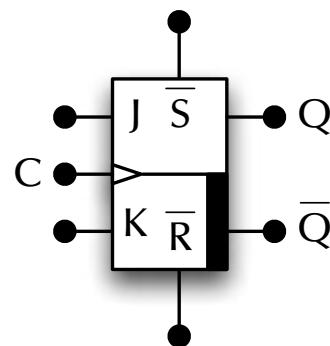
≡



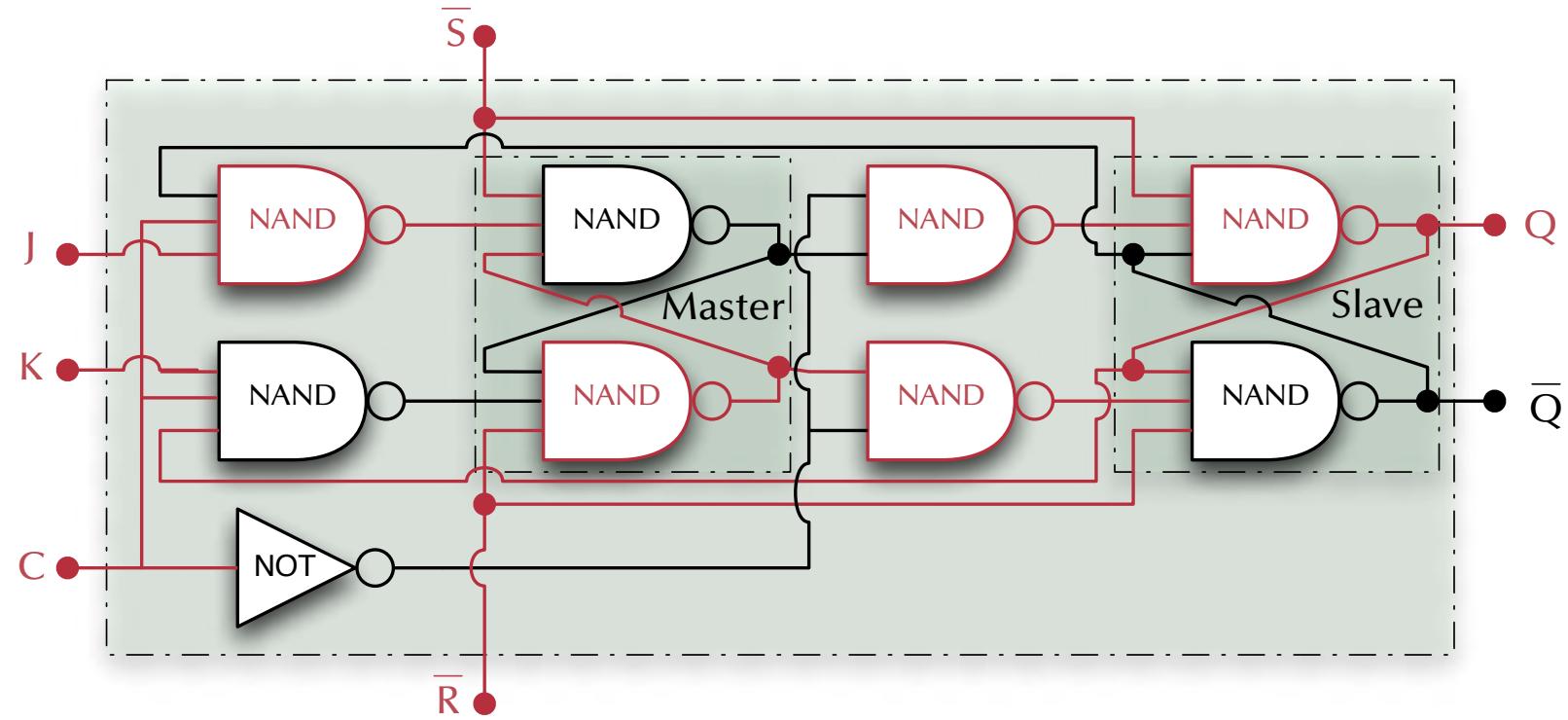


Digital Logic

Master-Slave JK Flip-Flop



≡

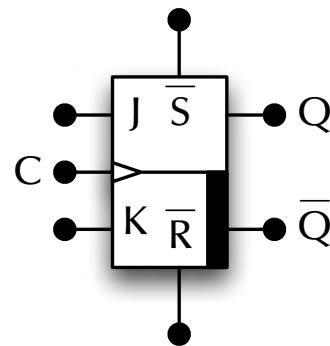


Master is reset on the rising clock edge

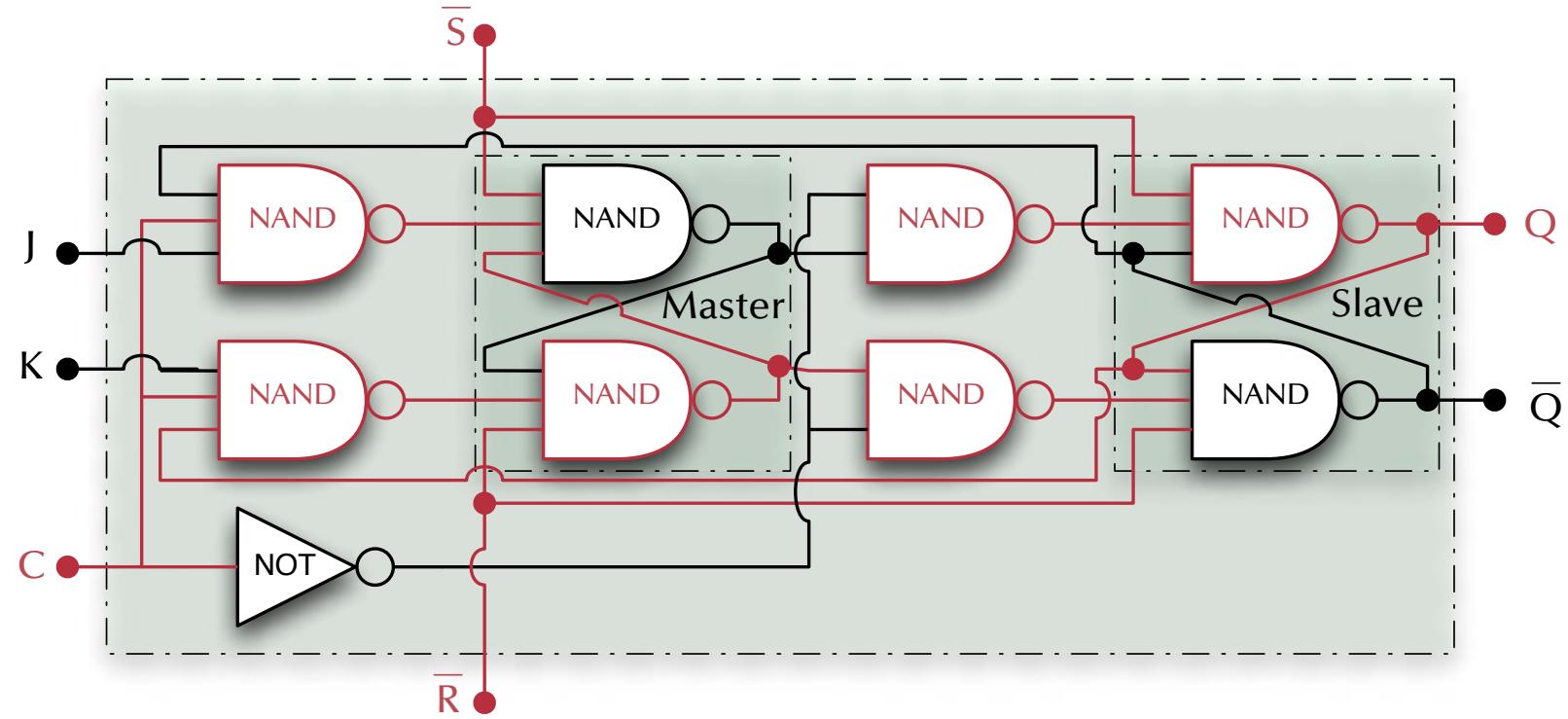


Digital Logic

Master-Slave JK Flip-Flop



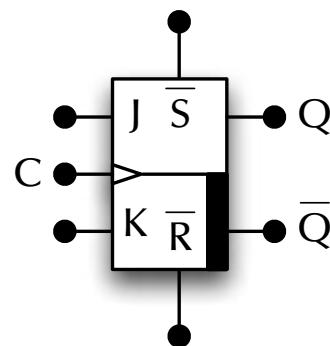
≡



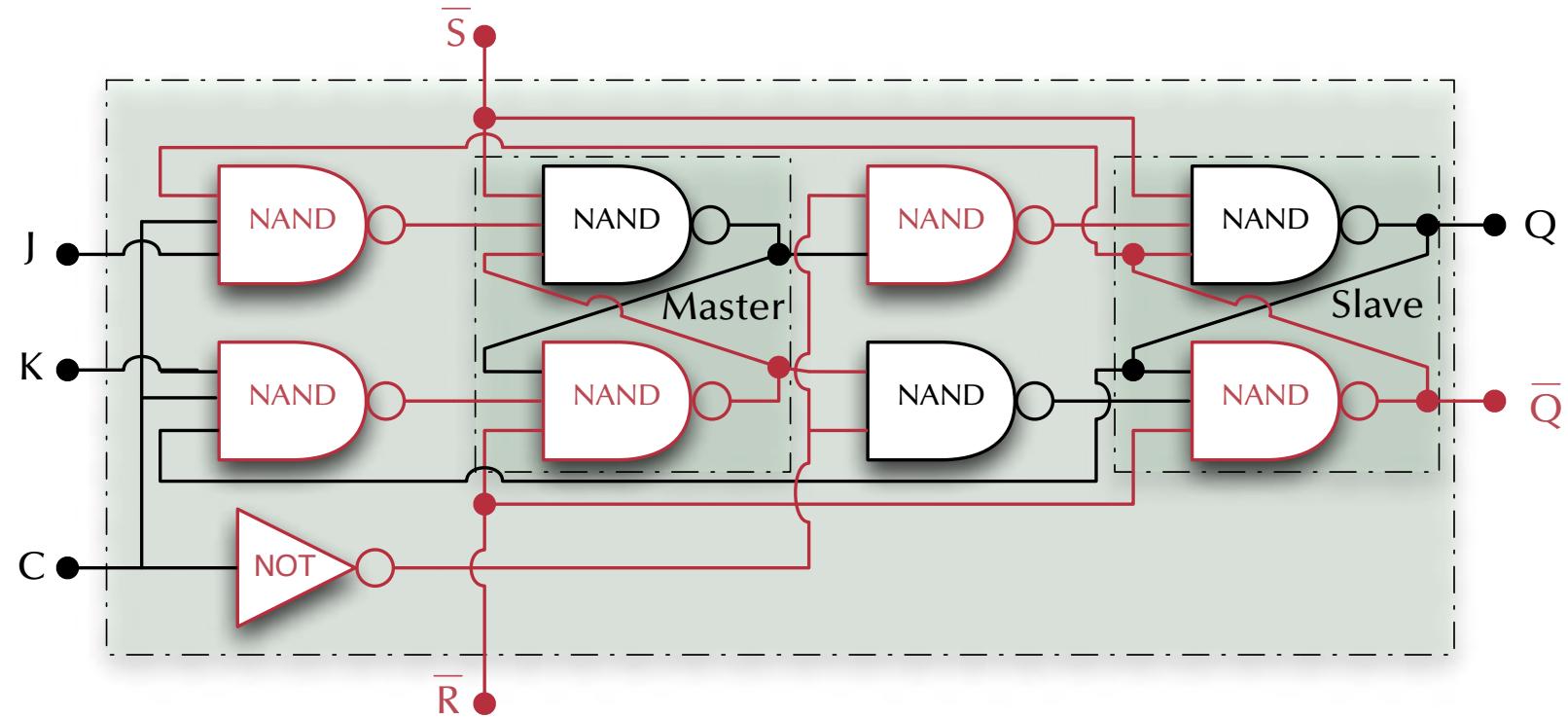


Digital Logic

Master-Slave JK Flip-Flop



≡

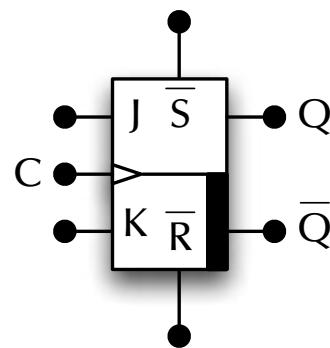


Slave follows on the falling clock edge

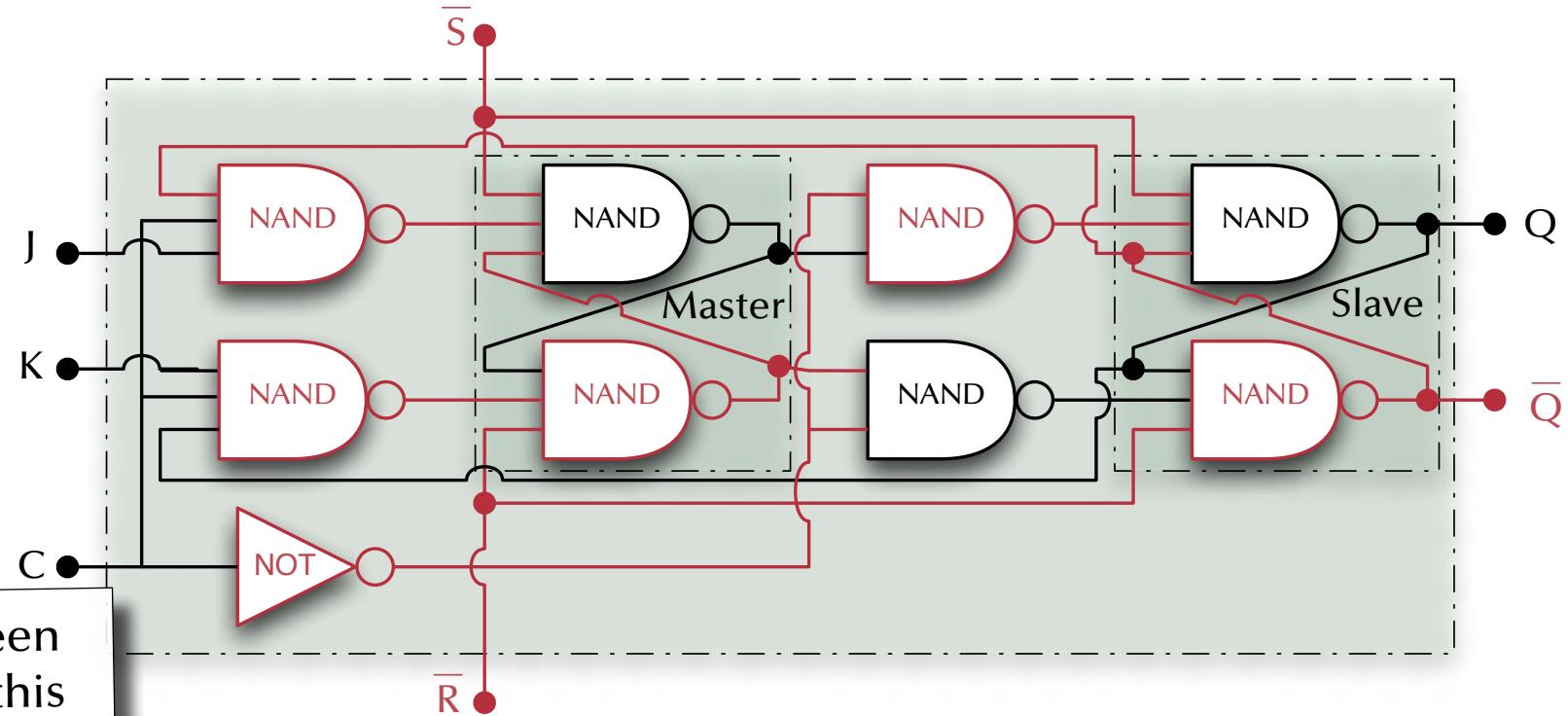


Digital Logic

Master-Slave JK Flip-Flop



≡



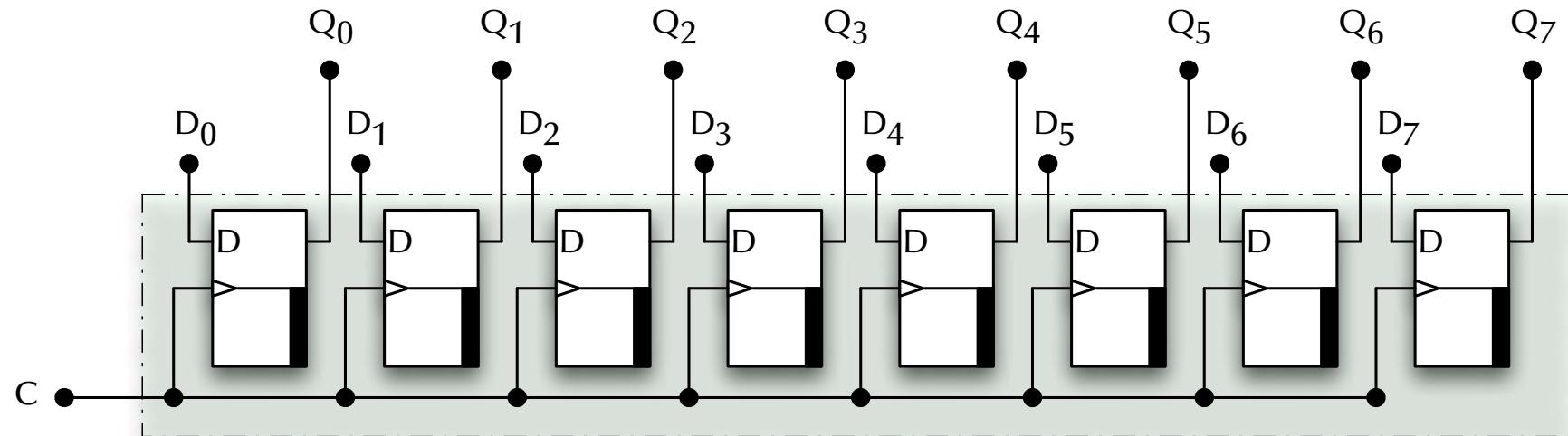
The decoupling between the two stages makes this flip-flop race free – even in JK-toggle mode.

Slave follows on the falling clock edge



Digital Logic

Register



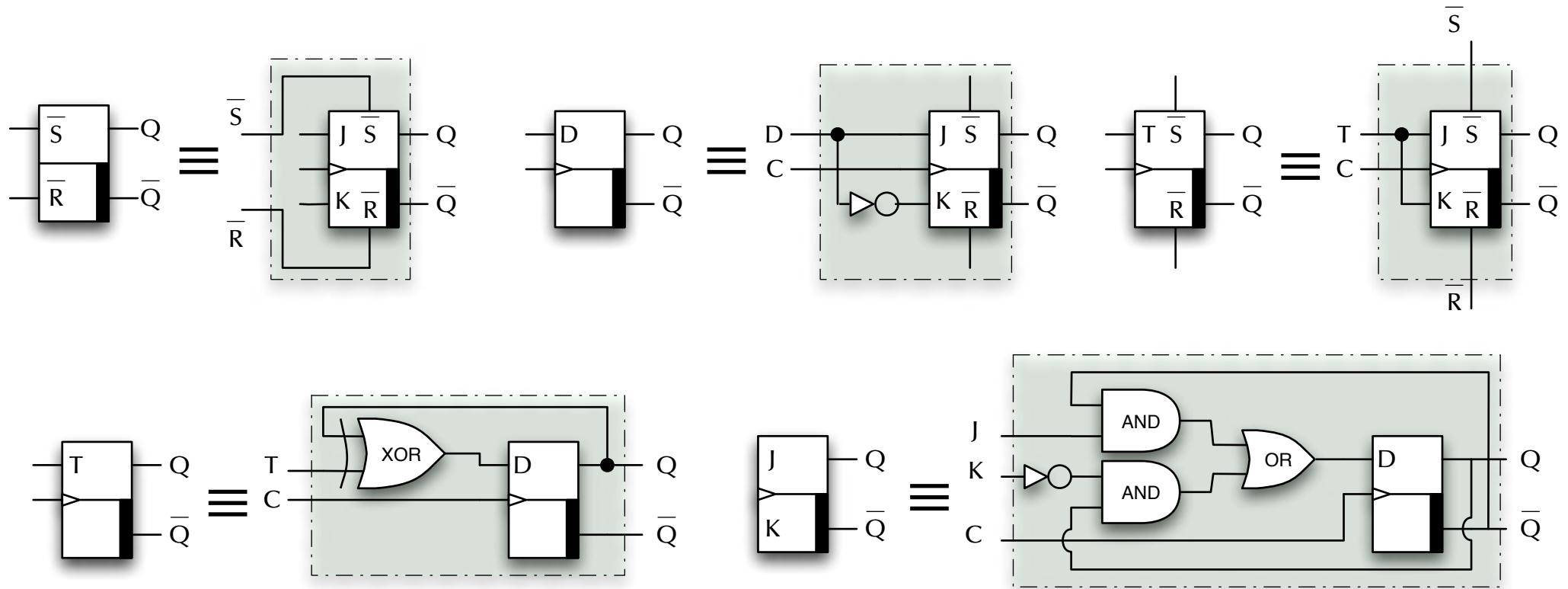
Could serve as a generic, fast storage inside the CPU (general register)

Or to hold internal states (e.g. ALU overflow) of the CPU which are used by e.g. branching instructions.



Digital Logic

Toggle Flip-Flop

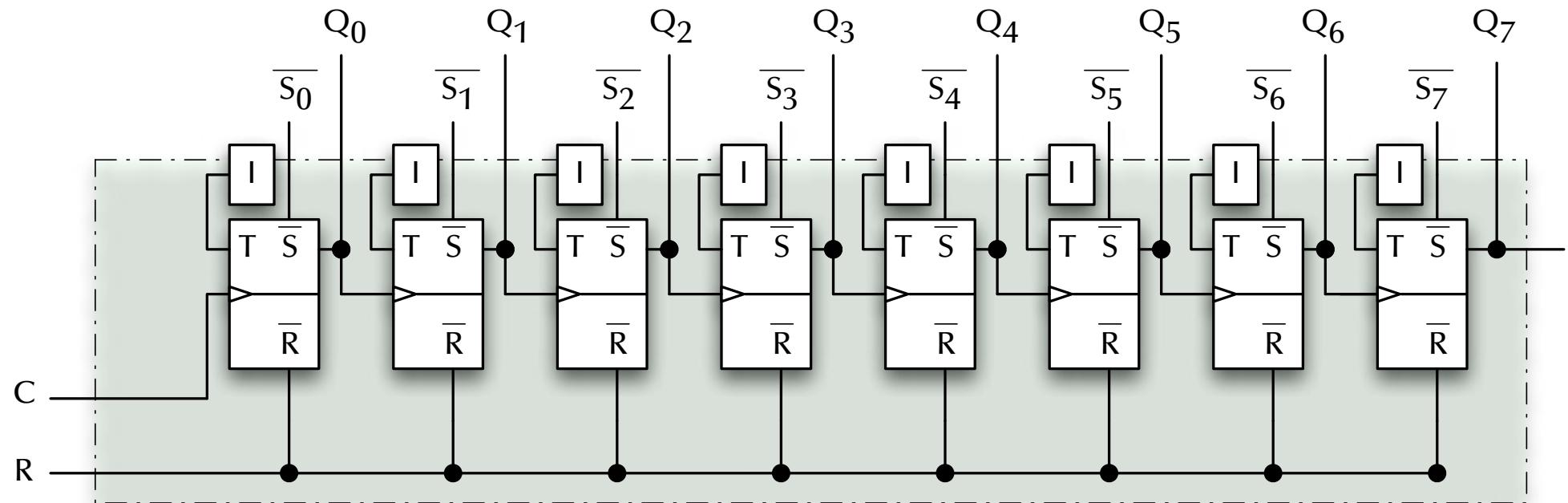


Toggle Flip-Flops change state with every clock cycle.



Digital Logic

Counter

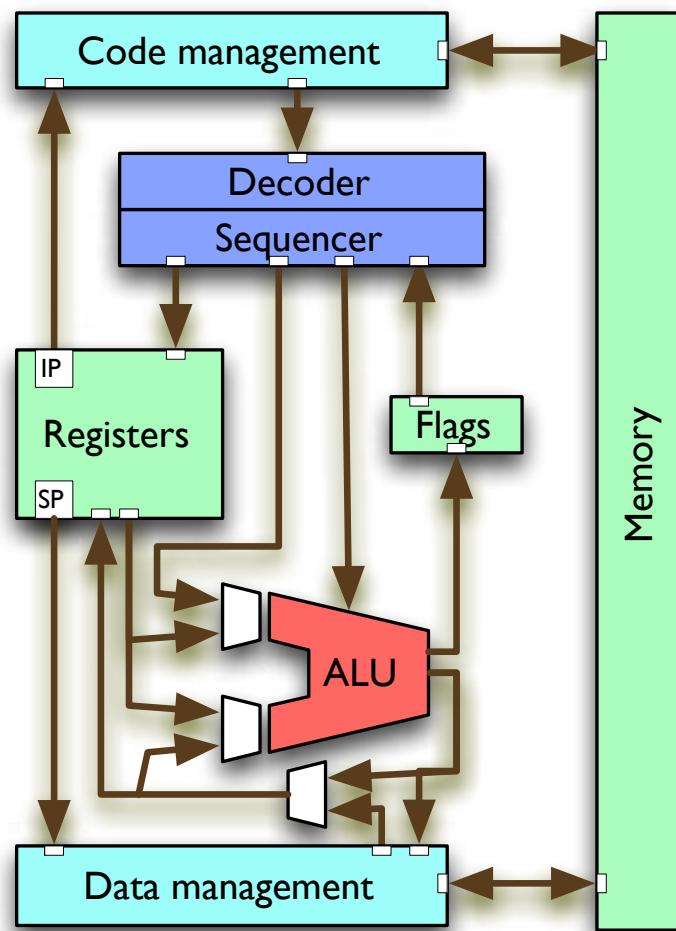


Your controller has many counters which can e.g. be used to delay operations without the need to execute instructions.



Digital Logic

Simple CPU Architecture



You can already build most of the components of a CPU by now.

(The most essential missing component is the **sequencer** which is a specialized state-machine.)

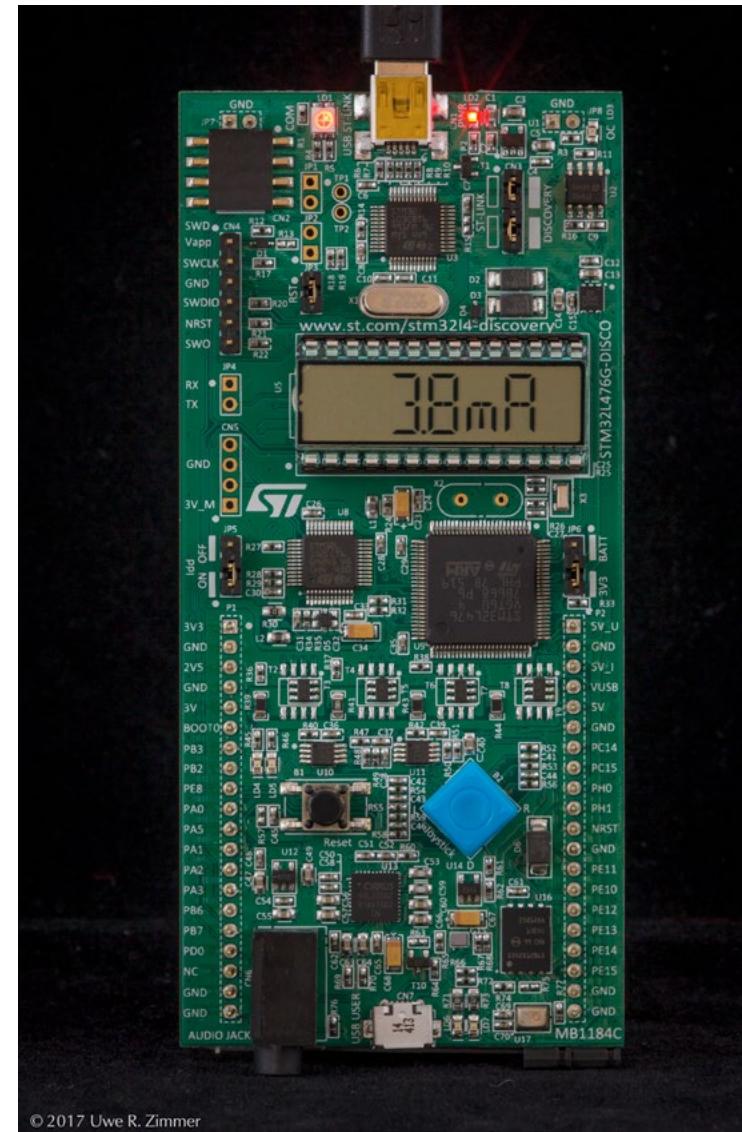
We will come back to the CPU architectures towards the end of the course.

☞ The next chapter will be about programming a CPU at machine level.



Digital Logic

STM32L476 Discovery

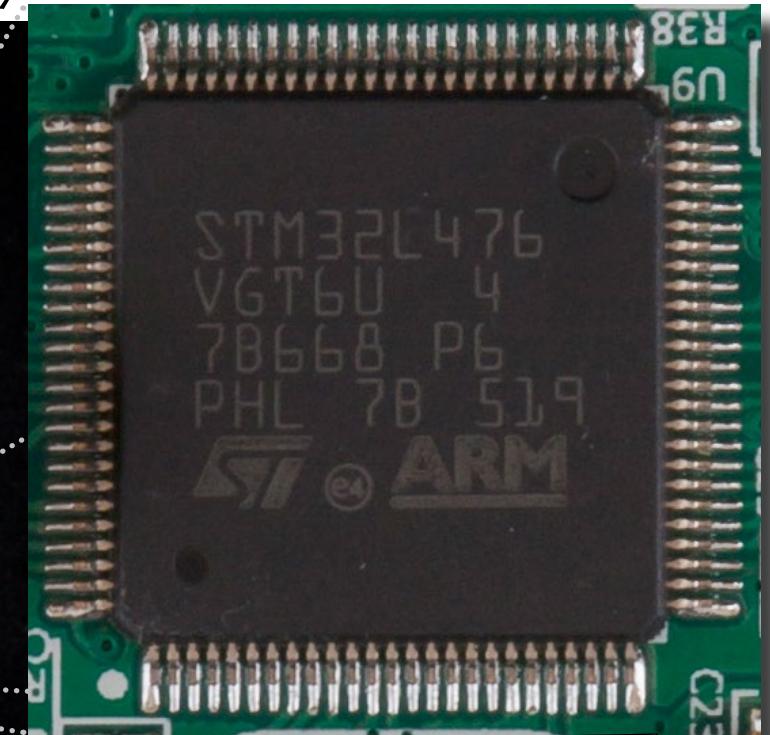
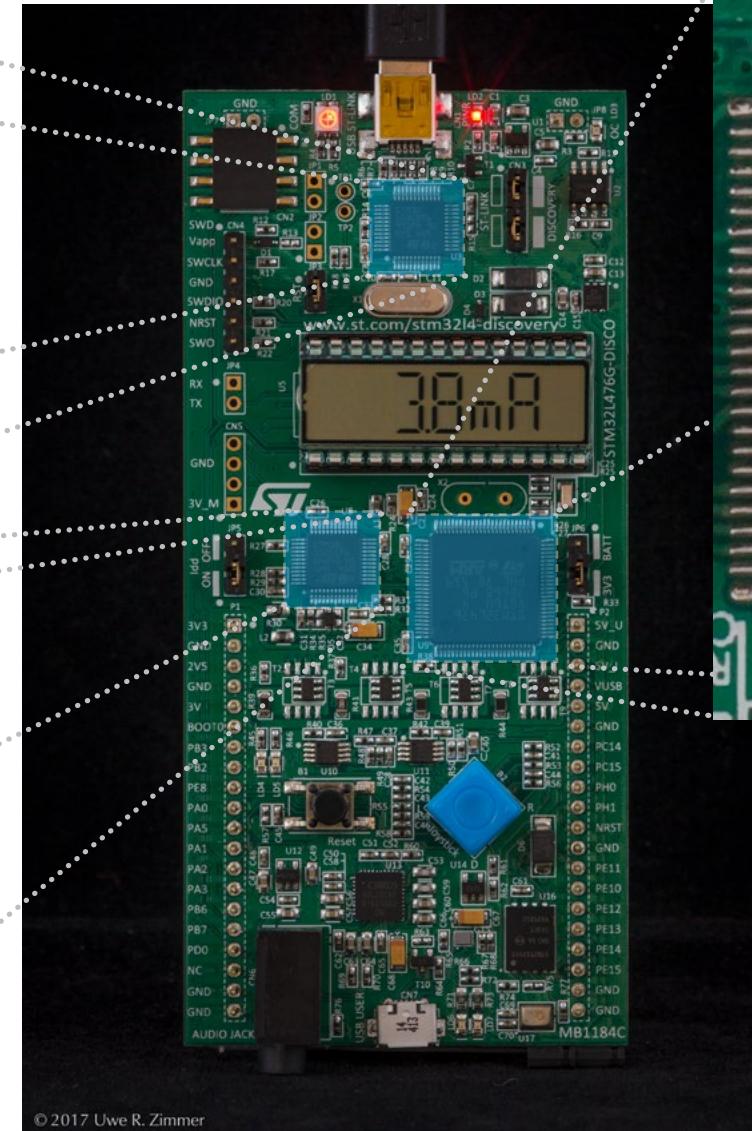
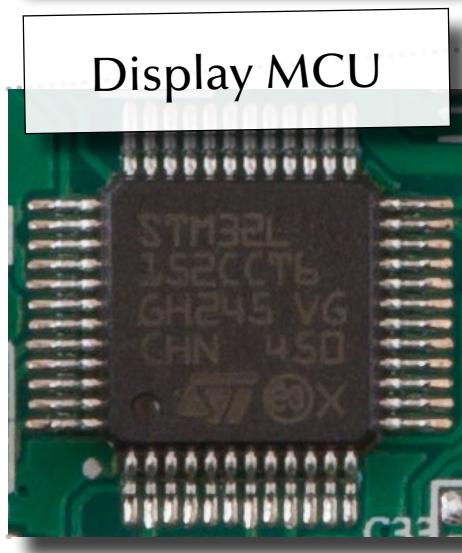
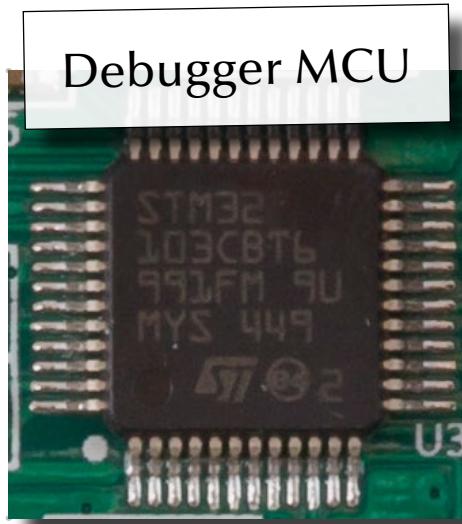


© 2017 Uwe R. Zimmer



Digital Logic

STM32L476 Discovery

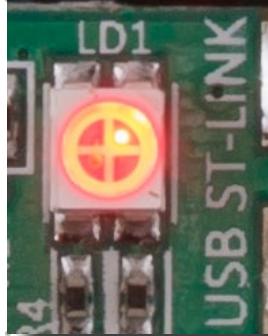


Main MCU

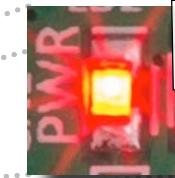


Digital Logic

STM32L476 Discovery

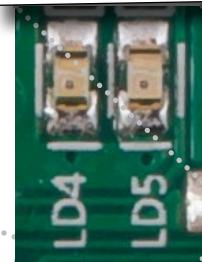


Debugger state

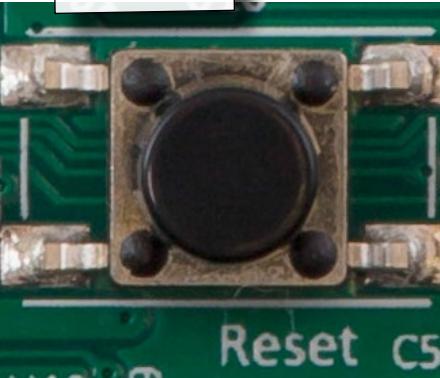


Power

User LEDs

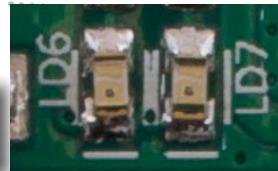


Reset

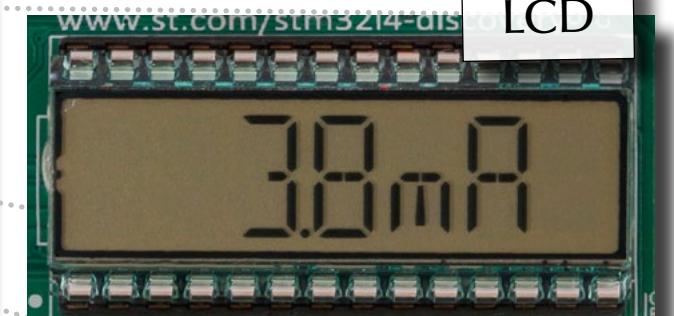


Reset C5

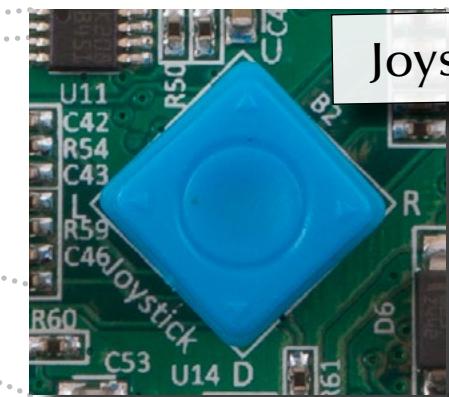
OTG LEDs



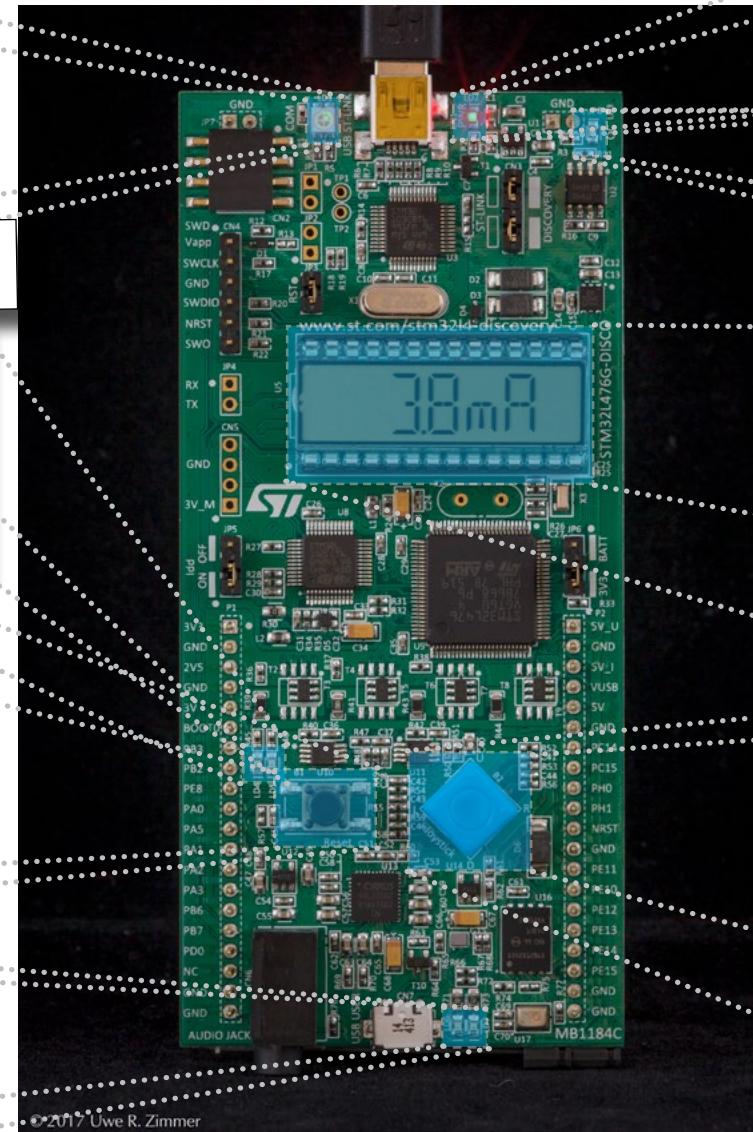
Over current



LCD



Joystick



© 2017 Uwe R. Zimmer



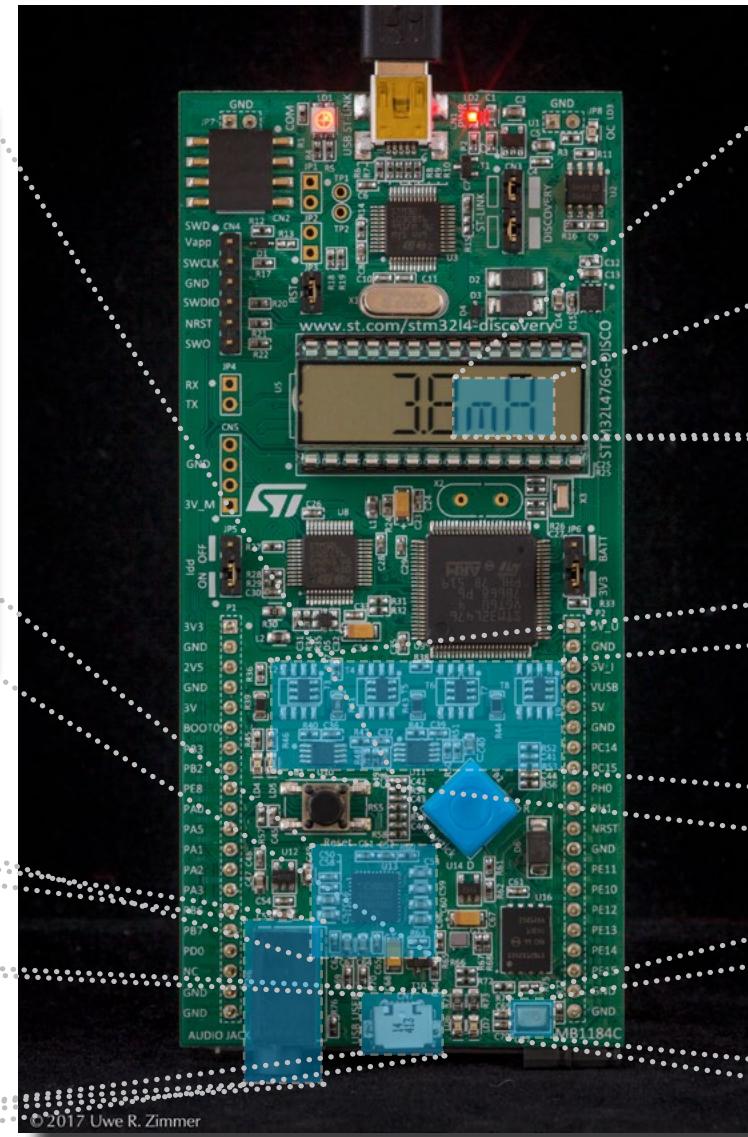
Digital Logic

STM32L476 Discovery

Multiplexed 24 bit
 $\Sigma\Delta$ -DAConverter with
stereo power amp



Headphone jack



“9 axis” motion sensor
(underneath display):

- 3 axis accelerometer
- 3 axis gyroscope
- 3 axis magnetometer

Current meter to MCU
60 nA ... 50 mA



Microphone

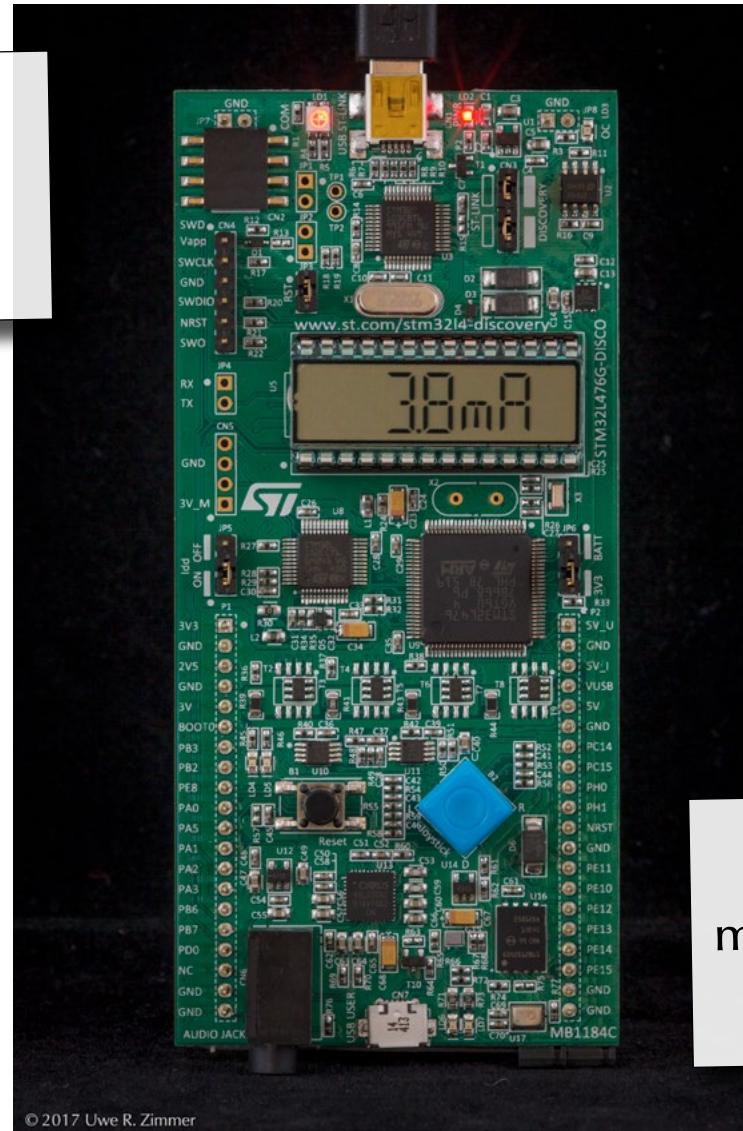




Digital Logic

STM32L476 Discovery

There is a lot more hardware here than you could possibly master in one semester ...



© 2017 Uwe R. Zimmer

... and you will master a lot more about CPUs at the end the course than you think now.



Digital Logic

Summary

Digital Logic

- **Boolean Algebra**
 - Truth tables and Boolean operations
 - Minterms and simplifying expressions
- **Combinational Logic**
 - Logic gates
 - Numbers
 - Adders, ALU
- **State-oriented Logic**
 - Flip-Flops, registers and counters
- **CPU Architecture**

Computer Organisation & Program Execution 2021



2

Hardware/Software Interface

Uwe R. Zimmer - The Australian National University



Hardware/Software Interface

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

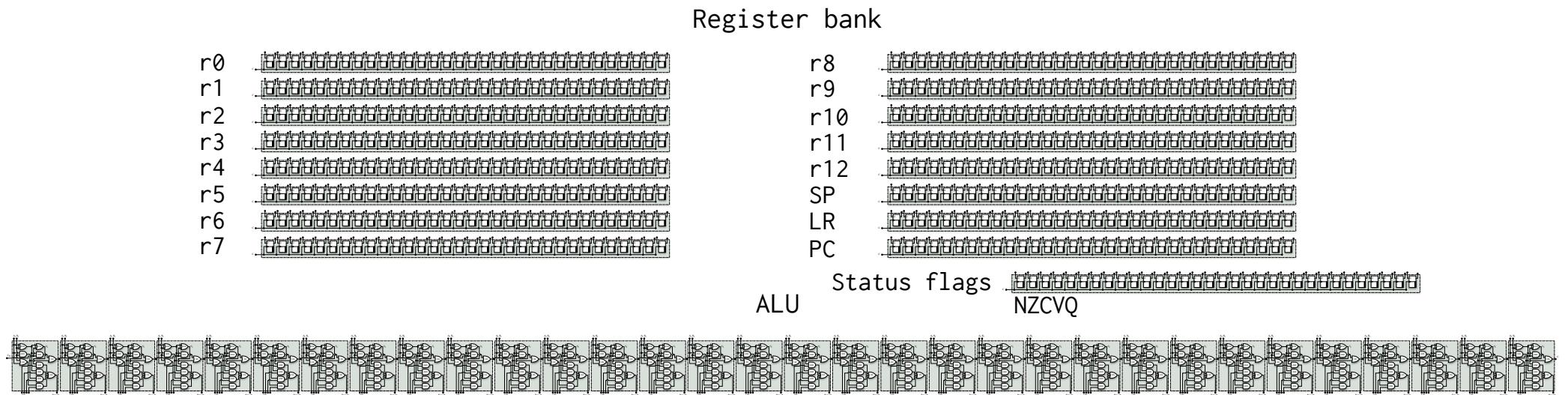
Chapter 2 “Instructions: Language of the Computer” & Chapter 3 “Arithmetic for Computers”

ARM edition, Morgan Kaufmann 2017



Hardware/Software Interface

Adding the value of two registers



The CPU will fetch the content of the memory cell which PC is pointing to.

👉 We want the CPU to execute:

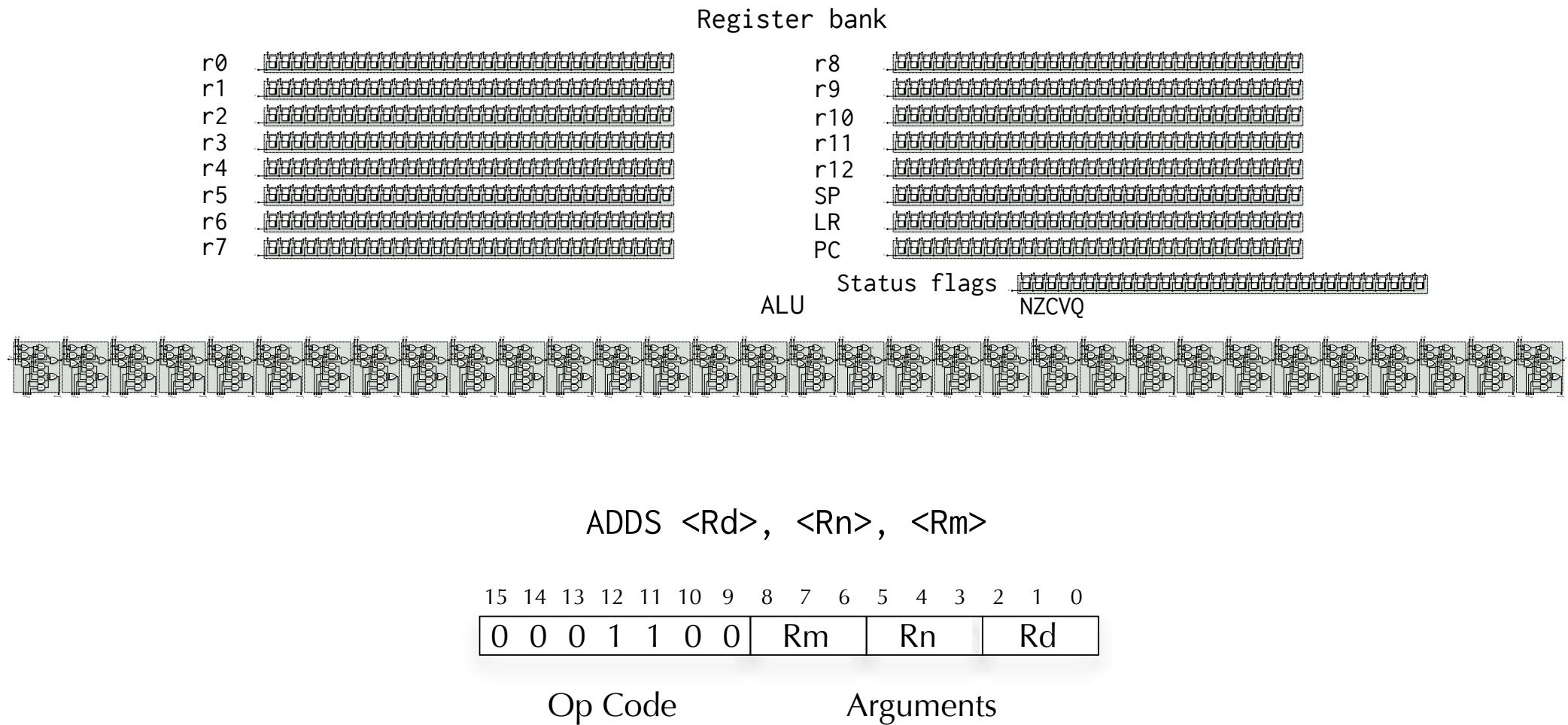
$$r4 := r2 + r3$$

👉 What to store in this memory cell?



Hardware/Software Interface

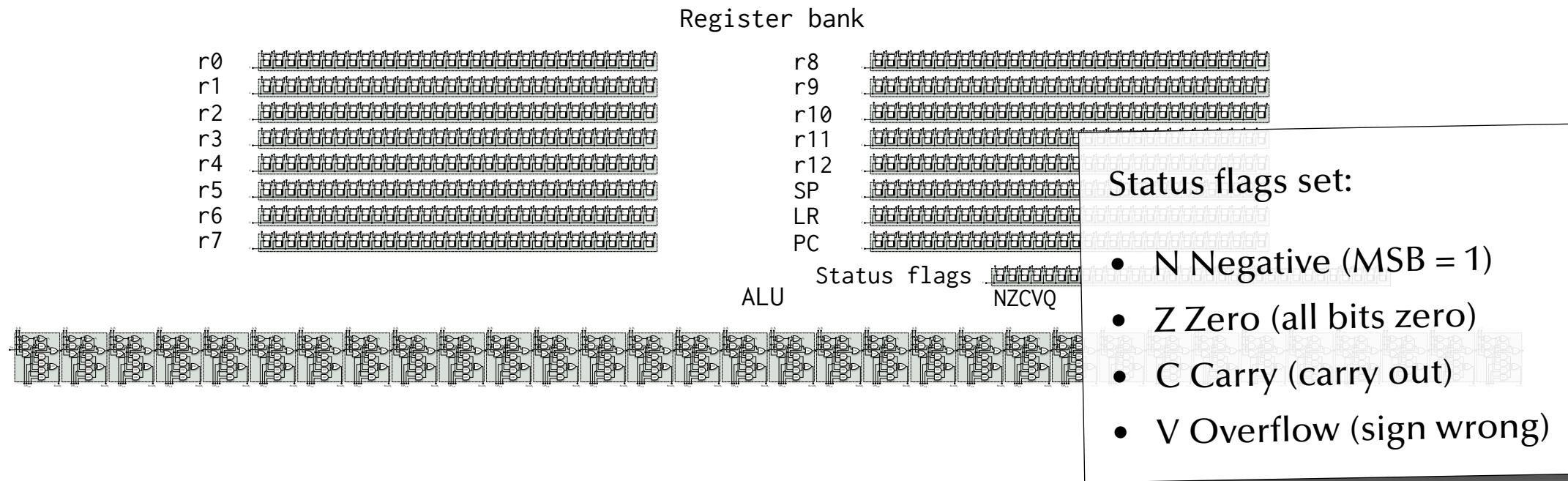
Adding the value of two registers





Hardware/Software Interface

Adding the value of two registers



ADDS r4, r2, r3

Assembler

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	1	1	0	1	0	1	0	0
16#18#								16#D4#							

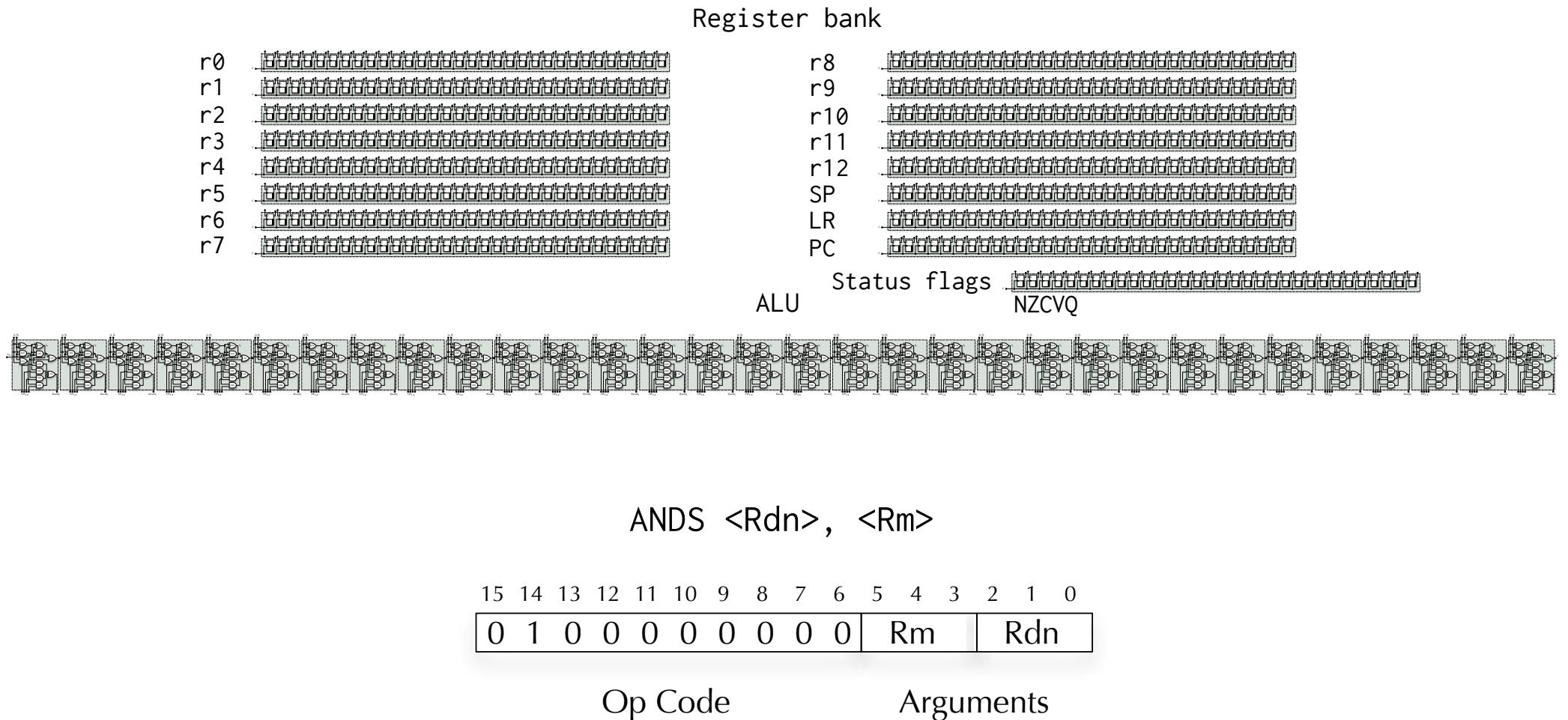
Disassembler

r4 := r2 + r3



Hardware/Software Interface

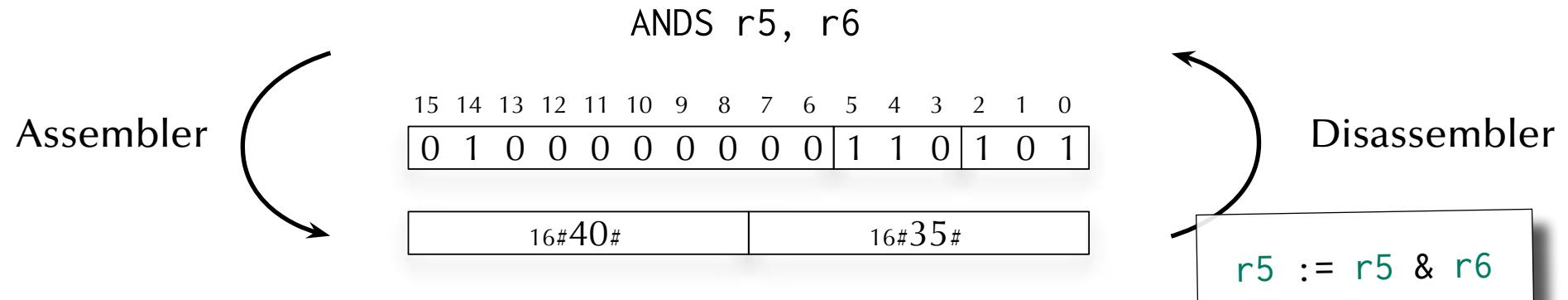
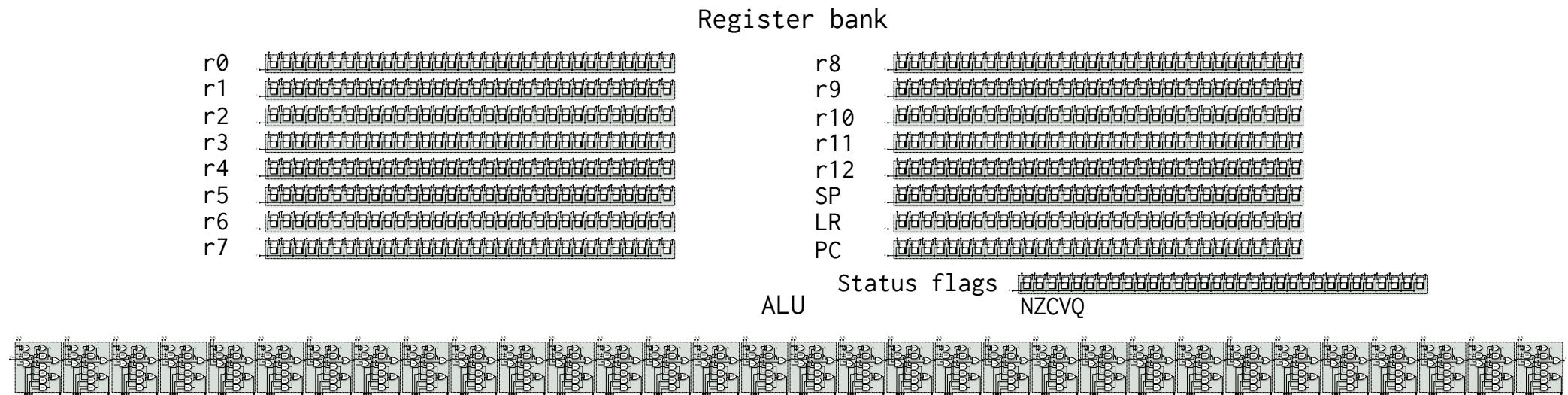
Adding the value of two registers





Hardware/Software Interface

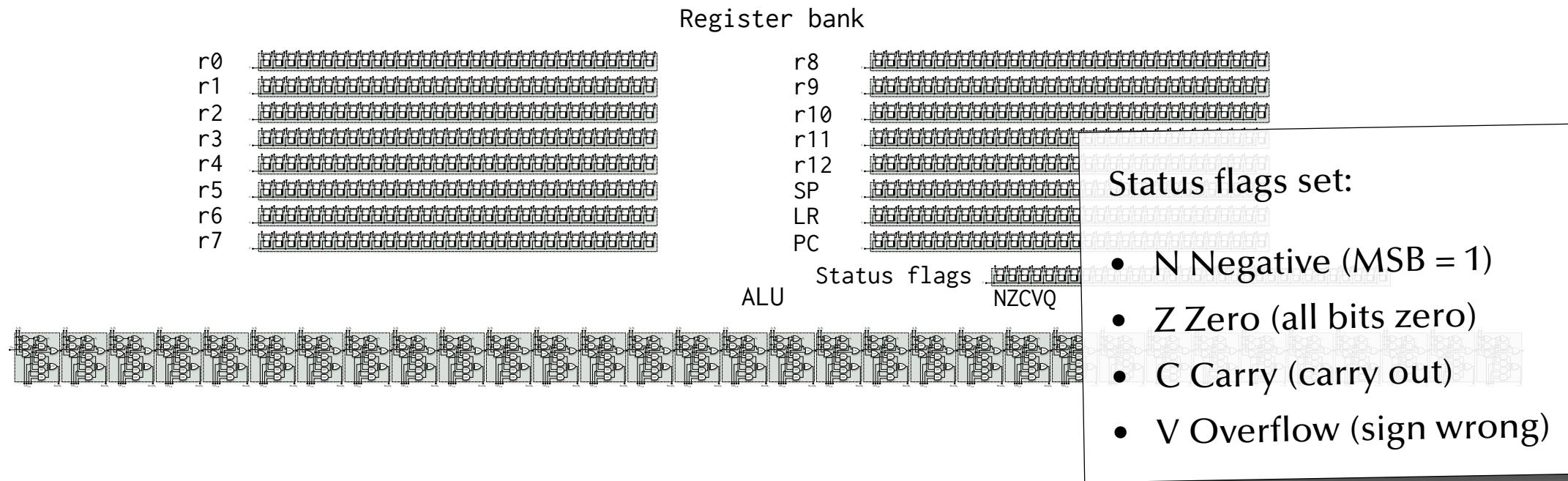
Adding the value of two registers





Hardware/Software Interface

Adding the value of two registers



ADDS r4, r2, r3

Assembler

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	1	1	0	1	0	1	0	0
16#18#								16#D4#							

Disassembler

r4 := r2 + r3



Hardware/Software Interface

ARM v7-M 32 bit add instructions

add{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}

adc{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}

add{s}<c><q> {<Rd>, } <Rn>, #<const>

adc{s}<c><q> {<Rd>, } <Rn>, #<const>

qadd<c><q> {<Rd>, } <Rn>, <Rm>

adds r1, r4, r5

add r1, #1

adcs r1, r4

qadd r1, r4, r5

s: **sets the flags** based on the result

c: makes the command **conditional**. <c> can be EQ (equal), NE (not equal), CS (carry set), CC (carry clear), MI (minus), PL (plus), VS (overflow set), VC (overflow clear), HI (unsigned higher), LS (unsigned lower or same), GE (signed greater or equal), LT (signed less), GT (signed greater), LE (signed less or equal), AL (always)

q: **instruction width**. Can be .N for narrow (16 bit) or .W for wide (32 bit)

Rd, Rn, Rm: **any register**, incl. SP, LR and PC (with some restrictions). Result goes to Rn (if no Rd).

shift: value of Rm is **preprocessed** with LSL (logical shift left – fills zeros), LSR (logical shift right – fills zeros), ASR (arithmetic shift right – keeps sign) or ROR (rotate right) followed by the #number of bits to shift/rotate by. There is also a RRX (rotate right by one incl. carry flag)

const: an **immediate value** in the range 0..4095 directly or in the range 0..255 with rotation.



Hardware/Software Interface

ARM v7-M 32 bit add instructions

add{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}
adc{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}
add{s}<c><q> {<Rd>, } <Rn>, #<const>
adc{s}<c><q> {<Rd>, } <Rn>, #<const>
qadd<c><q> {<Rd>, } <Rn>, <Rm>

Any of those instructions requires exactly *one* CPU cycle (in terms of throughput).

“Reduced Instruction Set Computing (RISC)”

s: **sets the flags** based on the result

c: makes the command **conditional**. <c> can be EQ (equal), NE (not equal), CS (carry set), CC (carry clear), MI (minus), PL (plus), VS (overflow set), VC (overflow clear), HI (unsigned higher), LS (unsigned lower or same), GE (signed greater or equal), LT (signed less), GT (signed greater), LE (signed less or equal), AL (always)

q: **instruction width**. Can be .N for narrow (16 bit) or .W for wide (32 bit)

Rd, Rn, Rm: **any register**, incl. SP, LR and PC (with some restrictions). Result goes to Rn (if no Rd).

shift: value of Rm is **preprocessed** with LSL (logical shift left – fills zeros), LSR (logical shift right – fills zeros), ASR (arithmetic shift right – keeps sign) or ROR (rotate right) followed by the #number of bits to shift/rotate by. There is also a RRX (rotate right by one incl. carry flag)

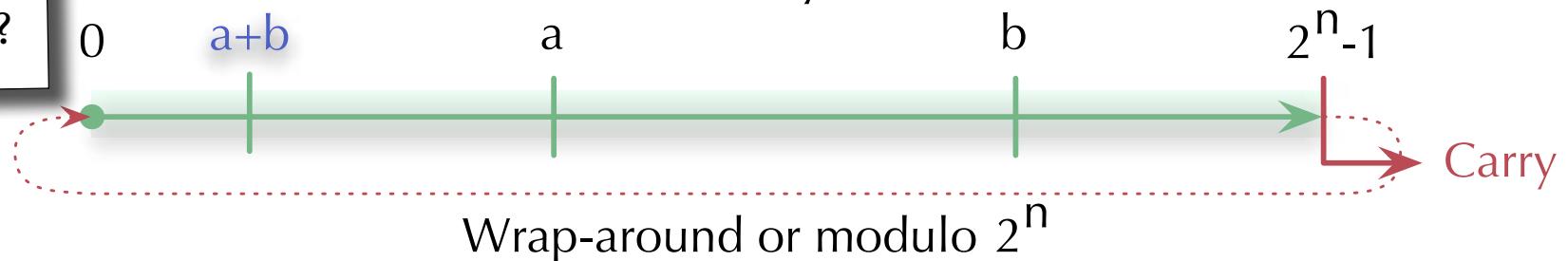
const: an **immediate value** in the range 0..4095 directly or in the range 0..255 with rotation.



Hardware/Software Interface

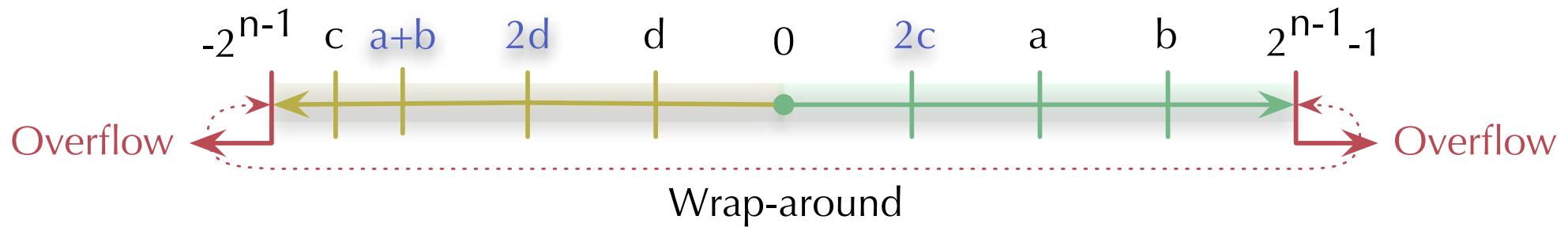
Numeric CPU status flags

Which of those operations will set which flag?

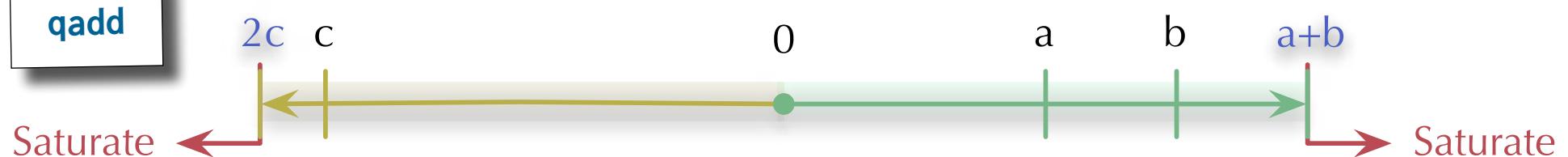


adds
adcs

2's complement binary numbers



qadd





Hardware/Software Interface

ARM v7-M 32 bit Addition, Subtraction instructions

add{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}	; $Rd := Rn + Rm(\text{shifted})$
adc{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}	; $Rd := Rn + Rm(\text{shifted}) + C$
add{s}<c><q> {<Rd>, } <Rn>, #<const>	; $Rd := Rn + \#<\text{const}>$
adc{s}<c><q> {<Rd>, } <Rn>, #<const>	; $Rd := Rn + \#<\text{const}> + C$
qadd<c><q> {<Rd>, } <Rn>, <Rm>	; $Rd := Rn + Rm$; saturated
sub{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}	; $Rd := Rn - Rm(\text{shifted})$
sbc{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}	; $Rd := Rn - Rm(\text{shifted}) - \text{NOT}(C)$
rsb{s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>}	; $Rd := Rm(\text{shifted}) - Rn$
sub{s}<c><q> {<Rd>, } <Rn>, #<const>	; $Rd := Rn - \#<\text{const}>$
sbc{s}<c><q> {<Rd>, } <Rn>, #<const>	; $Rd := Rn - \#<\text{const}> - \text{NOT}(C)$
rsb{s}<c><q> {<Rd>, } <Rn>, #<const>	; $Rd := \#<\text{const}> - Rn$
qsub<c><q> {<Rd>, } Rn, Rm	; $Rd := Rn - Rm$; saturated

All instructions operate on 32 bit wide numbers.

... versions for narrower numbers, as well as versions which operate on multiple narrower numbers in parallel exist as well.



Hardware/Software Interface

64 bit Addition, Subtraction

As your registers are 32 bit wide, you need two steps to add two 64 bit numbers in r3:r2, r5:r4 (with r2 and r4 being the lower 32 bits) to one 64 bit number in r1:r0:

```
adds r0, r2, r4 ; r0 := r2 + r4      add least significant words, set flags  
adcs r1, r3, r5 ; r1 := r3 + r5 + C  add most significant words and carry bit
```

... and symmetrically if you need a 64 bit subtraction:

```
subs r0, r2, r4 ; r0 := r2 - r4      least significant words, set flags  
sbcs r1, r3, r5 ; r1 := r3 - r5 - NOT (C)  most significant words and carry bit
```



Hardware/Software Interface

ARM v7-M 32 bit Boolean (bit-wise) instructions

and {s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>} ;	$Rd := Rn \wedge \underline{Rm^{\text{shifted}}}$
bic {s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>} ;	$Rd := Rn \wedge \underline{Rm^{\text{shifted}}}$
orr {s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>} ;	$Rd := Rn \vee \underline{Rm^{\text{shifted}}}$
orn {s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>} ;	$Rd := Rn \vee \underline{Rm^{\text{shifted}}}$
eor {s}<c><q> {<Rd>, } <Rn>, <Rm> {,<shift>} ;	$Rd := Rn \oplus \underline{Rm^{\text{shifted}}}$
and {s}<c><q> {<Rd>, } <Rn>, #<const>	$; Rd := Rn \wedge const$
bic {s}<c><q> {<Rd>, } <Rn>, #<const>	$; Rd := Rn \wedge \overline{const}$
orr {s}<c><q> {<Rd>, } <Rn>, #<const>	$; Rd := Rn \vee const$
orn {s}<c><q> {<Rd>, } <Rn>, #<const>	$; Rd := Rn \vee \overline{const}$
eor {s}<c><q> {<Rd>, } <Rn>, #<const>	$; Rd := Rn \oplus const$
cmp <c><q> <Rn>, <Rm> {,<shift>} ;	$; (Rn - Rm^{\text{shifted}}) \rightarrow Flags$
cmn <c><q> <Rn>, <Rm> {,<shift>} ;	$; (Rn + Rm^{\text{shifted}}) \rightarrow Flags$
tst <c><q> <Rn>, <Rm> {,<shift>} ;	$; (Rn \wedge Rm^{\text{shifted}}) \rightarrow Flags$
teq <c><q> <Rn>, <Rm> {,<shift>} ;	$; (Rn \oplus Rm^{\text{shifted}}) \rightarrow Flags$
cmp <c><q> <Rn>, #<const>	$; (Rn - const) \rightarrow Flags$
cmn <c><q> <Rn>, #<const>	$; (Rn + const) \rightarrow Flags$
tst <c><q> <Rn>, #<const>	$; (Rn \wedge const) \rightarrow Flags$
teq <c><q> <Rn>, #<const>	$; (Rn \oplus const) \rightarrow Flags$

This exhausts
the simple
ALU from
chapter 1 ...



Hardware/Software Interface

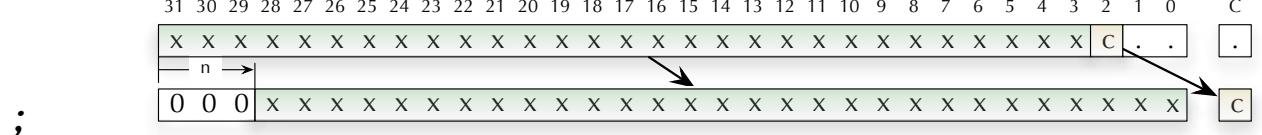
ARM v7-M Move data inside the CPU

mov{s}<c><q> <Rd>, <Rm> ; $Rd := Rm$

mov{s}<c><q> <Rd>, #<const> ; $Rd := const$

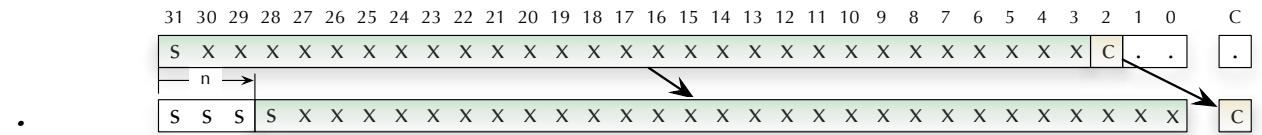
lsr{s}<c><q> <Rd>, <Rm>, #<n>

lsr{s}<c><q> <Rd>, <Rm>, <Rs>



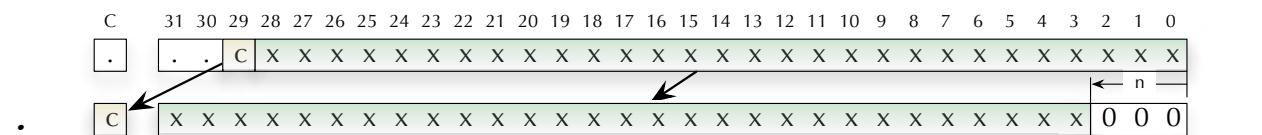
asr{s}<c><q> <Rd>, <Rm>, #<n>

asr{s}<c><q> <Rd>, <Rm>, <Rs>



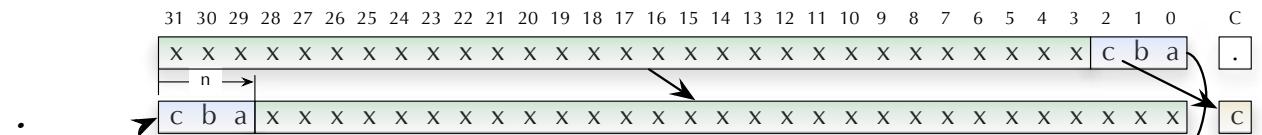
lsl{s}<c><q> <Rd>, <Rm>, #<n>

lsl{s}<c><q> <Rd>, <Rm>, <Rs>

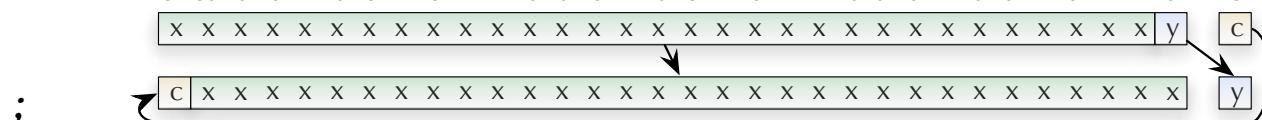


ror{s}<c><q> <Rd>, <Rm>, #<n>

ror{s}<c><q> <Rd>, <Rm>, <Rs>



rrx{s}<c><q> <Rd>, <Rm>





Hardware/Software Interface

ARM v7-M Move data inside the CPU

mov{s}<c><q> <Rd>, <Rm>

; $Rd := Rm$

mov{s}<c><q> <Rd>, #<const>

; $Rd := const$

lsr{s}<c><q> <Rd>, <Rm>, #<n>

lsr{s}<c><q> <Rd>, <Rm>, <Rs>

;

asr{s}<c><q> <Rd>, <Rm>, #<n>

asr{s}<c><q> <Rd>, <Rm>, <Rs>

;

lsl{s}<c><q> <Rd>, <Rm>, #<n>

lsl{s}<c><q> <Rd>, <Rm>, <Rs>

;

ror{s}<c><q> <Rd>, <Rm>, #<n>

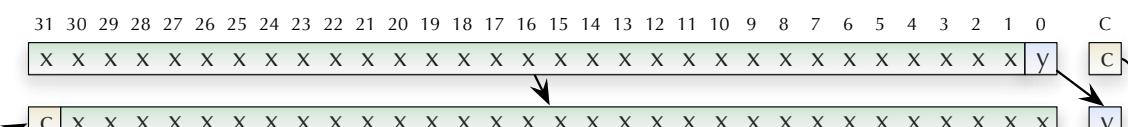
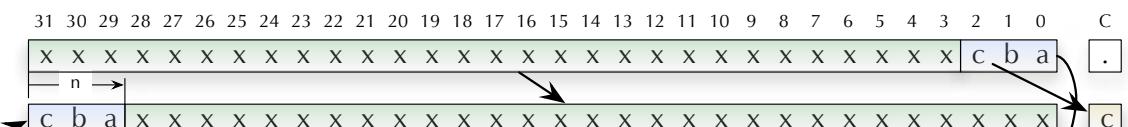
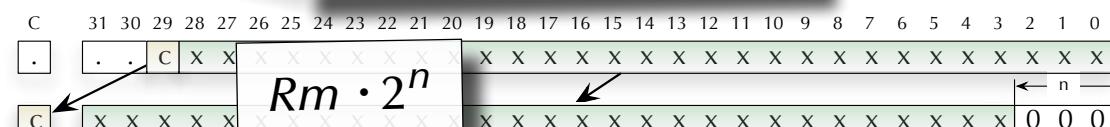
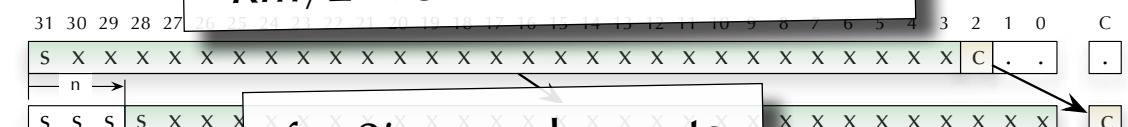
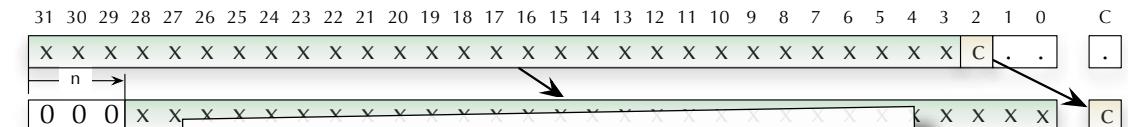
ror{s}<c><q> <Rd>, <Rm>, <Rs>

;

rrx{s}<c><q> <Rd>, <Rm>

;

If this is numbers then ...





Hardware/Software Interface

Simple arithmetic inside the CPU

Calculate:

$e := a + b - 2*c$

assuming all types are 32 bit 2's complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.



Hardware/Software Interface

Simple arithmetic inside the CPU

Calculate:

$e := a + b - 2*c$

assuming all types are 32 bit 2's complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

```
add  r5, r1, r2
lsl  r6, r3, #1      ; you could also write: mov r6, r3, lsl #1
sub  r4, r5, r6
```

We need temporary storage (r5, r6) in the process as we didn't want to overwrite the original values. Yet the total number of registers is always limited.



Hardware/Software Interface

Simple arithmetic inside the CPU

Calculate:

$e := a + b - 2*c$

assuming all types are 32 bit 2's complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

```
add  r5, r1, r2
lsl  r6, r3, #1      ; you could also write: mov r6, r3, lsl #1
sub  r4, r5, r6
```

We need temporary storage (r5, r6) in the process as we didn't want to overwrite the original values. Yet the total number of registers is always limited.

How about we assume that values are no longer needed after this expression:



Hardware/Software Interface

Simple arithmetic inside the CPU

Calculate:

```
e := a + b - 2*c
```

assuming all types are 32 bit 2's complement numbers (Integer),
r1 holds a, r2 holds b, r3 holds c, and the results should be in r4.

```
add  r5, r1, r2
lsl  r6, r3, #1      ; you could also write: mov r6, r3, lsl #1
sub  r4, r5, r6
```

We need temporary storage (r5, r6) in the process as we didn't want to overwrite the original values. Yet the total number of registers is always limited.

How about we assume that values are no longer needed after this expression:

```
add  r1, r1, r2
lsl  r3, r3, #1
sub  r4, r1, r3
```

... your compiler will know when such side-effects are ok and when not.

☞ Any overflows?



Hardware/Software Interface

Simple arithmetic inside the CPU

Calculate:

$e := a + b - 2*c$

We need to check results after each step:

```
adds r1, r1, r2 ; need to check overflow flag  
lsl  r3, r3, #1  ; need to check that the sign did not change  
subs r4, r1, r3 ; need to check overflow flag again
```

- ☞ We don't have the means yet to branch off into different actions in case things go bad ... to come soon.



Hardware/Software Interface

Simple arithmetic inside the CPU

Calculate:

```
e := a + b - 2*c
```

We need to check results after each step:

```
adds r1, r1, r2 ; need to check overflow flag  
lsl  r3, r3, #1  ; need to check that the sign did not change  
subs r4, r1, r3 ; need to check overflow flag again
```

☞ We don't have the means yet to branch off into different actions in case things go bad ... to come soon.

Or we use saturation arithmetic and live with the error:

```
qadd r1, r1, r2  
qadd r3, r3, r3  
qsub r4, r1, r3
```

☞ If we know we need to carry on either way, this at least minimizes the local errors.



Hardware/Software Interface

Cortex-M4 Address Space

Your CPU has 32 bit of address space

☞ 4 GB

... address space does not equate to physical memory!

Not all memory is equal: Some memory ...

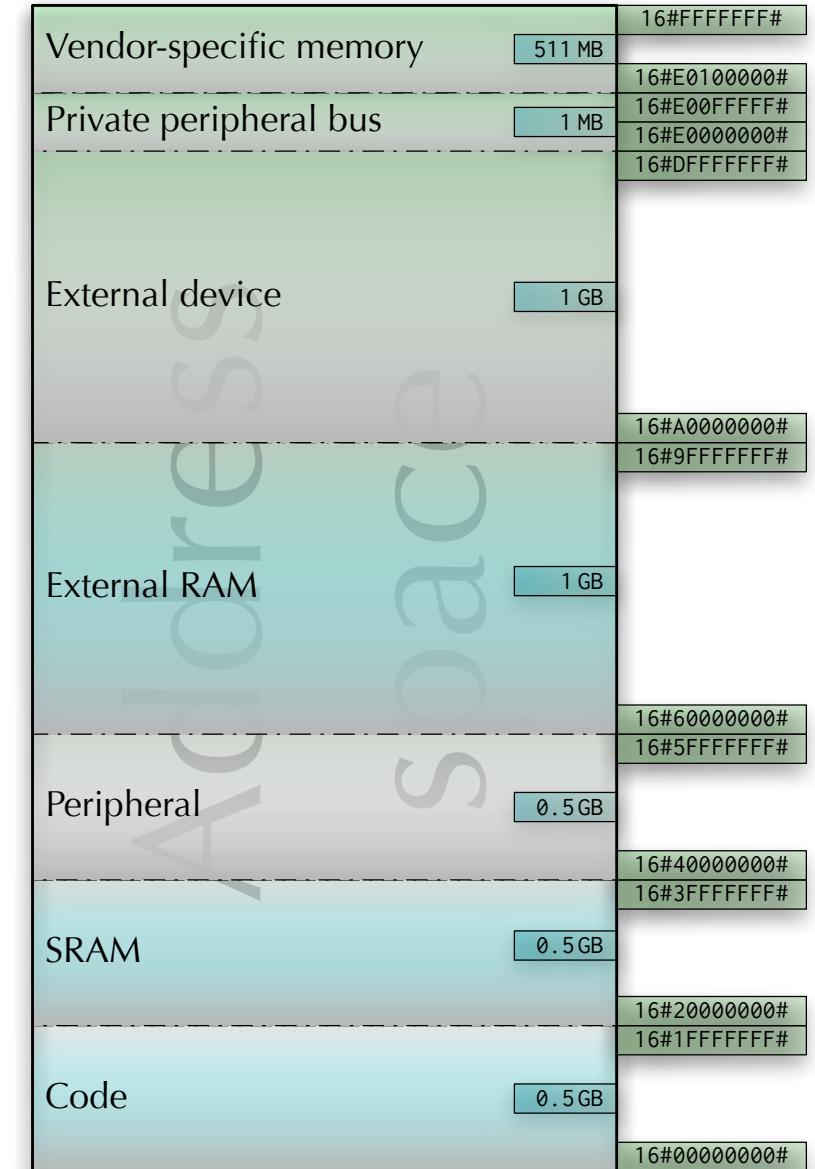
... can be **executed**

... can be **written to or read from or both**

... has **side-effects** (coffee cups fall over)

... has **strictly-ordered access**

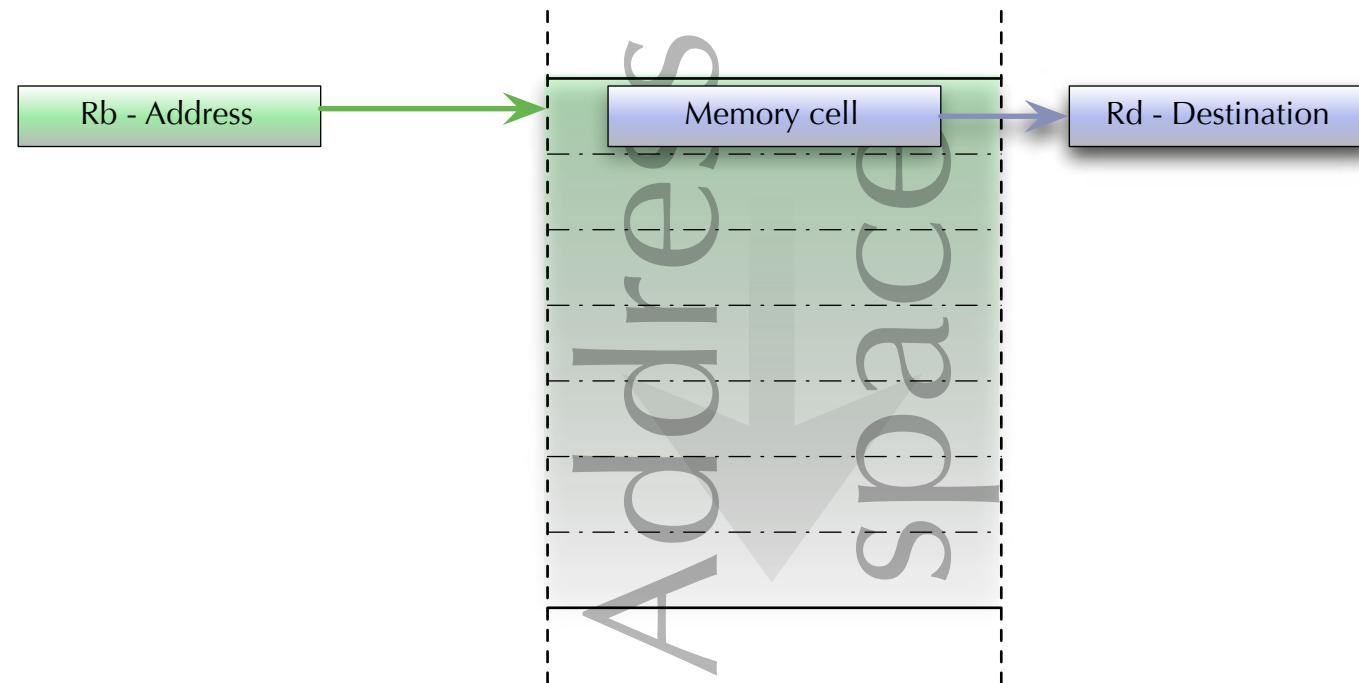
... **does not physically exist**





Hardware/Software Interface

ARM v7-M Copy data in and out of the CPU

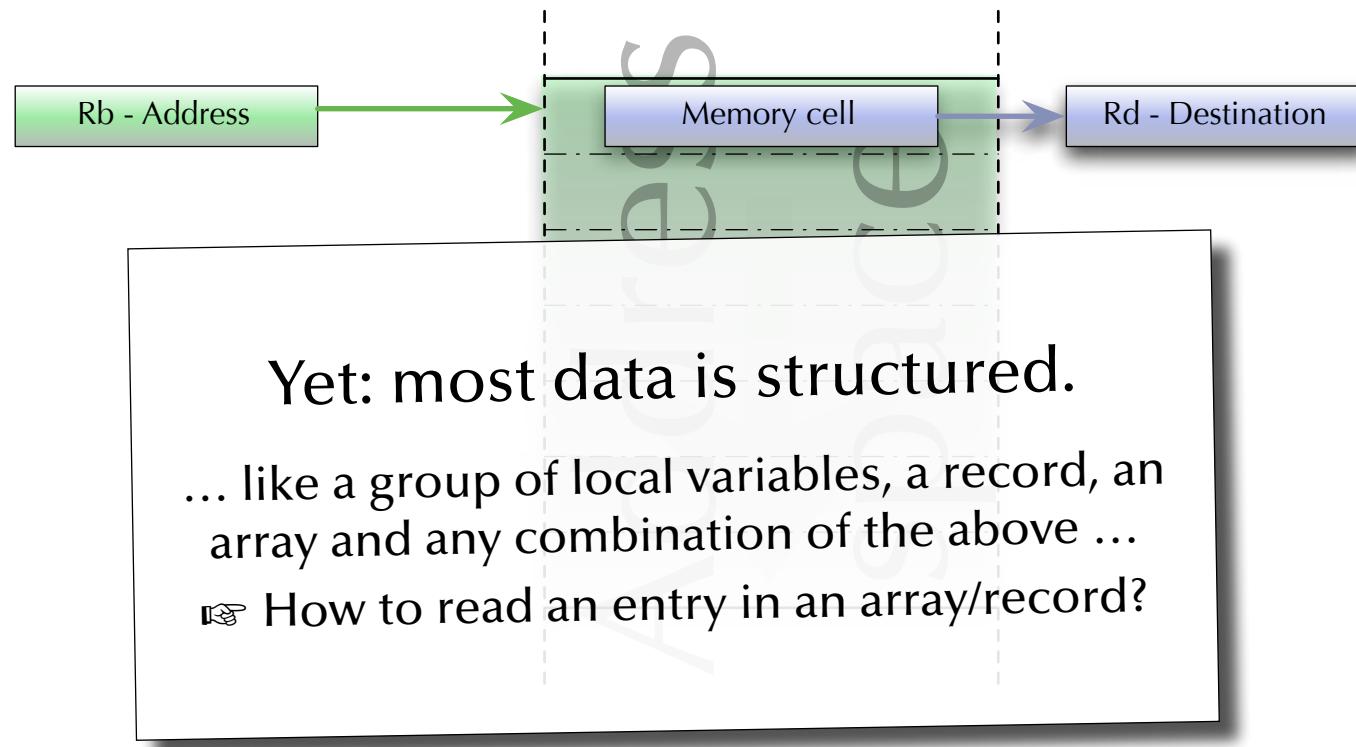


In its most basic form the value of a register is interpreted as an **address** and the **memory content** there is loaded into another register.



Hardware/Software Interface

ARM v7-M Copy data in and out of the CPU

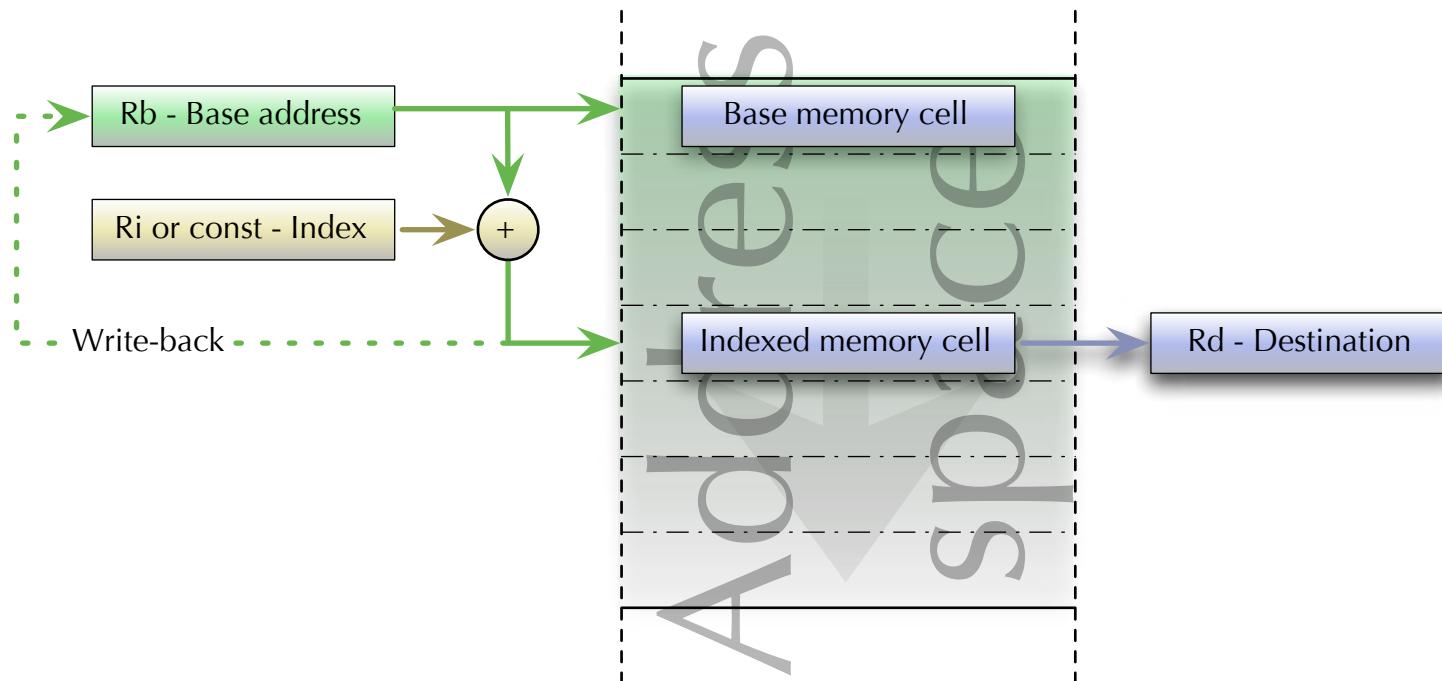


In its most basic form the value of a register is interpreted as an **address** and the **memory content** there is loaded into another register.



Hardware/Software Interface

ARM v7-M Copy data in and out of the CPU

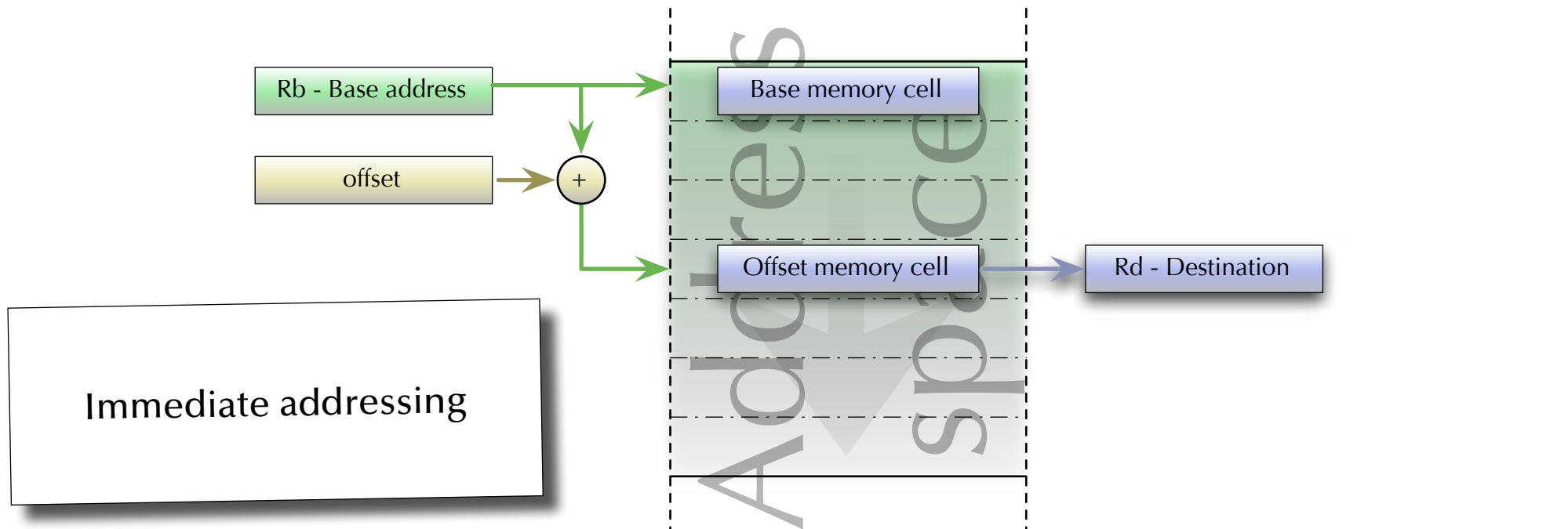


Most copy operations between CPU and memory follow this basic scheme.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



ldr<c><q> <Rd>, [<Rb> {, #+/-<offset>}]
str<c><q> <Rs>, [<Rb> {, #+/-<offset>}]

ldr r1, [r4]

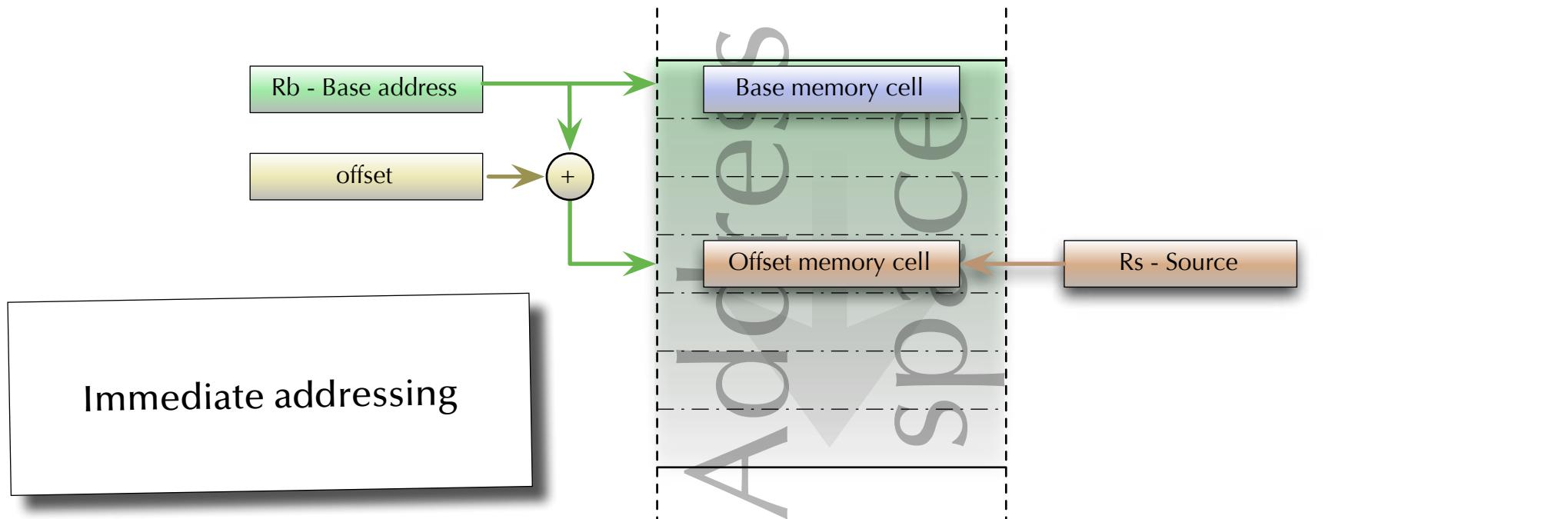
ldr r1, [r4, #8]

Reads from a potentially offset memory cell with a base register address.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



ldr<c><q> <Rd>, [<Rb> {, #+/-<offset>}]
str<c><q> <Rs>, [<Rb> {, #+/-<offset>}]

str r1, [r4]

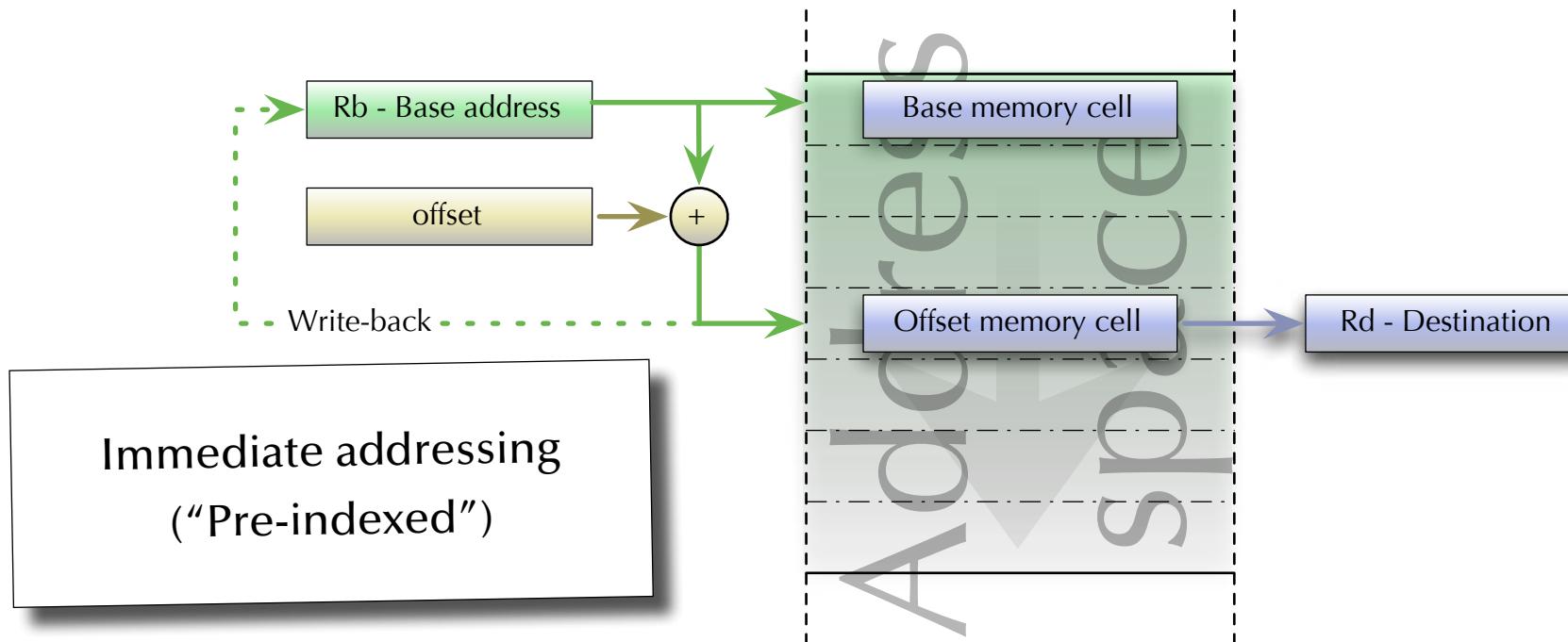
str r1, [r4, #-12]

Writes to a potentially offset memory cell with a base register address.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



ldr<c><q> <Rd>, [<Rb>, #+/-<offset>]!
str<c><q> <Rs>, [<Rb>, #+/-<offset>]!

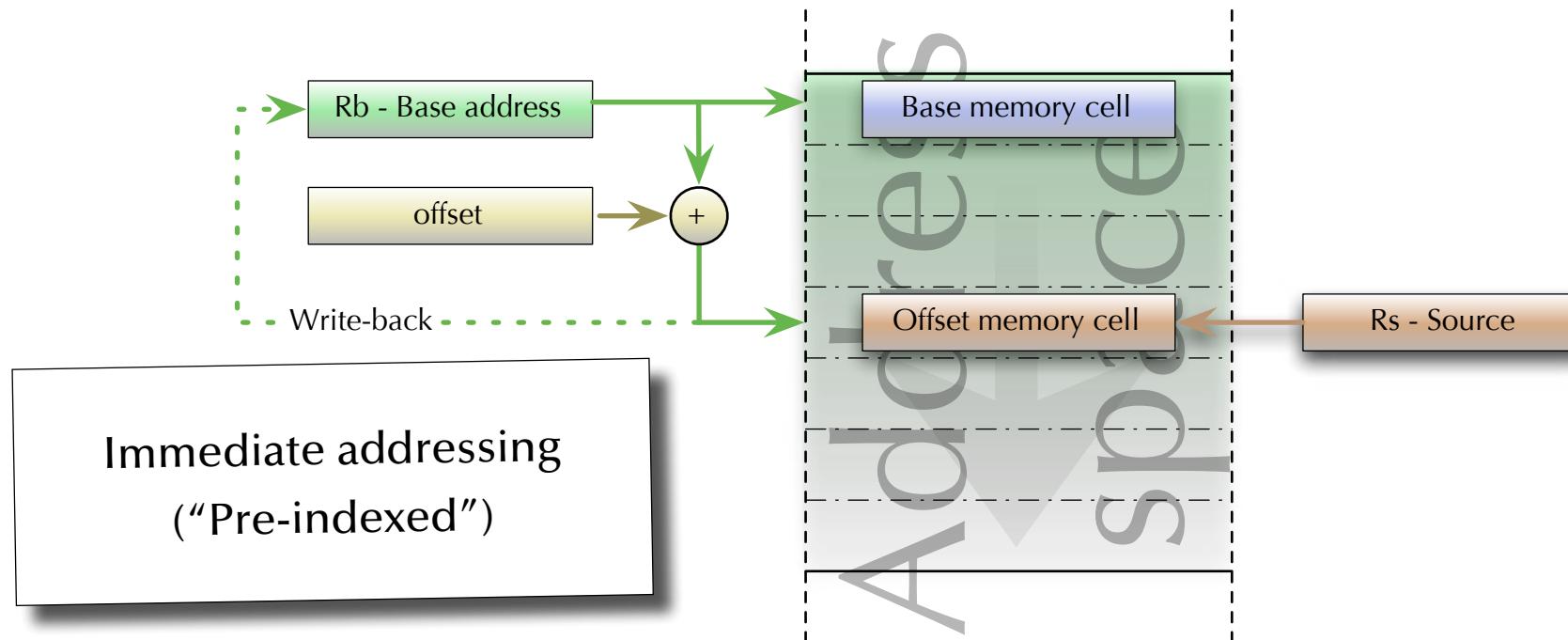
ldr r1, [r4, #8]!

Reads from an offset memory cell with a base register address and writes the offset address back into the original base register.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



ldr<c><q> <Rd>, [<Rb>, #+/-<offset>]!
str<c><q> <Rs>, [<Rb>, #+/-<offset>]!

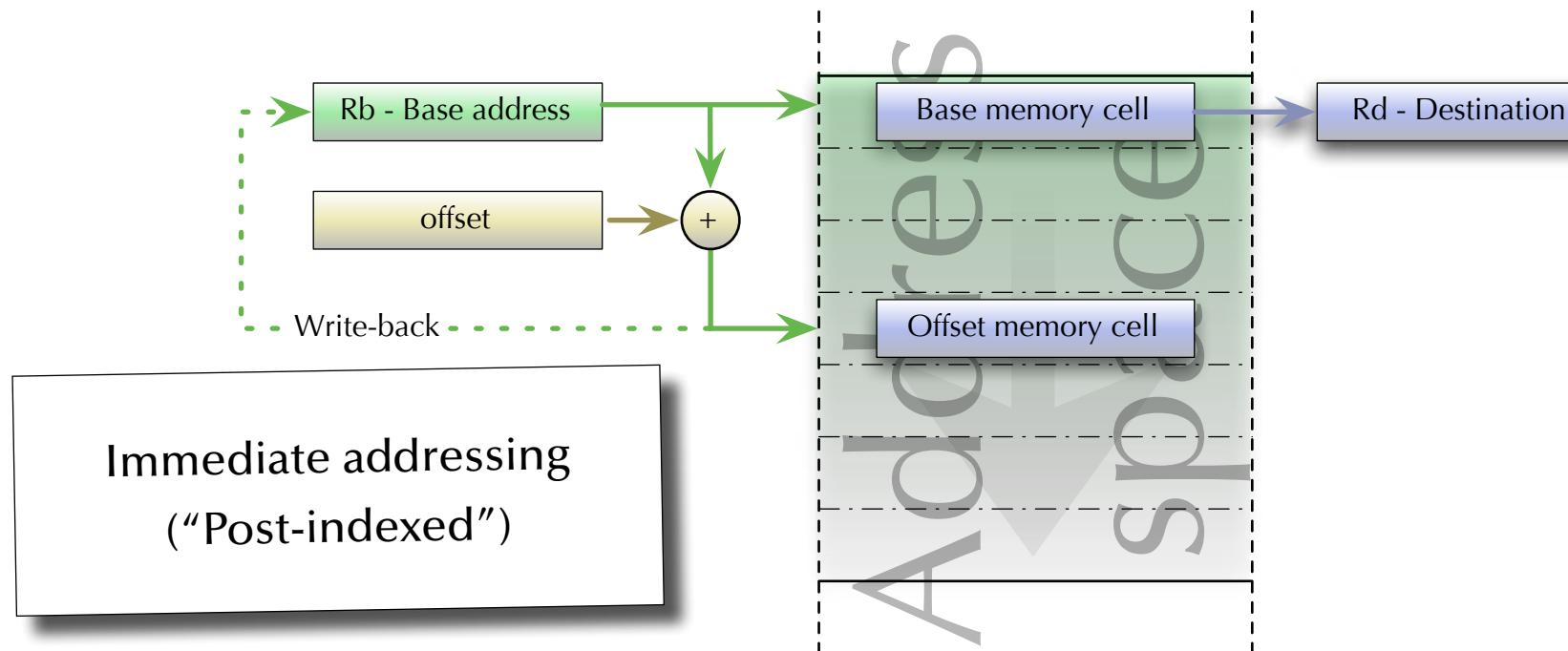
str r1, [r4, #-12]!

Writes to an offset memory cell with a base register address
and writes the offset address back into the original base register.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



ldr<c><q> <Rd>, [<Rb>], #+/-<offset>
str<c><q> <Rs>, [<Rb>], #+/-<offset>

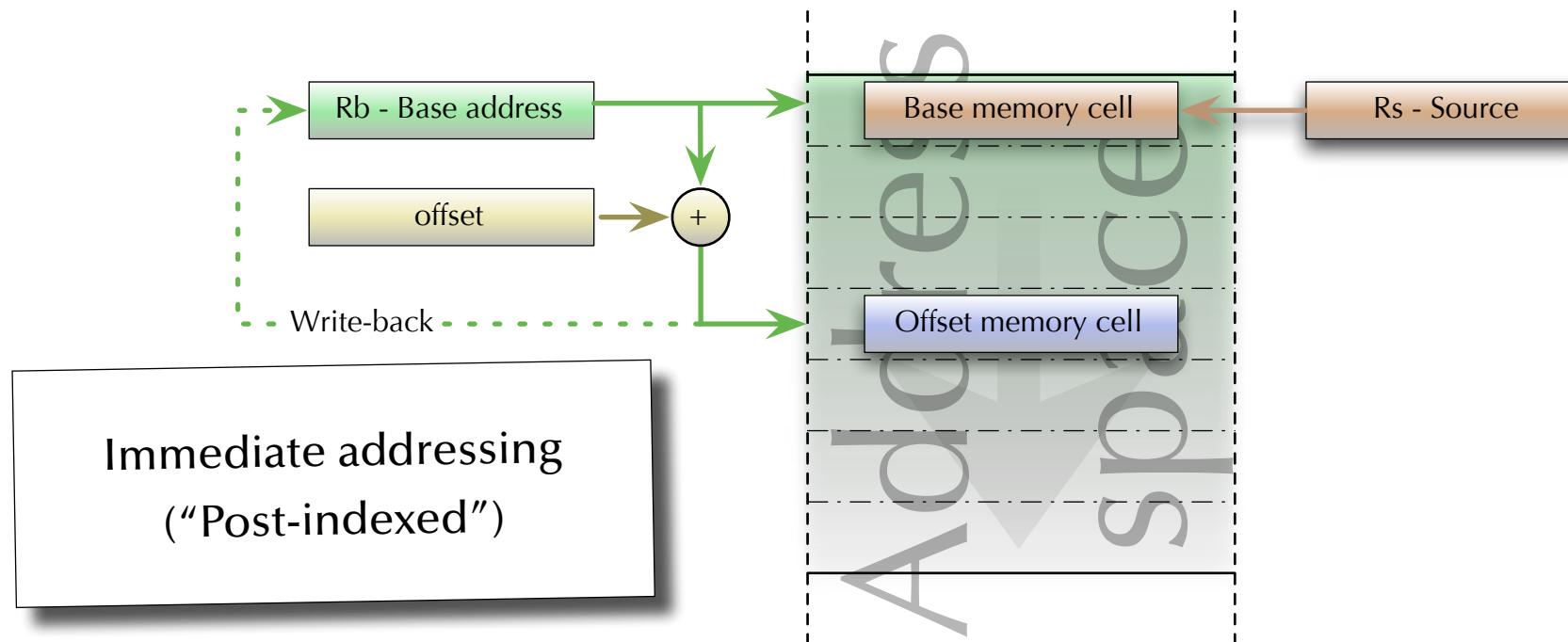
ldr r1, [r4], #8

Reads from a memory cell with a base register address
and writes the offset address back into the original base register.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



ldr<c><q> <Rd>, [<Rb>], #+/-<offset>
str<c><q> <Rs>, [<Rb>], #+/-<offset>

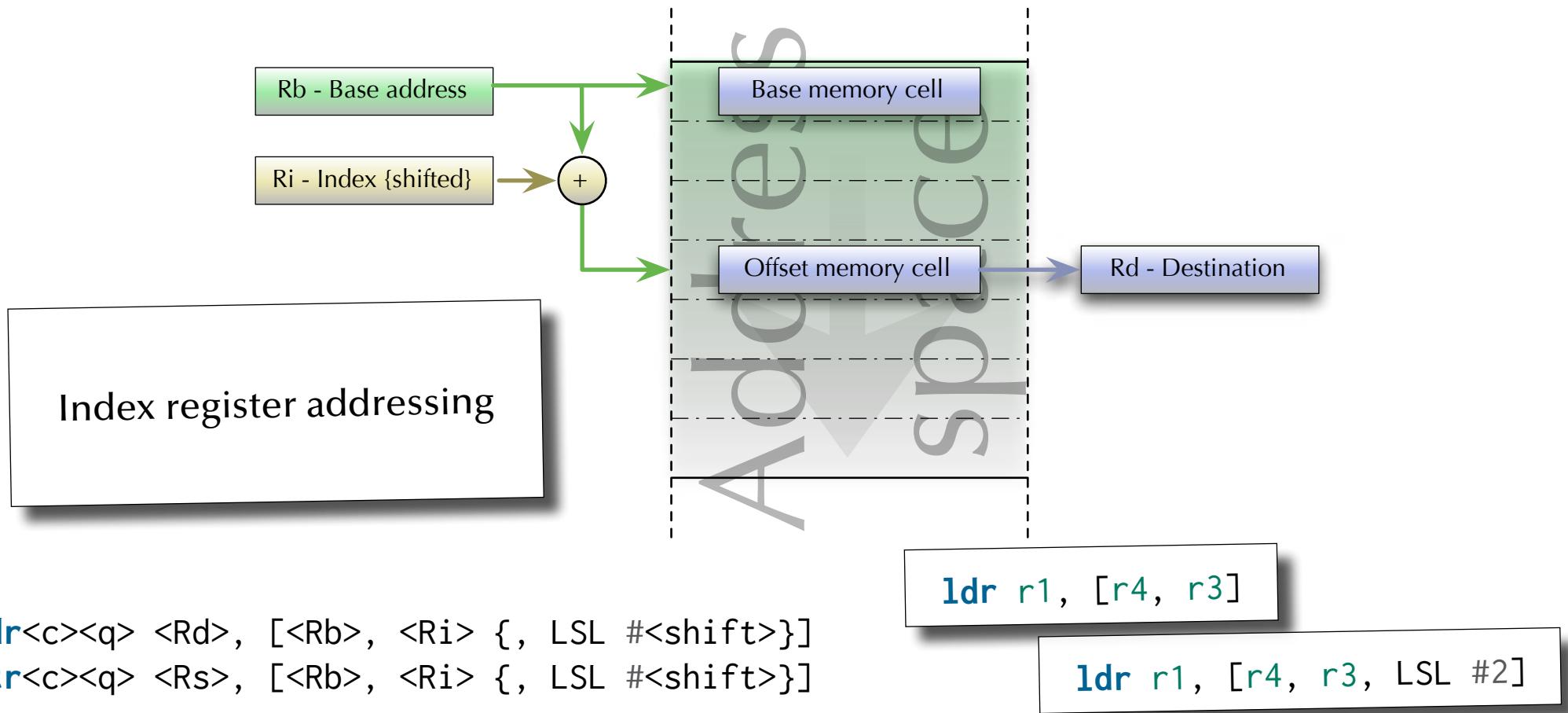
str r1, [r4], #8

Writes to a memory cell with a base register address and writes the offset address back into the original base register.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU

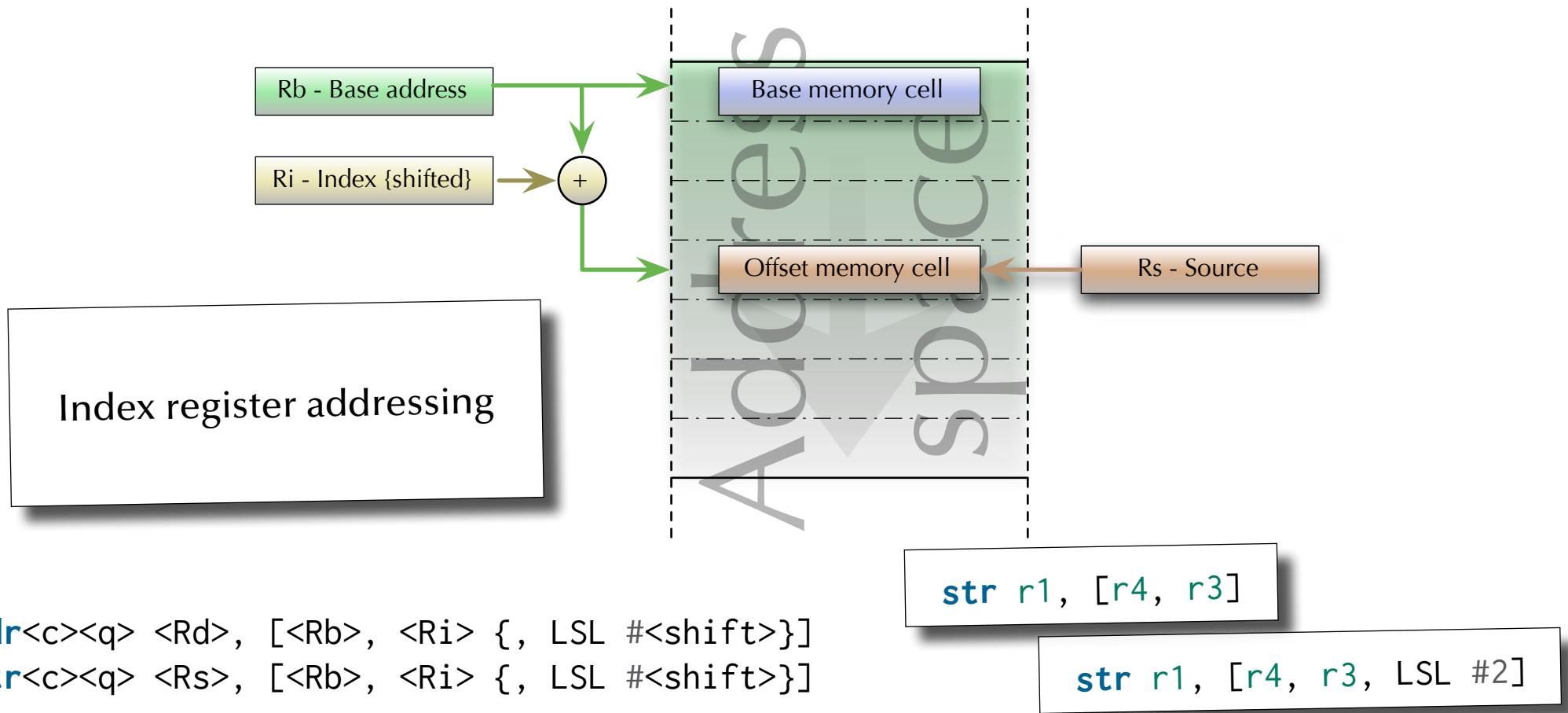


Reads from a memory cell with a base register address plus a potentially shifted index register.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU

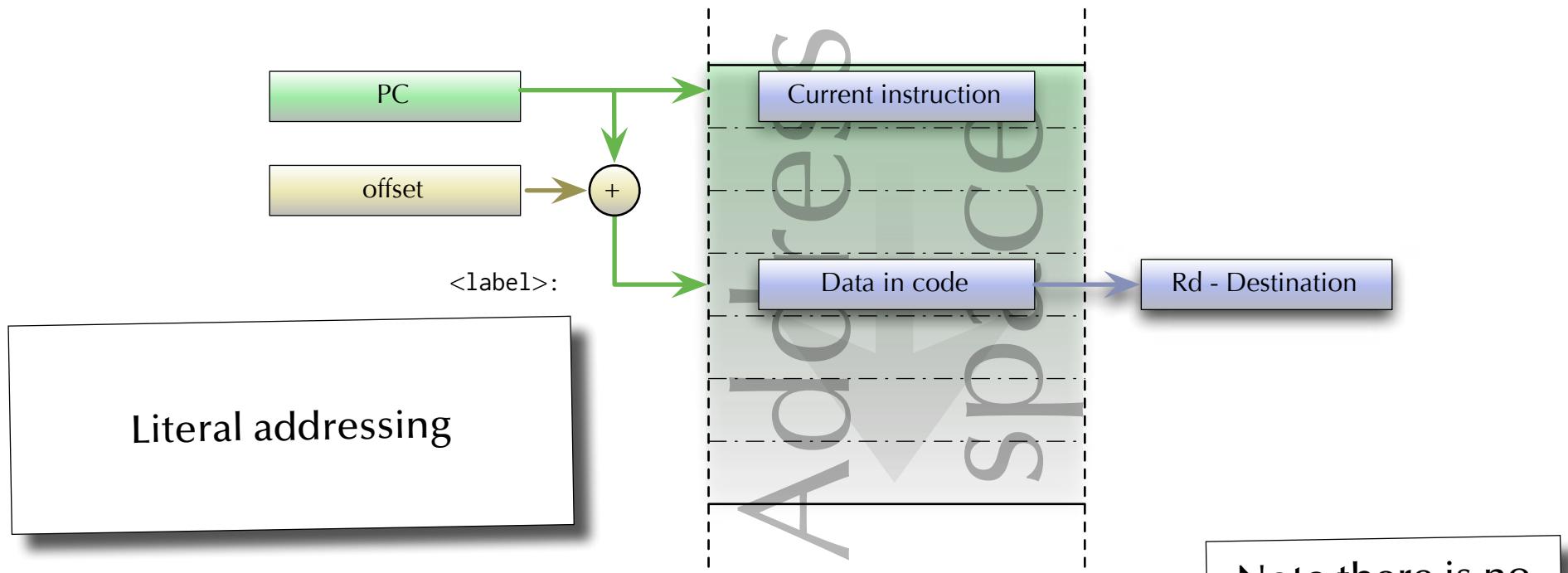


Writes to a memory cell with a base register address plus a potentially shifted index register.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



ldr<c><q> <Rd>, <label>

ldr<c><q> <Rd>, [PC, #+/-<offset>]

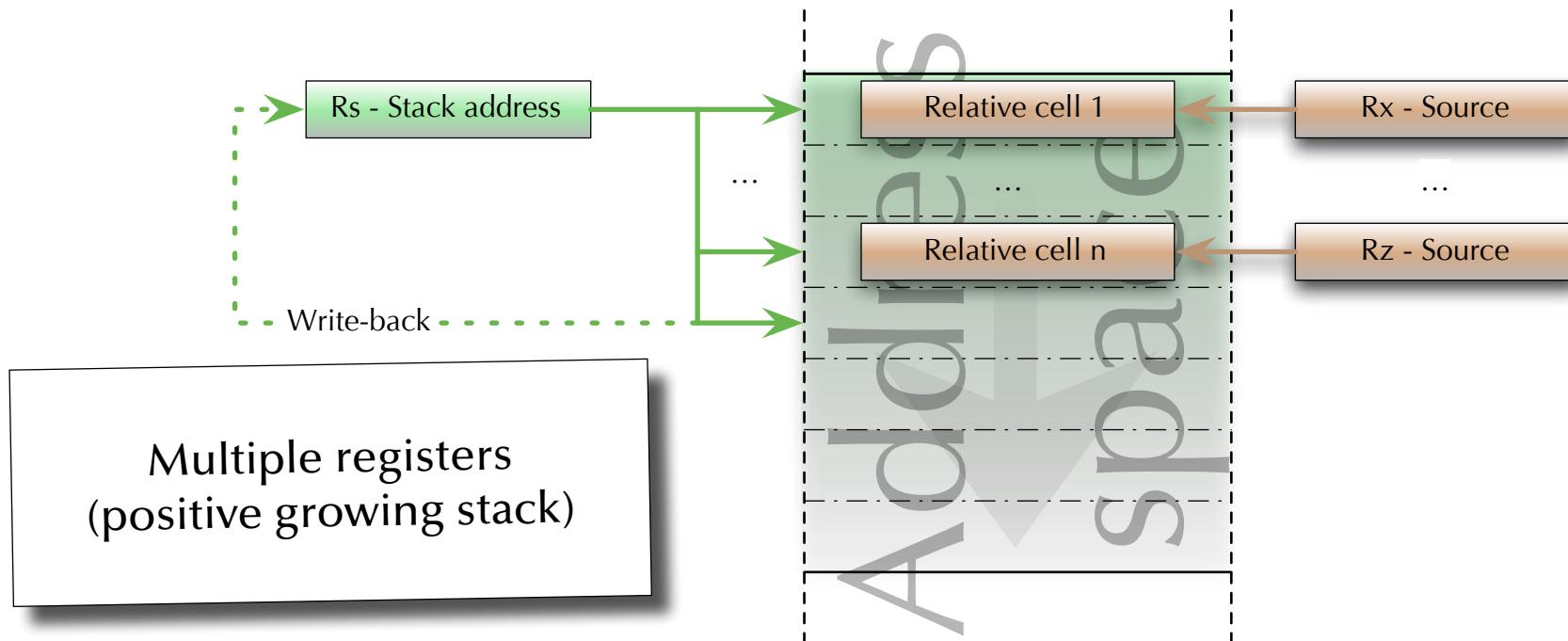
Reads from a data area embedded into the code section.

ldr r1, data



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



```
stmia<c><q> <Rs>{!}, <registers>  
ldmdb<c><q> <Rs>{!}, <registers>
```

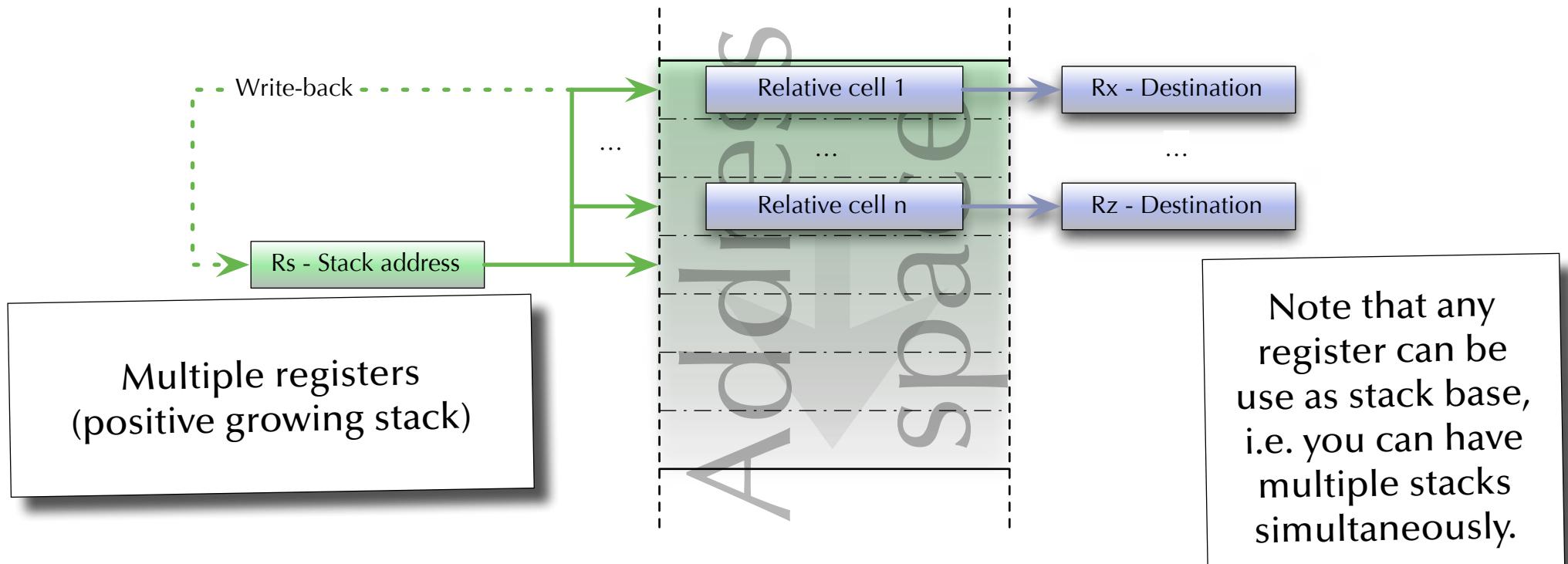
```
stmia r9!, {r1, r3, r4, fp}
```

Stores multiple registers into sequential memory addresses.
Stores “increment after” and loads “decrement before”.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



```
stmia<c><q> <Rs>{!}, <registers>  
ldmdb<c><q> <Rs>{!}, <registers>
```

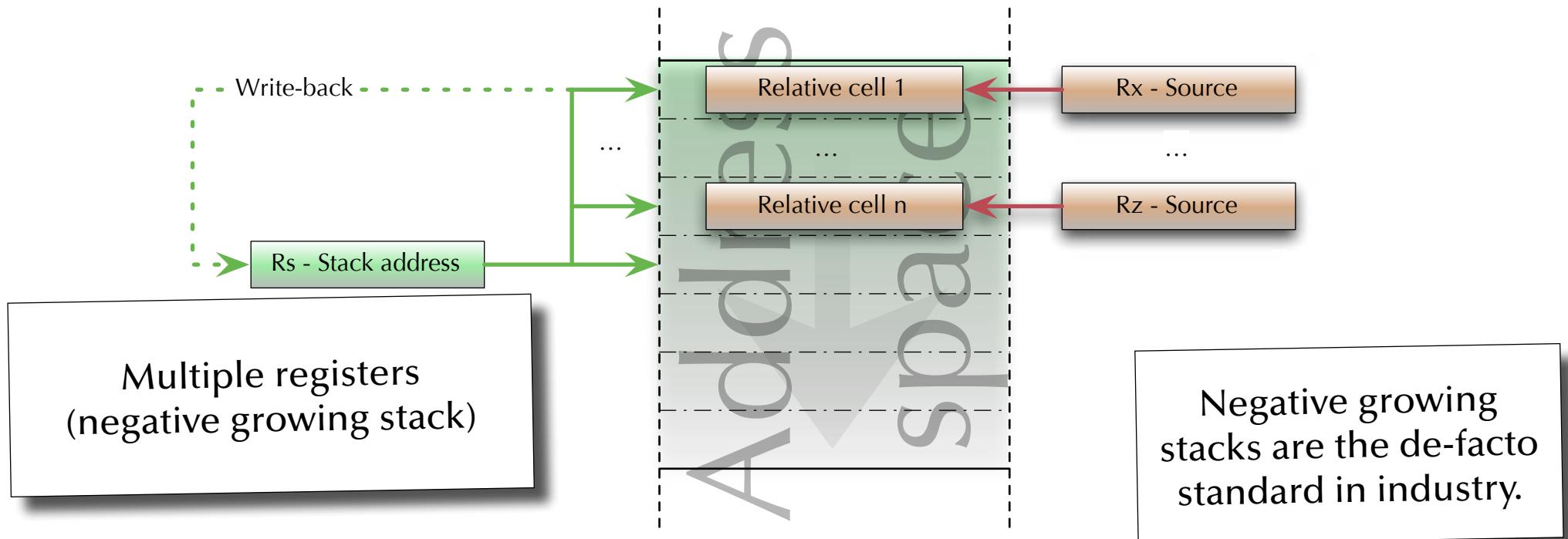
```
ldmdb r9!, {r1, r3, r4, fp}
```

Reads multiple registers from sequential memory addresses.
Stores “increment after” and loads “decrement before”.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



`stmdb<c><q> <Rs>{!}, <registers>
ldmia<c><q> <Rs>{!}, <registers>`

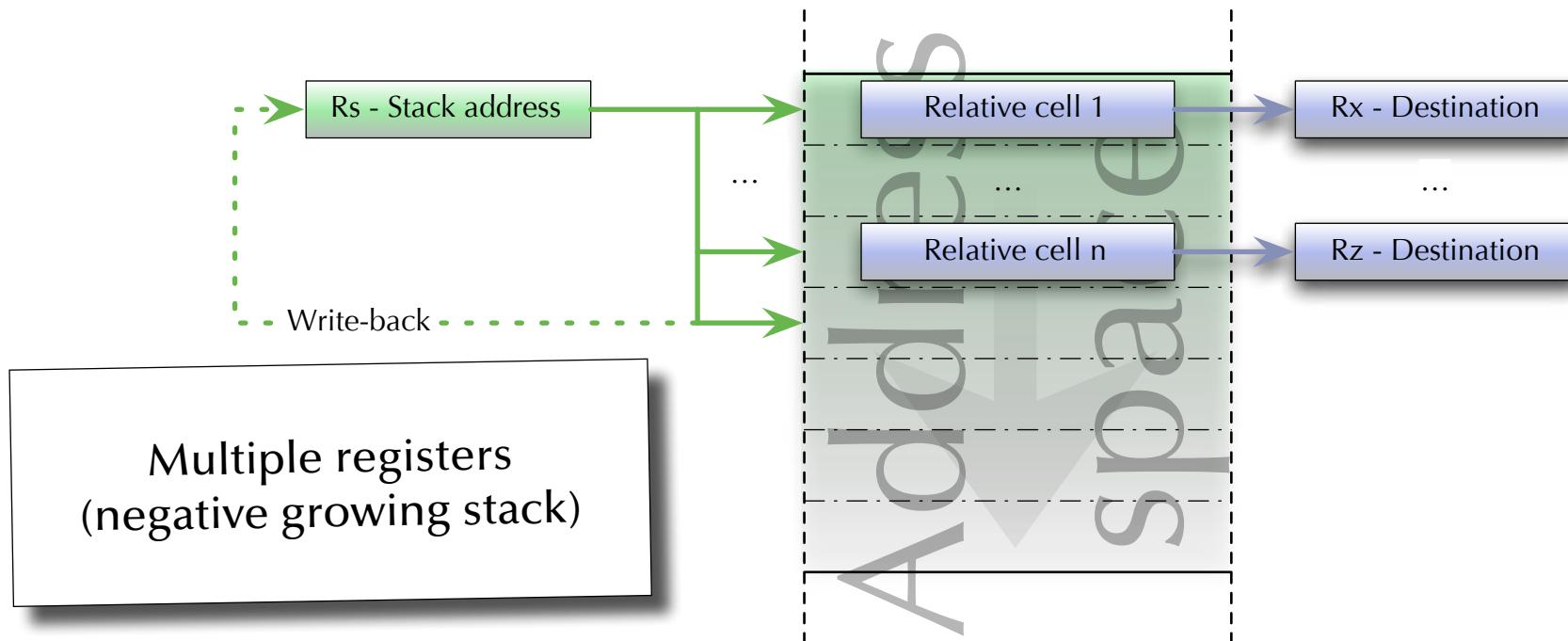
`stmdb SP!, {r1, r3, r4, fp}`

Stores multiple registers to sequential memory addresses.
Stores “decrement before” and loads “increment after”.



Hardware/Software Interface

ARM v7-M Move data in and out of the CPU



stmdb<c><q> <Rs>{!}, <registers>
ldmia<c><q> <Rs>{!}, <registers>

ldmia SP!, {r1, r3, r4, fp}

Reads multiple registers from sequential memory addresses.
Stores “decrement before” and loads “increment after”.



Hardware/Software Interface

Simple arithmetic in memory

Calculate again:

$e := a + b - 2*c$

but now a, b, c and e are stored in memory, relative to an address stored in FP (“Frame Pointer”):
a is held at [fp - 12], b at [fp - 16], c at [fp - 20] and e at [fp - 24]

In order to do arithmetic we need to load those values into the CPU
first and afterwards we need to store the result in memory:

```
ldr  r1, [fp, #-12]
ldr  r2, [fp, #-16]
add r1, r1, r2
ldr  r2, [fp, #-20]
lsl  r2, r2, #1
sub r1, r1, r2
str  r1, [fp, #-24]
```

☞ Notice that this time we only used two registers.



Hardware/Software Interface

Simple arithmetic in memory

Calculate again:

```
e := a + b - 2*c
```

Or in saturation arithmetic:

```
ldr  r1, [fp, #-12]
ldr  r2, [fp, #-16]
qadd r1, r1, r2
ldr  r2, [fp, #-20]
qadd r2, r2, r2
qsub r1, r1, r2
str  r1, [fp, #-24]
```



Hardware/Software Interface

Simple arithmetic in memory

Calculate again:

```
e := a + b - 2*c
```

Or with overflow checks:

```
ldr r1, [fp, #-12]
ldr r2, [fp, #-16]
adds r1, r1, r2      ; need to check overflow flag
ldr r2, [fp, #-20]
lsl r2, r2, #1        ; need to check that the sign did not change
subs r1, r1, r2       ; need to check overflow flag
str r1, [fp, #-24]
```

👉 It's time we learn about branching off into alternative execution paths.



Hardware/Software Interface

ARM v7-M Branch instructions

b <c><q> <label>	; if c then	PC := label
bl <c> <label>	; if c then LR := PC_next; PC := label	
bx <c> <Rm>	; if c then	PC := Rm
blx <c><q> <Rm>	; if c then LR := PC_next; PC := Rm	
cbz <q> <Rn>, <label>	; if Rn = 0 then	PC := label
cbnz <q> <Rn>, <label>	; if Rn /= 0 then	PC := label

<c>	Meanings	Flags
eq	Equal	Z = 1
ne	Not equal	Z = 0
cs, hs	Carry set, Unsigned higher or same	C = 1
cc, lo	Carry clear, Unsigned lower	C = 0
mi	Minus, Negative	N = 1
pl	Plus, Positive or zero	N = 0
vs	Overflow	V = 1
vc	No overflow	V = 0
hi	Unsigned higher	C = 1 \wedge Z = 0
ls	Unsigned lower or same	C = 0 \vee Z = 1
ge	Signed greater or equal	N = Z
lt	Signed less	N \neq Z
gt	Signed greater	Z = 0 \wedge N = V
le	Signed less or equal	Z = 1 \vee N \neq V
al, <none>	Always	any



Hardware/Software Interface

Simple arithmetic in memory

Calculate again:

```
e := a + b - 2*c
```

Or with overflow checks:

```
ldr r1, [fp, #-12]
ldr r2, [fp, #-16]
adds r1, r1, r2
bvs Overflow      ; branch if overflow is set
ldr r2, [fp, #-20]
adds r2, r2, r2
bvs Overflow      ; branch if overflow is set
subs r1, r1, r2
bvs Overflow      ; branch if overflow is set
str r1, [fp, #-24]
```

...

Overflow:

```
svc #5           ; call the operating system or runtime environment with #5
; (assuming that #5 indicates an overflow situation)
```



Hardware/Software Interface

Simple arithmetic in memory

Calculate again:

```
e := a + b - 2*c
```

Or with overflow checks:

```
ldr r1, [fp, #-12]
ldr r2, [fp, #-16]
adds r1, r1, r2
bvs Overflow      ; branch if overflow is set
ldr r2, [fp, #-20]
adds r2, r2, r2
bvs Overflow      ; branch if overflow is set
subs r1, r1, r2
bvs Overflow      ; branch if overflow is set
str r1, [fp, #-24]
```

...

Overflow:

```
svc #5          ; call the operating system or runtime environment with #5
; (assuming that #5 indicates an overflow situation)
```

... but how do we know
where this happened or how
to continue operations?



Hardware/Software Interface

Simple arithmetic in memory

Calculate again:

```
e := a + b - 2*c
```

Or with overflow checks:

```
ldr r1, [fp, #-12]
ldr r2, [fp, #-16]
adds r1, r1, r2
blvs Overflow      ; branch if overflow is set; keep next location in LR
ldr r2, [fp, #-20]
adds r2, r2, r2
blvs Overflow      ; branch if overflow is set; keep next location in LR
subs r1, r1, r2
blvs Overflow      ; branch if overflow is set; keep next location in LR
str r1, [fp, #-24]
```

...

Overflow:

```
...           ; ... for example writing a log entry with location
bx lr       ; resume operations - assuming the above did not change LR
```



Hardware/Software Interface

ARM v7-M Essential multiplications and divisions

32 bit to 32 bit

mul {s}<c><q> {<Rd>, } <Rn>,<Rm>	; Rd := (Rn*Rm)
mla <c> <Rd>, <Rn>,<Rm>,<Ra>	; Rd := Ra + (Rn*Rm)
mls <c> <Rd>, <Rn>,<Rm>,<Ra>	; Rd := Ra - (Rn*Rm)
udiv <c> <Rd>, <Rn>,<Rm>	; Rd := <i>unsigned</i> (Rn/Rm); rounded towards 0
sdiv <c> <Rd>, <Rn>,<Rm>	; Rd := <i>signed</i> (Rn/Rm); rounded towards 0

32 bit to 64 bit

umull <c> <RdLo>,<RdHi>,<Rn>,<Rm>	; RdHi:RdLo := <i>unsigned</i> (Rn*Rm)
umlal <c><q> <RdLo>,<RdHi>,<Rn>,<Rm>	; RdHi:RdLo := <i>unsigned</i> (RdHi:RdLo + (Rn*Rm))
smull <c> <RdLo>,<RdHi>,<Rn>,<Rm>	; RdHi:RdLo := <i>signed</i> (Rn*Rm)
smlal <c> <RdLo>,<RdHi>,<Rn>,<Rm>	; RdHi:RdLo := <i>signed</i> (RdHi:RdLo + (Rn*Rm))

... versions for narrower numbers, as well as versions which operate on multiple narrower numbers in parallel exist as well.



Hardware/Software Interface

Straight power

Calculate:

$c := a ^ b$

```
mov r1, #7          ; a
mov r2, #11         ; b ; has to be non-negative
mov r3, #1          ; c
```

$$7^{11} = 7 \cdot 7$$

power:

```
cbz r2, end_power ; exponent zero?
mul r3, r1
sub r2, #1
b power
```

end_power:

```
nop                ; c = a ^ b
```

How many iterations?

How many cycles?



Hardware/Software Interface

More power

Calculate:

```
c := a ^ b  
  
    mov r1, #7          ; a  
    mov r2, #11         ; b ; has to be non-negative  
    mov r3, #1          ; c  
    mov r4, r1          ; base a to the powers of two, starting with a ^ 1
```

power:

```
    cbz r2, end_power   ; exponent zero?  
    tst r2, #0b1        ; right-most bit of exponent set?  
    beq skip            ; skip this power if not  
    mul r3, r4           ; multiply the current power into result
```

skip:

```
    mul r4, r4           ; calculate next power  
    lsr r2, #1           ; divide exponent by 2  
    b      power
```

end_power:

```
    nop                 ; c = a ^ b
```

$$7^{11} = 7^8 \cdot 7^2 \cdot 7^1$$

How many iterations?
How many cycles?



Hardware/Software Interface

Table based branching

tbb<c><q> [<Rn>, <Rm>] ; for tables of offset bytes (8bit)
tbh<c><q> [<Rn>, <Rm>, **lsl** #1] ; for tables of offset halfwords (16bit)

Common usage for byte (8 bit) tables

tbb [PC, Ri] ; PC is base of branch table, Ri is index

Branch_Table:

.byte (Case_A - Branch_Table)/2 ; Case_A 8 bit offset
.byte (Case_B - Branch_Table)/2 ; Case_B 8 bit offset
.byte (Case_C - Branch_Table)/2 ; Case_C 8 bit offset
.byte 0x00 ; Padding to re-align with halfword boundaries

Case_A:

... ; any instruction sequence
b End_Case ; “break out”

Case_B:

... ; any instruction sequence
b End_Case ; “break out”

Case_C:

... ; any instruction sequence

End_Case:



Hardware/Software Interface

Table based branching

tbb<c><q> [<Rn>, <Rm>] ; for tables of offset bytes (8bit)

tbh<c><q> [<Rn>, <Rm>, **lsl** #1] ; for tables of offset halfwords (16bit)

Common usage for halfword (16 bit) tables

tbh [PC, Ri, **lsl** #1] ; PC used as base of branch table, Ri is index

Branch_Table:

.hword (Case_A - Branch_Table)/2 ; Case_A 16 bit offset

.hword (Case_B - Branch_Table)/2 ; Case_B 16 bit offset

.hword (Case_C - Branch_Table)/2 ; Case_C 16 bit offset

Case_A:

... ; any instruction sequence

b End_Case ; “break out”

Case_B:

... ; any instruction sequence

b End_Case ; “break out”

Case_C:

... ; any instruction sequence

End_Case:



Hardware/Software Interface

Basic instruction sets

Category	Side effects	ARM v7-M
Arithmetic, Logic	Sets and uses CPU flags	<code>add, adc, qadd, sub, sbc, qsub, rsb, mul, mla, mls, udiv, sdiv, umull, umlal, smull, smlal, and, bic, orr, orn, eor, cmp, cmn, tst, teq</code>
Move and shift registers		<code>mov, lsr, asr, lsl, ror, rrx</code>
Branching	Uses CPU flags	<code>b, bl, bx, blx, tbb, tbh</code>
Load & Store	Effects memory	<code>ldr, str, ldmdb, ldmia, stmia, stmdb</code>



Hardware/Software Interface

Basic instruction sets

Category	Side effects	ARM v7-M	Over 50 billion CPUs on this planet are running ARM instruction sets
Arithmetic, Logic	Sets and uses CPU flags	<code>add, adc, qadd, sub, sbc, qsub, rsb, mul, mla, mls, udiv, sdiv, umull, umlal, smull, smlal, and, bic, orr, orn, eor, cmp, cmn, tst, teq</code>	
Move and shift registers		<code>mov, lsr, asr, lsl, ror, rrx</code>	
Branching	Uses CPU flags	<code>b, bl, bx, blx, tbb, tbh</code>	
Load & Store	Effects memory	<code>ldr, str, ldmdb, ldmia, stmia, stmdb</code>	

Instruction sets in the field:

RISC: Power, ARM, MIPS, Alpha, SPARK, AVR, PIC, ...

CISC: x86, Z80, 6502, 68000, ...



Hardware/Software Interface

Basic instruction sets

Category	Side effects	ARM v7-M
Arithmetic, Logic	Sets and uses CPU flags	<code>add, adc, qadd, sub, sbc, qsub, rsb, mul, mla, mls, udiv, sdiv, umull, umlal, smull, smlal, and, bic, orr, orn, eor, cmp, cmn, tst, teq</code>
Move and shift registers		<code>mov, lsr, asr, lsl, ror, rrx</code>
Branching	Uses CPU flags	<code>b, bl, bx, blx, tbb, tbh</code>
Load & Store	Effects memory	<code>ldr, str, ldmdb, ldmia, stmia, stmdb</code>

What's missing?

- ☞ Changing CPU privileges and handling interrupts.
- ☞ Synchronizing instructions

Coming in later chapters
about concurrency and
operating systems.



Hardware/Software Interface

Summary

Hardware/Software Interface

- **Instruction formats**
 - Register sets
 - Instruction encoding
- **Arithmetic / Logic instructions inside the CPU**
 - Summation, Subtraction, Multiplication, Division
 - Logic and shift operations
- **Load / Store and addressing modes**
 - Direct, relative, indexed, and auto-index-increment addressing forms
- **Branching**
 - Conditional branching and unconditional jumps.

Computer Organisation & Program Execution 2021



3

Functions

Uwe R. Zimmer - The Australian National University



Functions

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

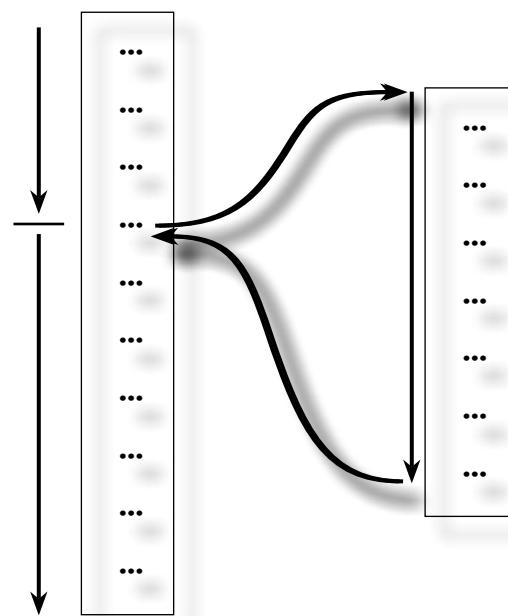
Computer Organization and Design – The Hardware/Software Interface

Chapter 2 “Instructions: Language of the Computer”

ARM edition, Morgan Kaufmann 2017



Functions





Functions

(Greatness from ...) Small beginnings

```
plus_1 :: (Num a) => a -> a
plus_1 x = x + 1
```

```
int plus1 (int x) {
    return x + 1;
}
```

```
function Plus_1 (x : Integer) return Integer is (x + 1);
```

```
def plus1 (x):
    return x + 1;
```

```
pure function plus_1 (x)
    int, intent (in) :: x
    int                  :: plus_1
    plus_1 = x + 1;
end function;
```

```
function Plus_1 (x : integer) : integer;
begin
    Plus_1 := x + 1;
end;
```



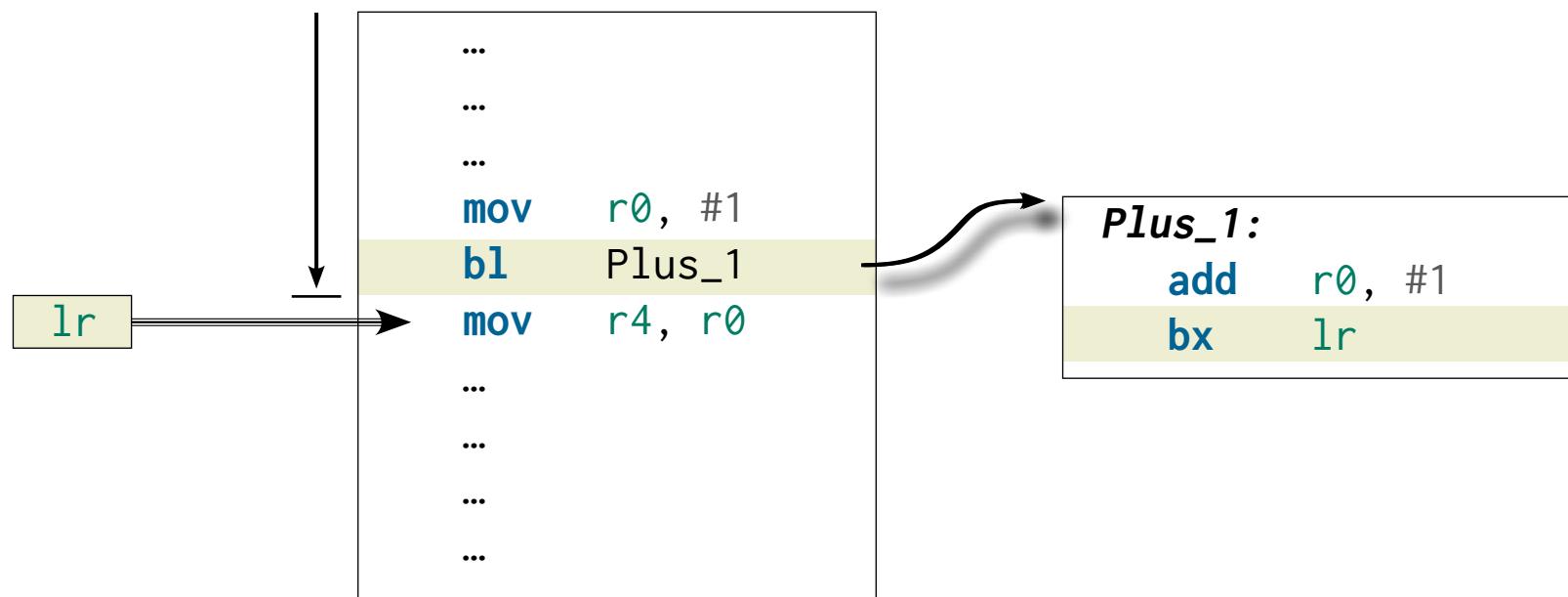
Functions

```
...
...
...
mov    r0, #1
bl     Plus_1
mov    r4, r0
...
...
...
...
```

```
Plus_1:
      add   r0, #1
      bx    lr
```



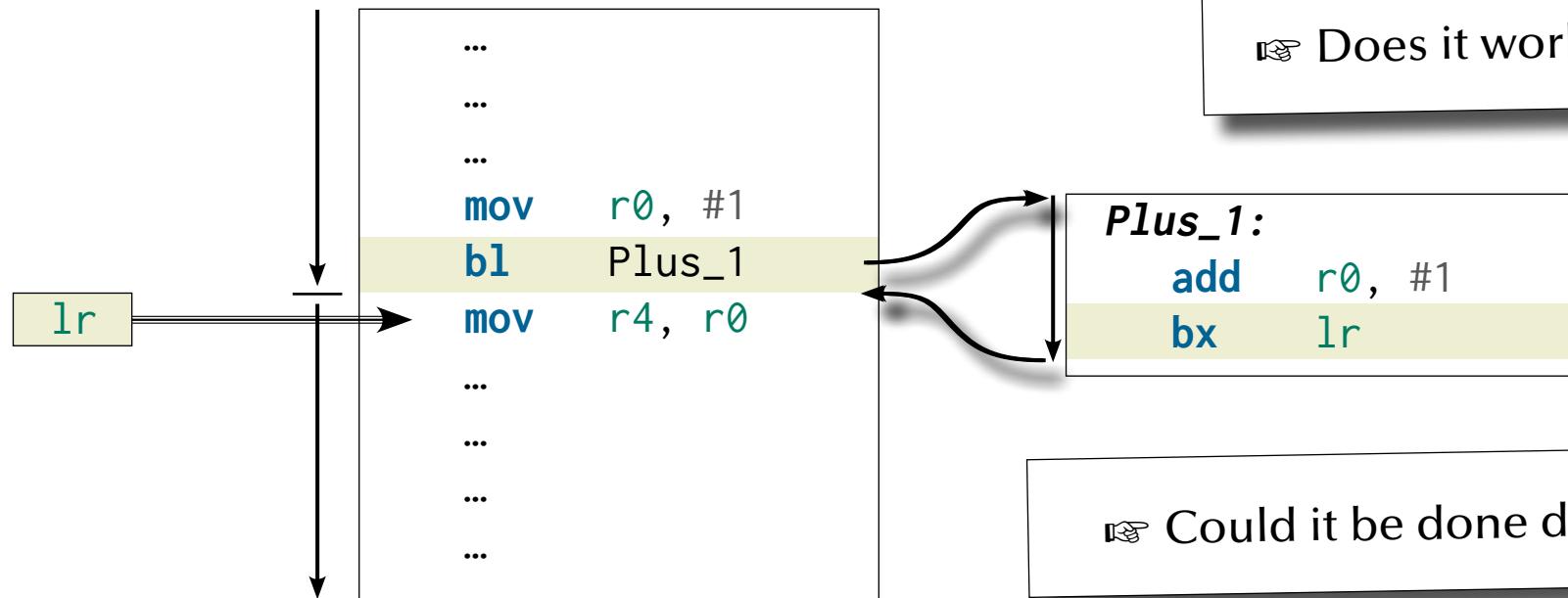
Functions





Functions

☞ How is the parameter x passed?



☞ Does it work?

☞ Could it be done differently?

☞ Where do you find the result after the function returns?



Functions

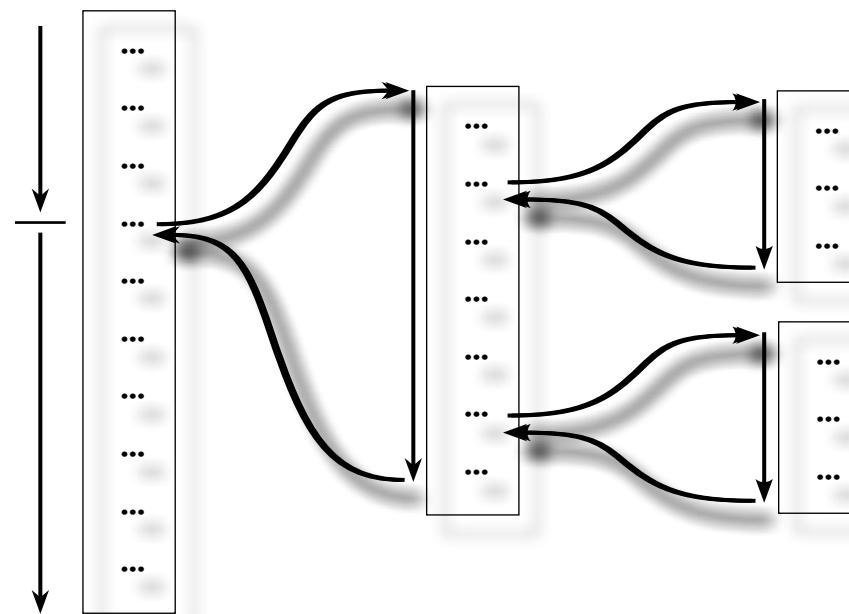
```
...
...
...
mov    r0, #1
add    r0, #1
mov    r4, r0
...
...
...
...
```

☞ This is called **inlining**

☞ Can/should this always be done?



Functions





Functions

(Greatness from ...) Small beginnings

```
plus_2 :: (Num a) => a -> a
plus_2 x = plus_1 $ plus_1 x
```

```
int plus2 (int x) {
    return plus1 (plus1 (x));
}
```

```
function Plus_2 (x : Integer) return Integer is (Plus_1 (Plus_1 (x)));
```

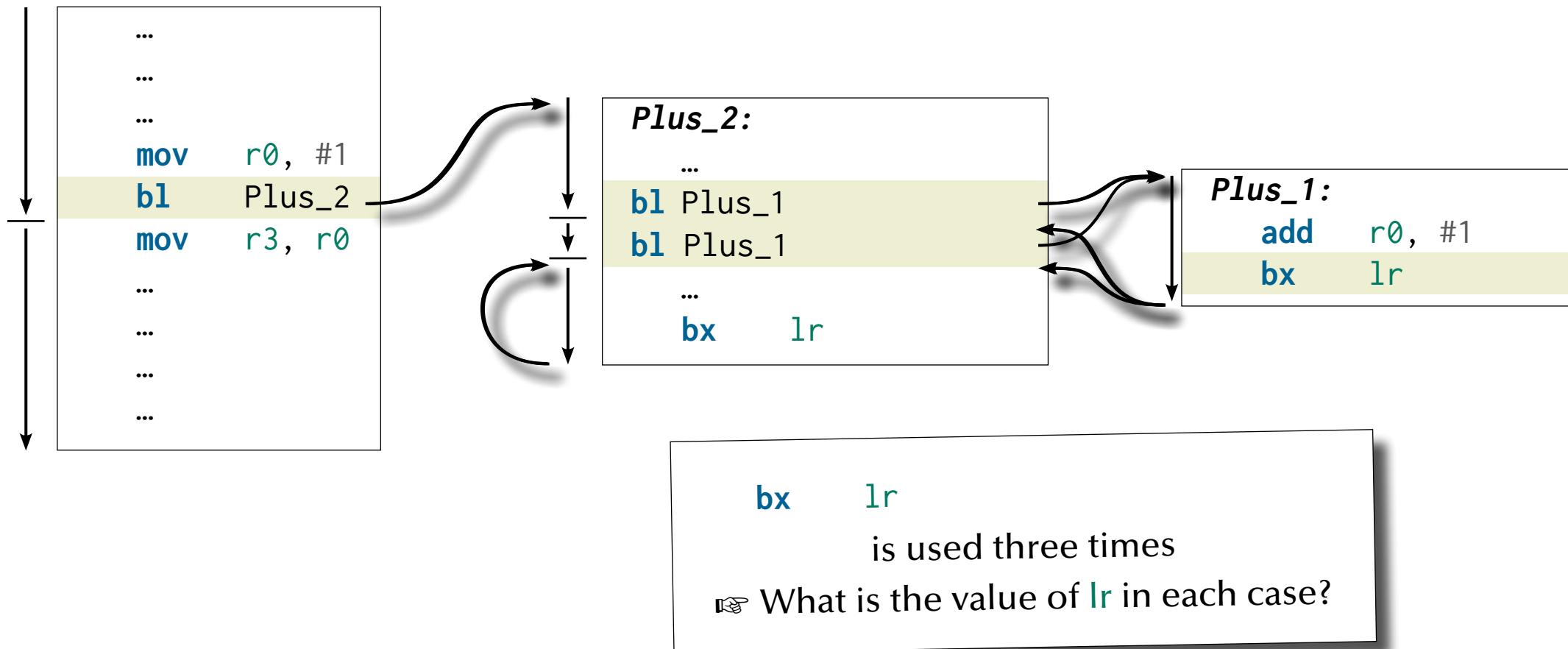
```
def plus2 (x):
    return plus1 (plus1 (x));
```

```
pure function plus_2 (x)
    int, intent (in) :: x
    int                  :: plus_2
    plus_2 = plus_1 (plus_1 (x));
end function;
```

```
function Plus_2 (x : integer) : integer;
begin
    Plus_2 := Plus_1 (Plus_1 (x));
end;
```

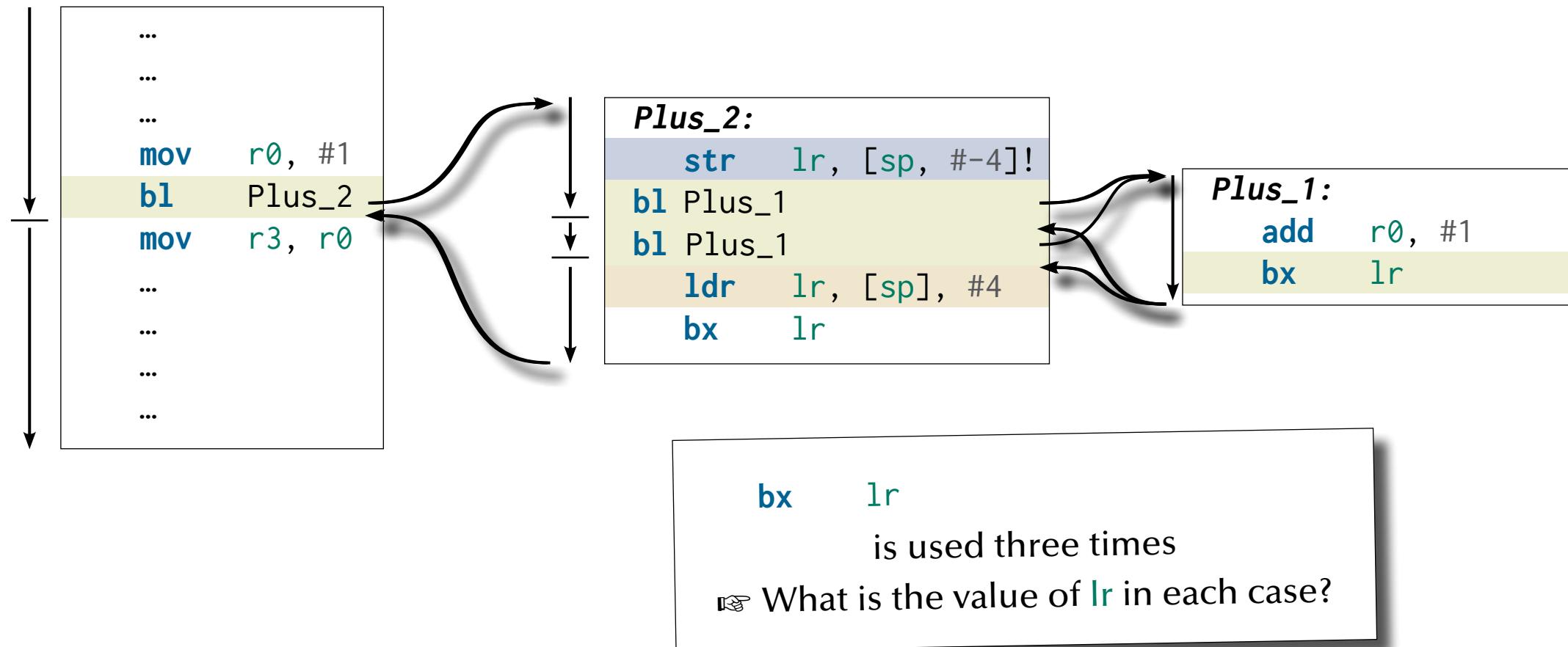


Functions





Functions





Functions

```
...
...
...
mov    r0, #1
add    r0, #2
mov    r3, r0
...
...
...
...
```

... we need an example, where a compiler
will not just remove all our code!



Functions

(Greatness from ...) Small beginnings

```
fib_fact :: (Num a) => a -> a
fib_fact x = (fib x) + (fact x)
```

```
unsigned int fibFact (unsigned int x) {
    return fib (x) + fact (x);
}
```

```
function Fib_Fact (x : Natural) return Natural is (Fib (x) + Fact (x));
```

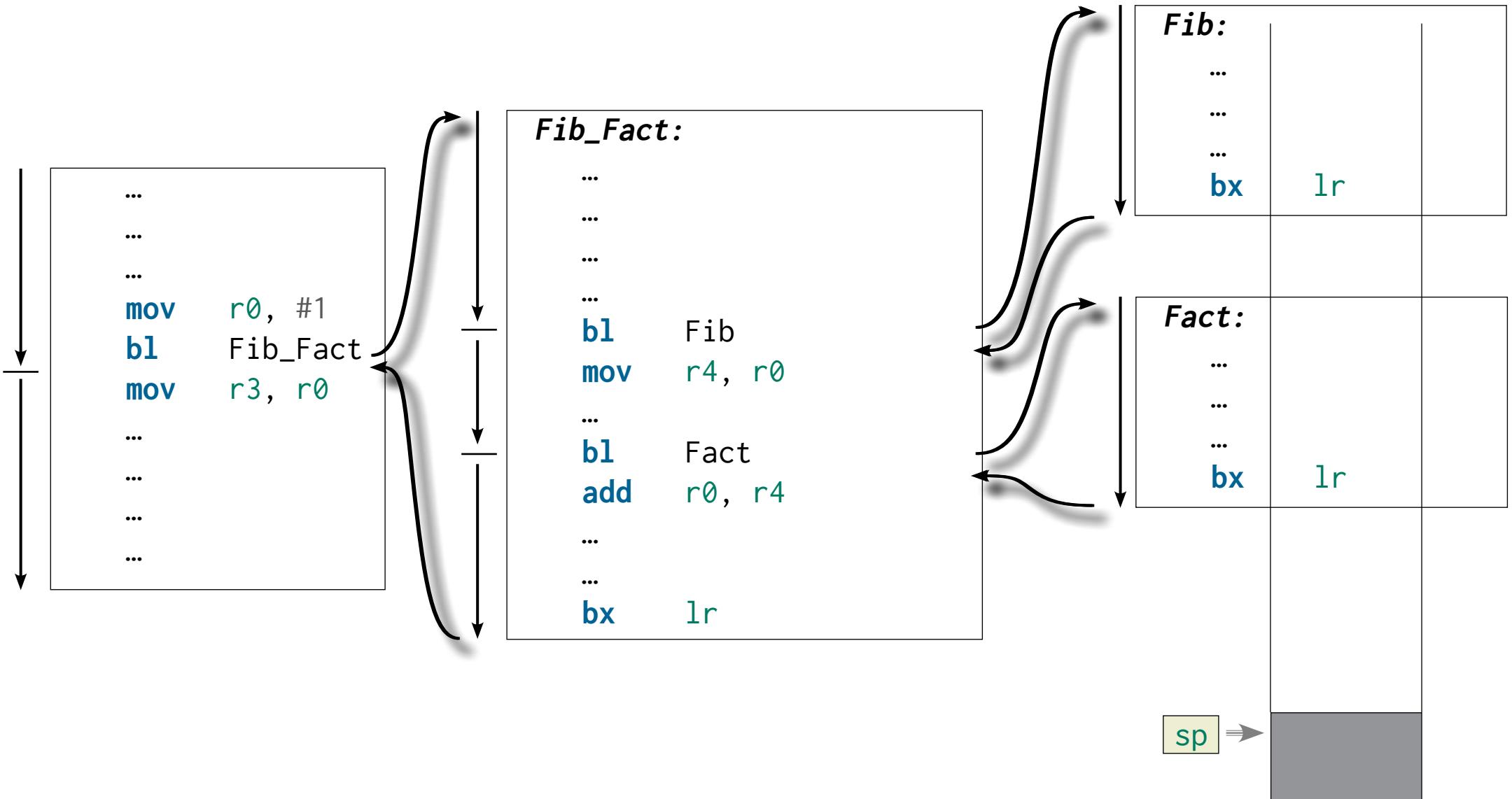
```
def fibFact (x):
    return fib (x) + fact (x);
```

```
pure function fib_fact (x)
    int, intent (in) :: x
    int                  :: fib_fact
    fib_fact = fib (x) + fact (x);
end function;
```

```
function Fib_Fact (x : cardinal) : cardinal;
begin
    Fib_Fact := Fib (x) + Fact (x);
end;
```

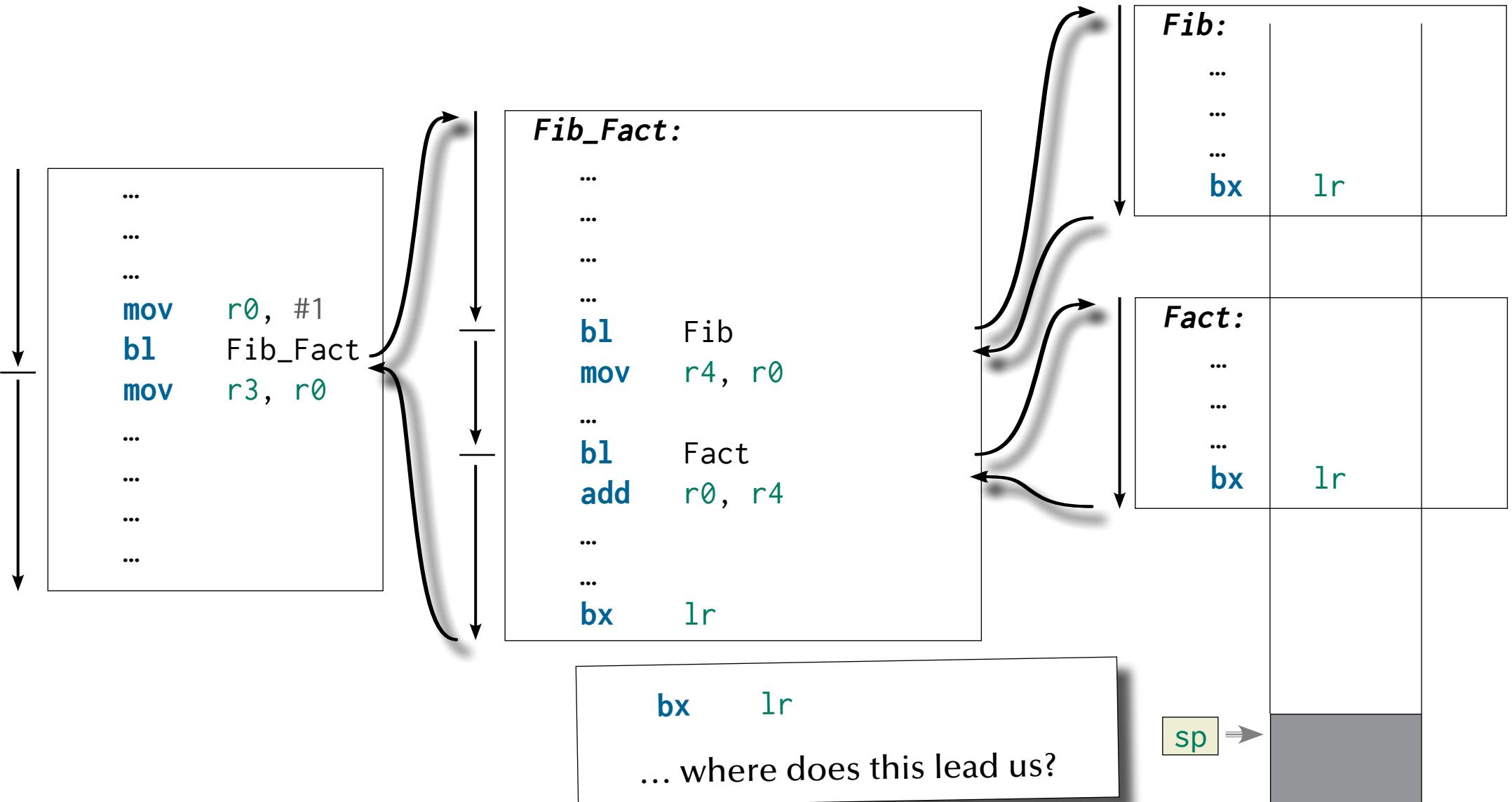


Functions



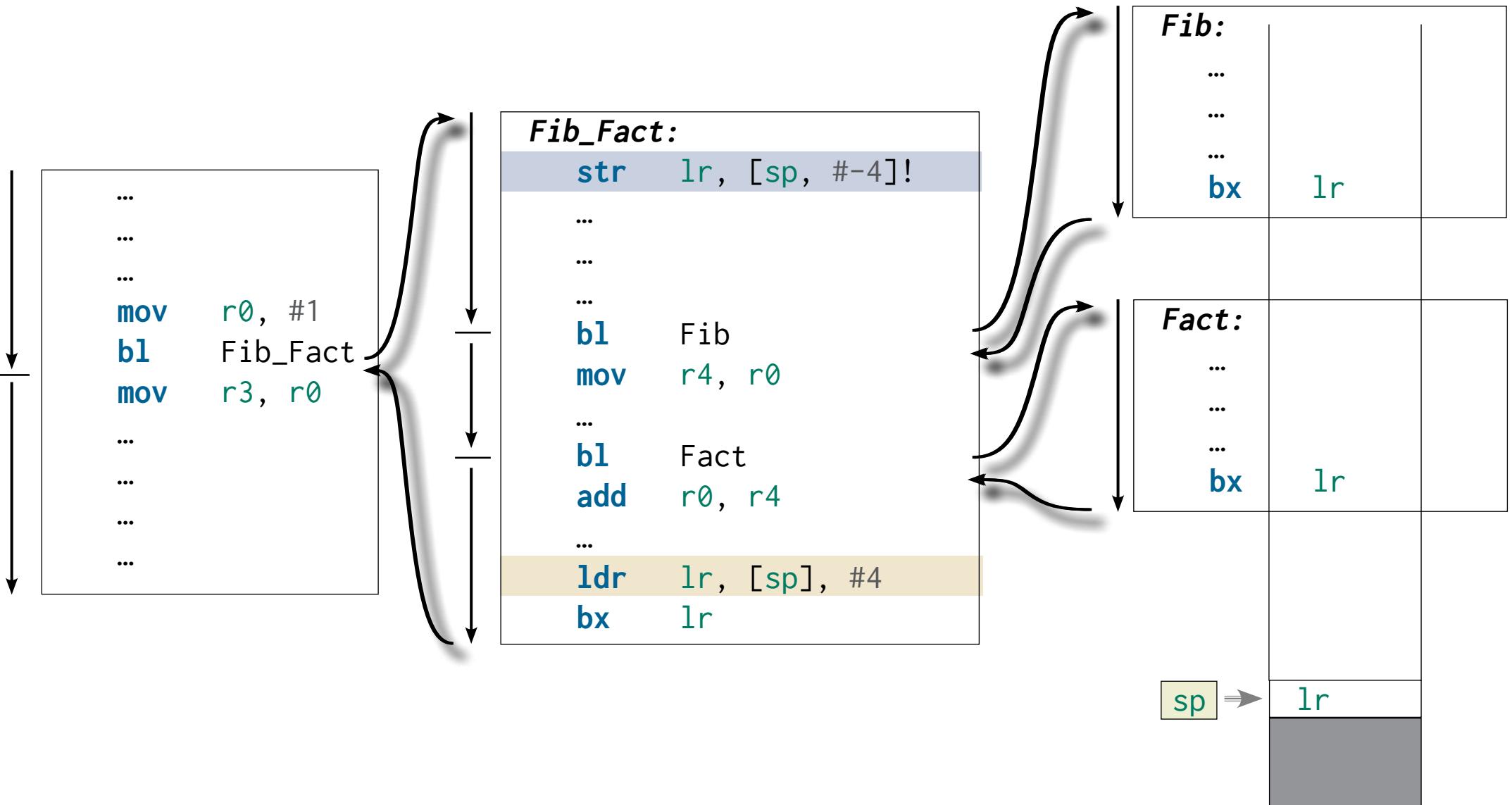


Functions



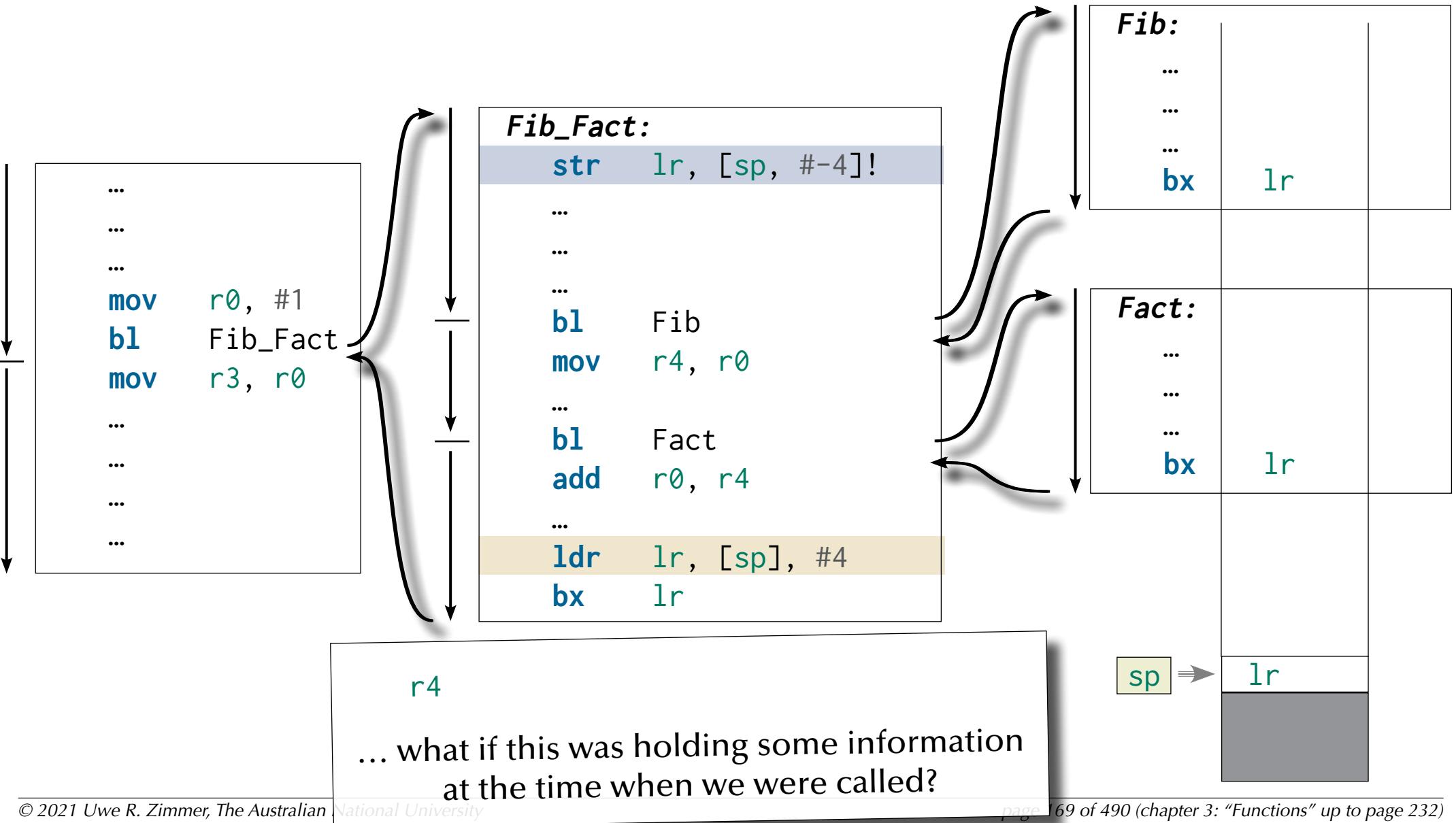


Functions



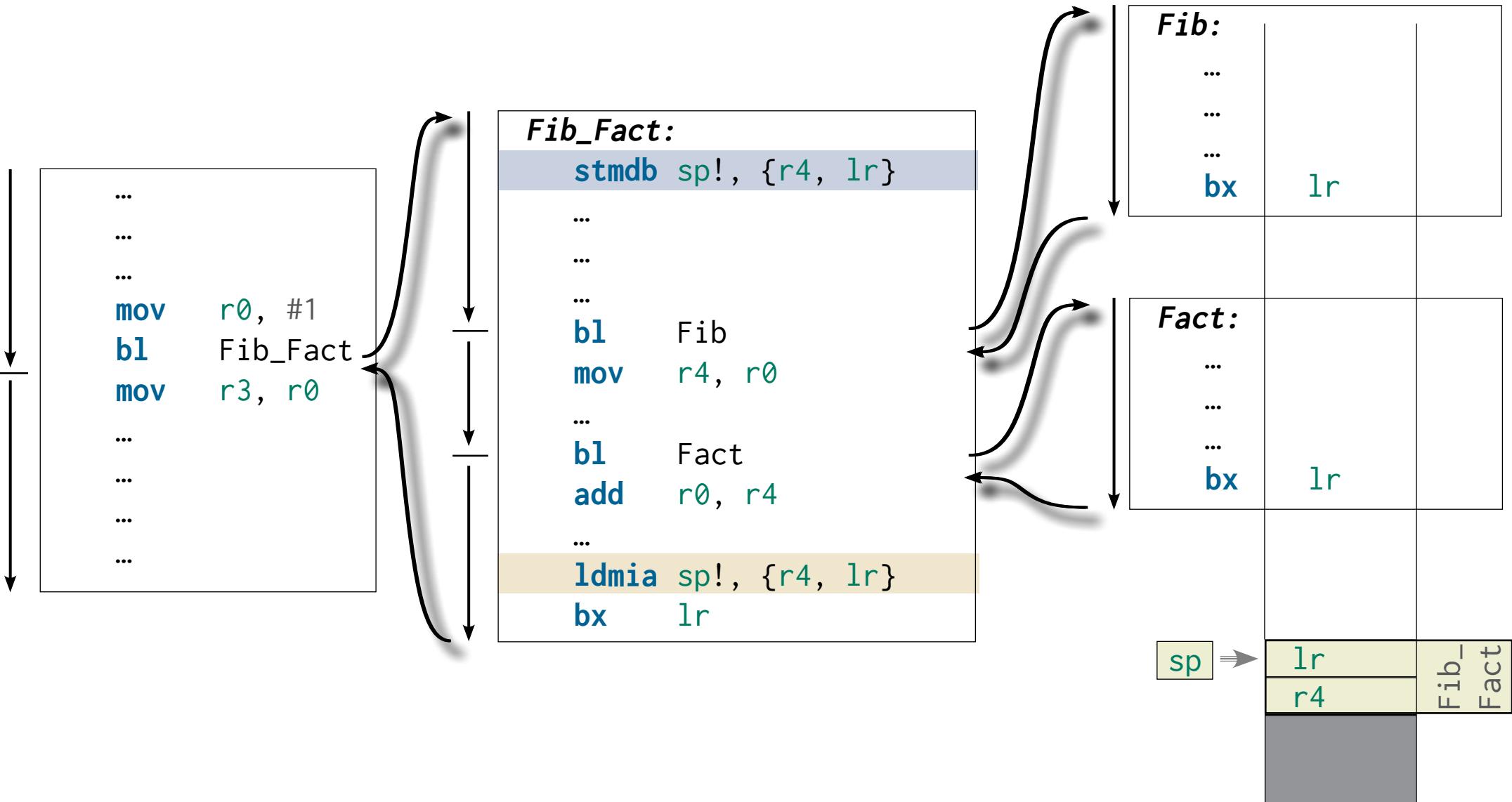


Functions



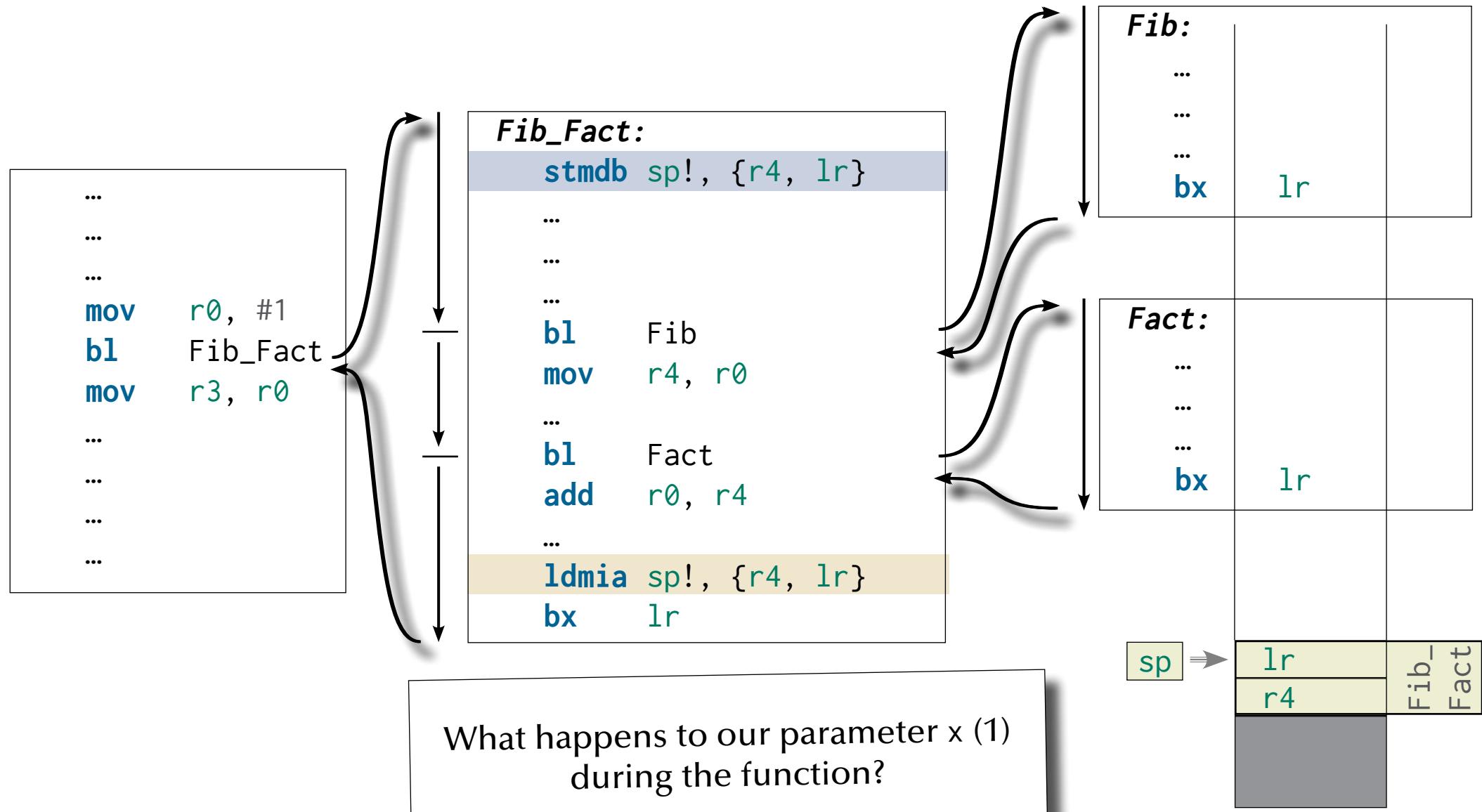


Functions



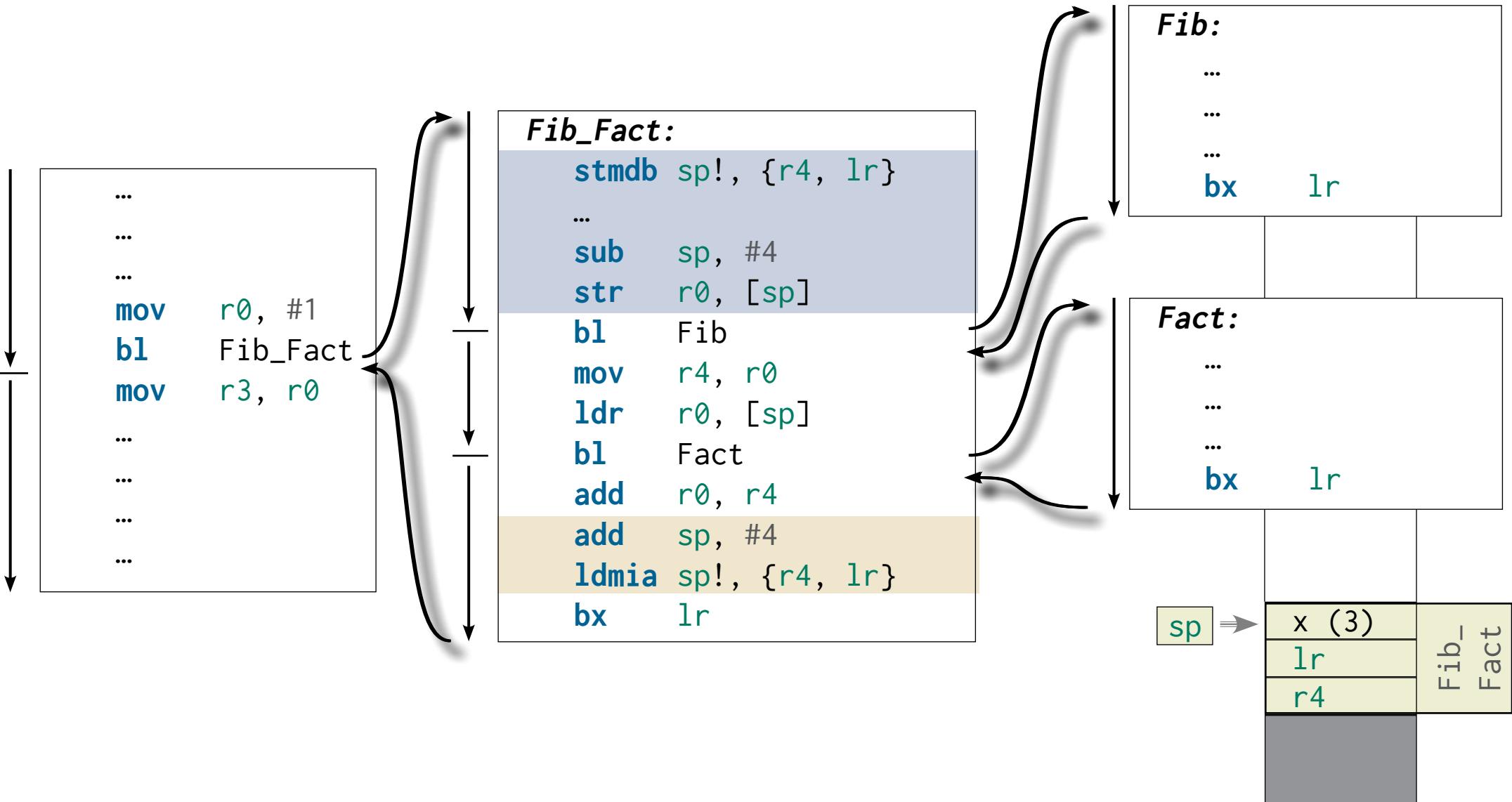


Functions



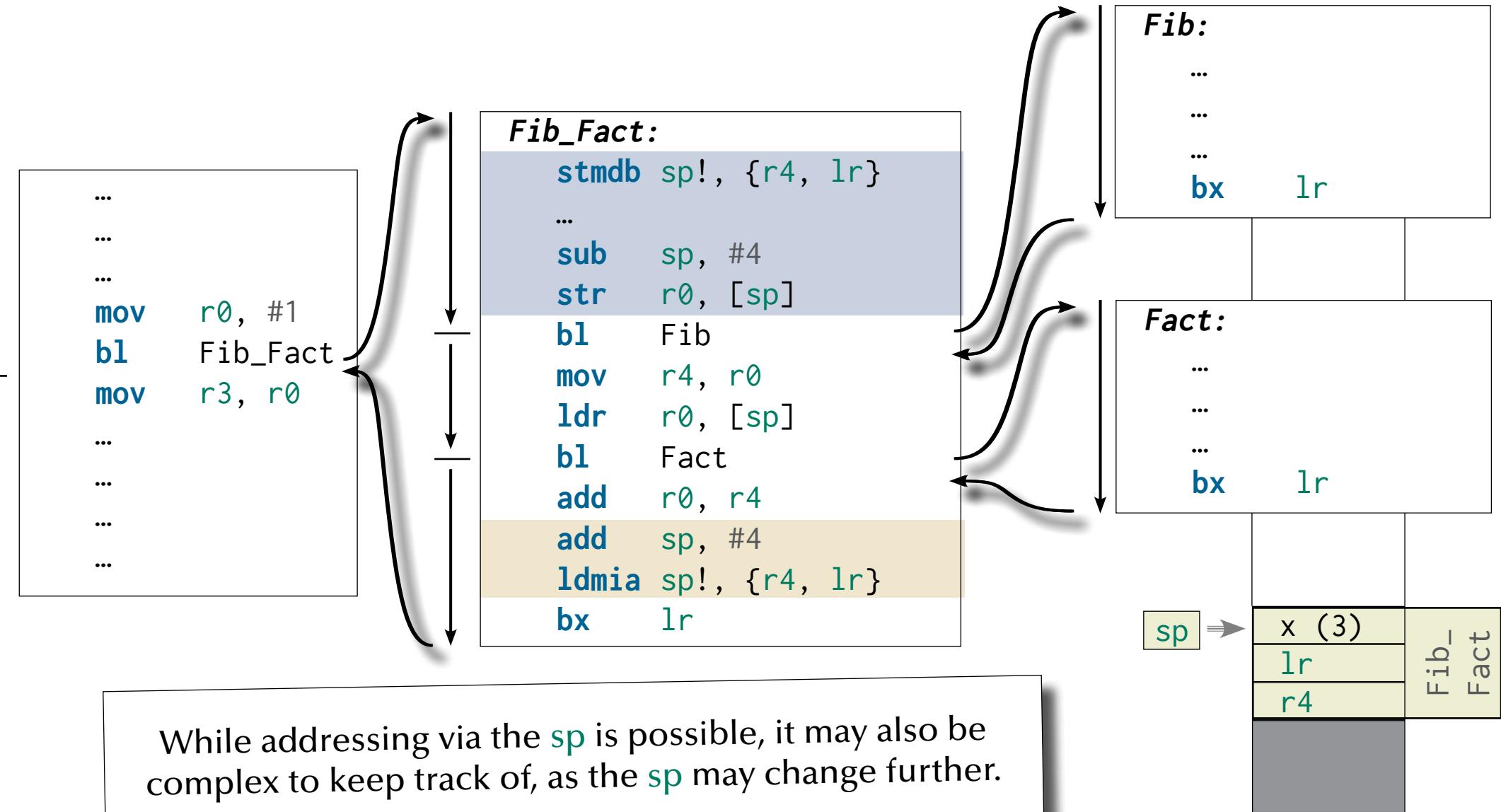


Functions



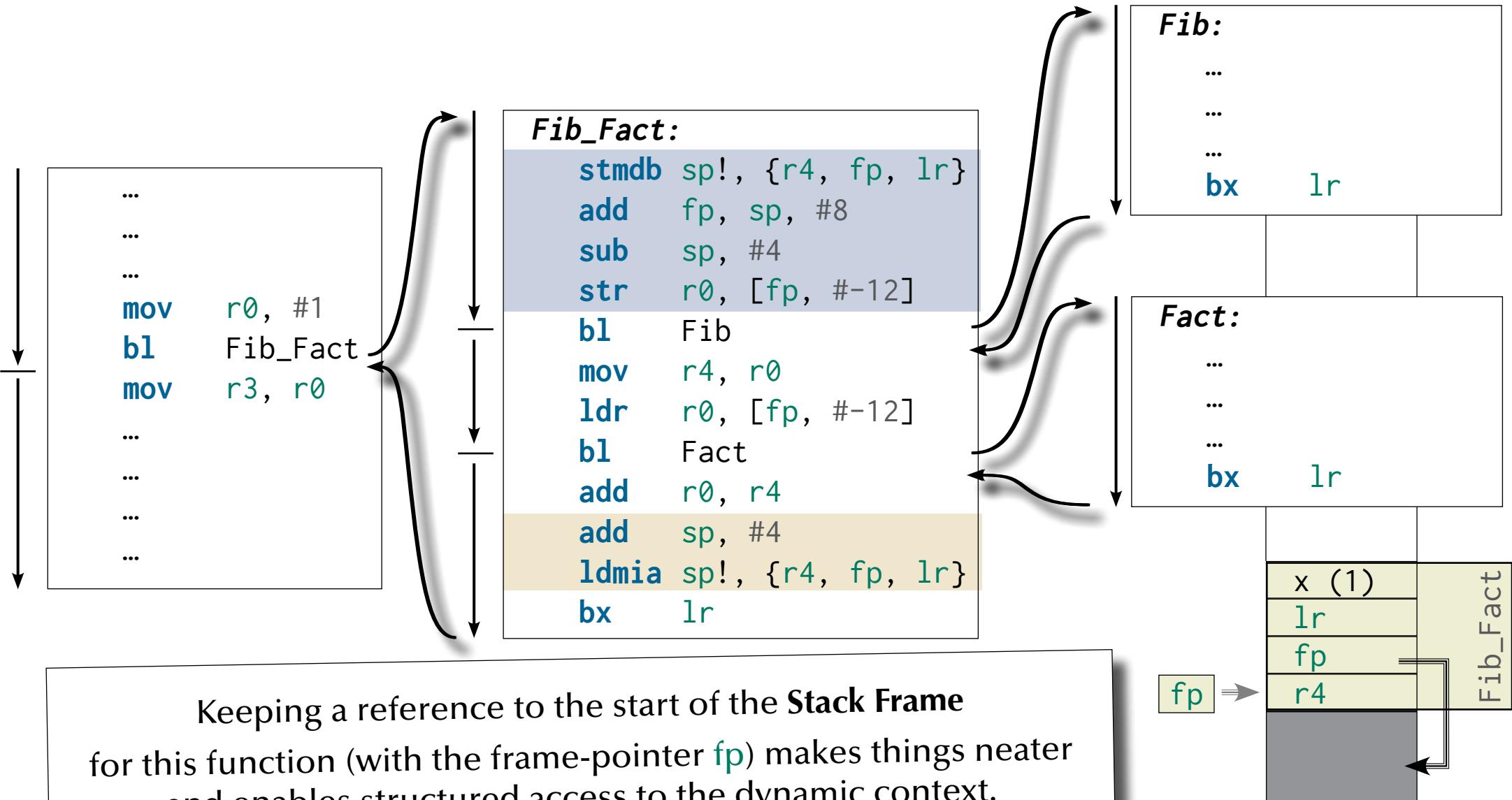


Functions





Functions





Functions

Recursive

```
unsigned int fib (unsigned int x) {
    switch (x) {
        case 0 : return 0;
        case 1 : return 1;
        default : return fib (x - 1) + fib (x - 2);
    } }
```

```
function Fib (x : Natural) return Natural is
    (case x is
        when 0      => 0,
        when 1      => 1,
        when others => Fib (x - 1) + Fib (x - 2));
```

```
unsigned int fact (unsigned int x) {
    if (x == 0) return 1;
    else        return x * fact (x - 1);
}
```

```
function Fact (x : Natural) return Positive is
    (if x = 0 then 1
     else x * Fact (x - 1));
```

Will our stack handling work
for recursive functions?



Functions

☞ Is Fact **reentrant**?

Fact:

```
stmdb sp!, {fp, lr}
add fp, sp, #4
sub sp, #4
str r0, [fp, #-8]
cmp r0, #0
bne Case_Others
mov r0, #1
b End_Fact
```

Case_Others:

```
sub r0, #1
bl Fact
mov r1, r0
ldr r0, [fp, #-8]
mul r0, r1
```

End_Fact:

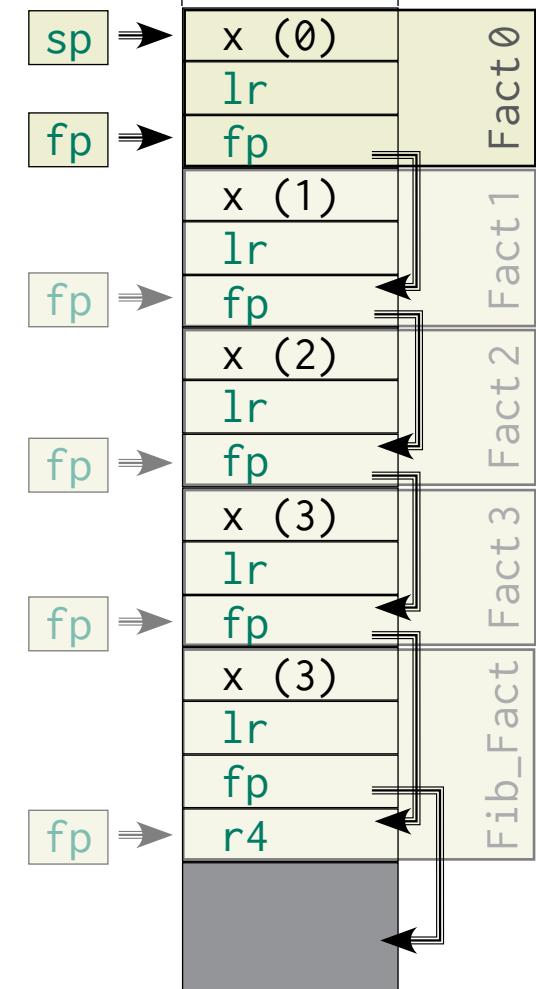
```
add sp, #4
ldmia sp!, {fp, lr}
bx lr
```

Fib_Fact:

```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

☞ Where is the last lr stored?

☞ How high do we stack?

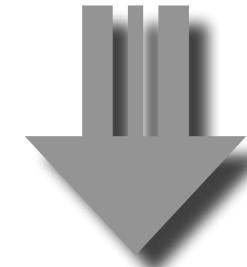




Functions

```
function Fact (x : Natural) return Positive is
    (if x = 0 then 1
     else x * Fact (x - 1));
```

☞ A compiler will likely replace such a recursion!



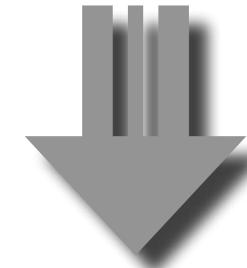
```
function Fact (x : Natural) return Positive is
    Fac : Positive := 1;
begin
    for i in 1 .. x loop
        Fac := Fac * i;
    end loop;
    return Fac;
end Fact;
```



Functions

```
unsigned int fact (unsigned int x) {  
    if (x == 0) return 1;  
    else          return x * fact (x - 1);  
}
```

☞ A compiler will likely replace such a recursion!



```
unsigned int fact (unsigned int x) {  
    int fac = 1;  
    for (i = 1, i <= x, i++) {  
        fac = fac * i;  
    }  
    return fac;  
}
```



Functions

Fib_Fact:

```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

Fib:

```
...
...
...
bx lr
```

Fact:

```
adds r3, r0, #0
mov r0, #1
beq End_Fact
```

Fact_Loop:

```
mul r0, r3
subs r3, #1
bne Fact_Loop
```

End_Fact:

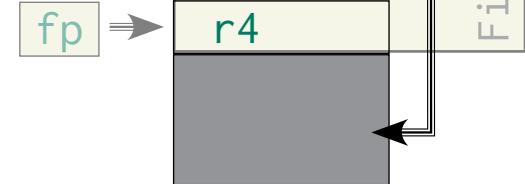
```
bx lr
```

☞ Complexity?
Runtimes?

☞ Is Fact **reentrant**?

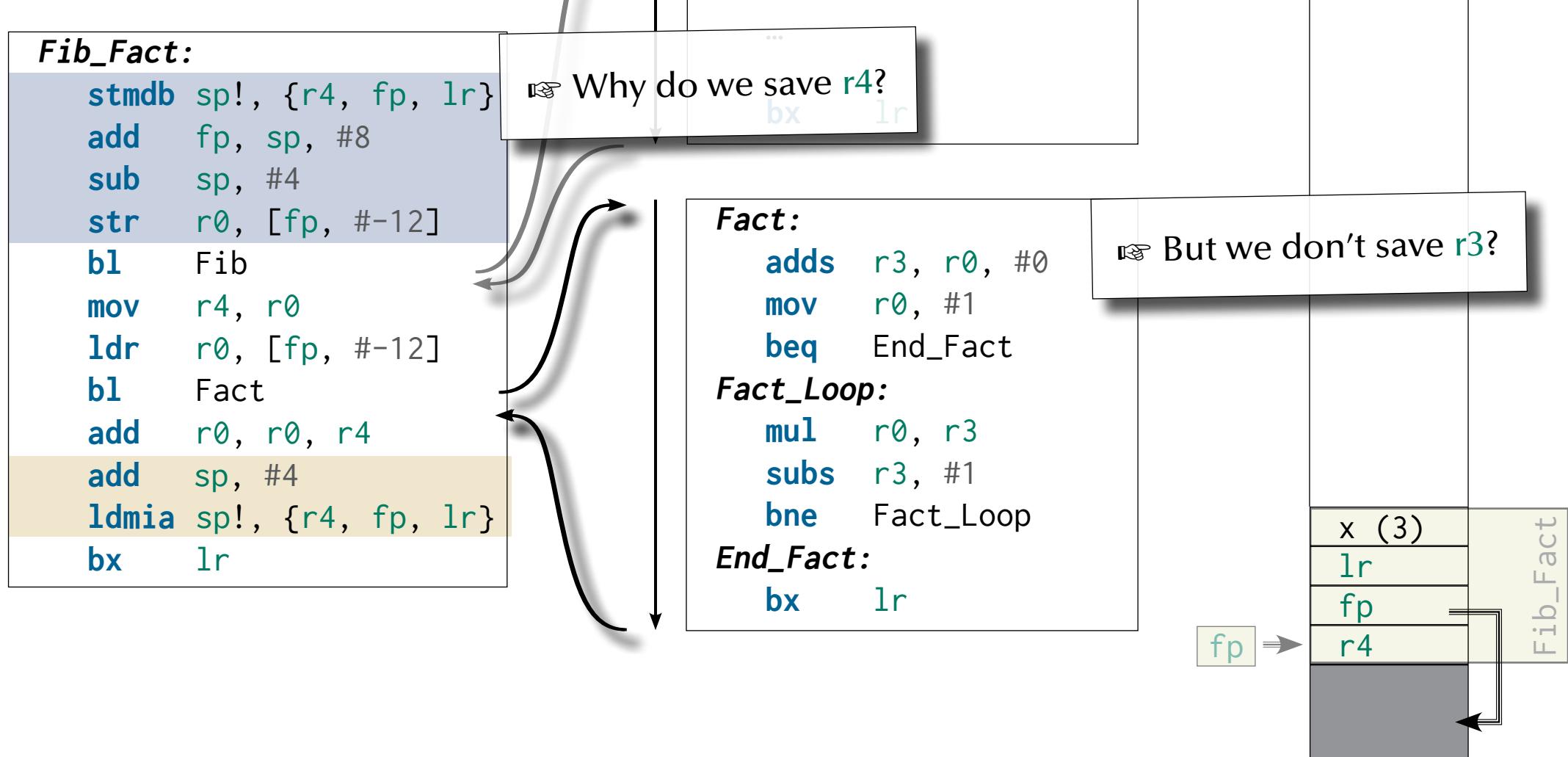
Besides all the inlining, unrolling, flattening, etc.:

☞ Stack operations are still vital for any non-trivial program.



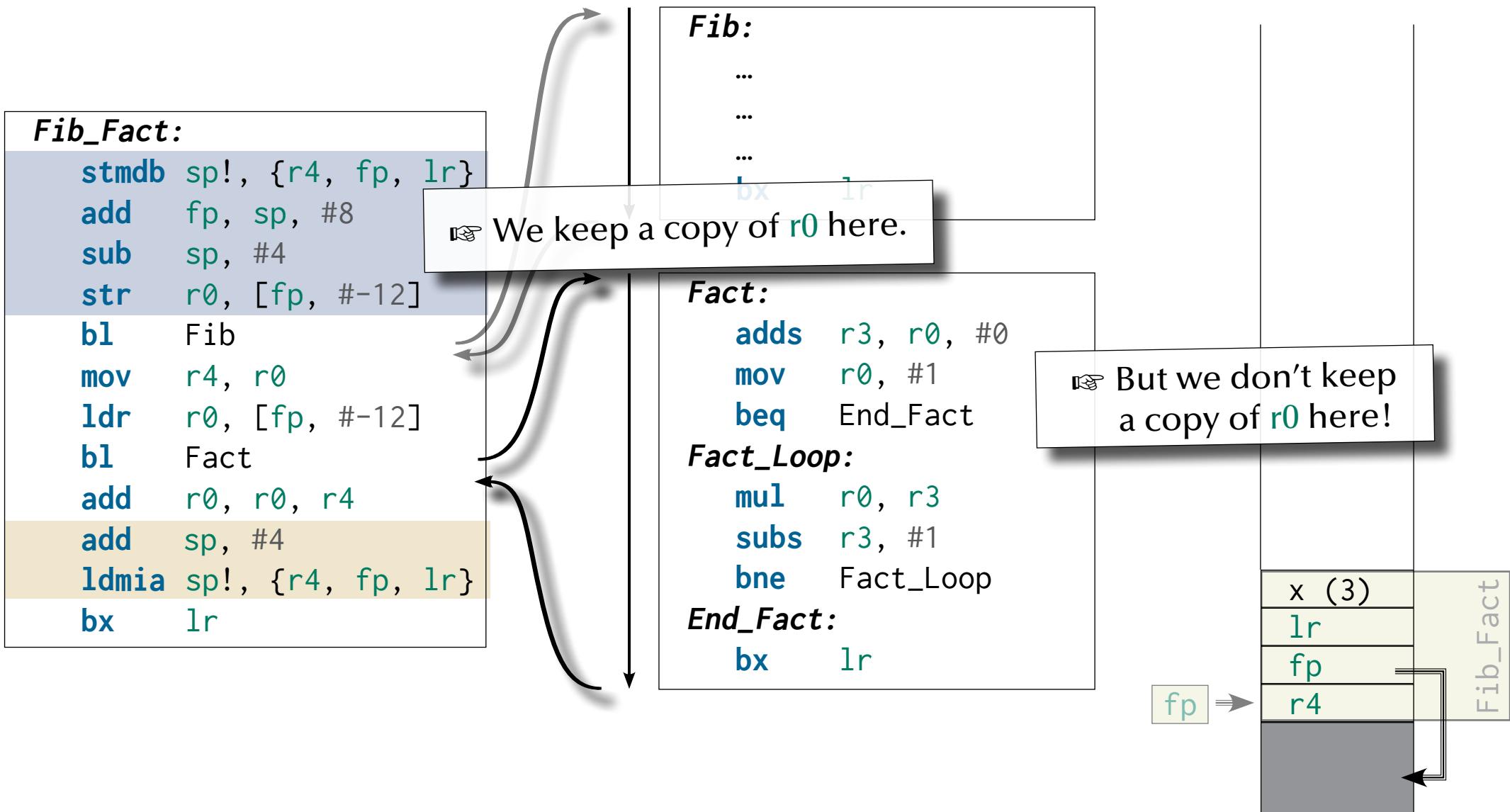


Functions





Functions





Functions

Fib_Fact:

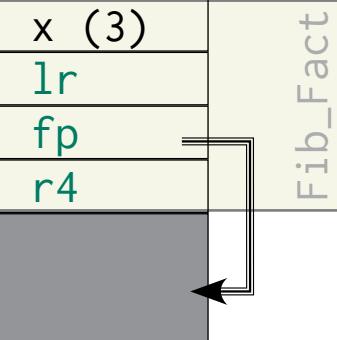
```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

Fib:

```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
cmp r0, #0
beq End_Fib
cmp r0, #1
beq End_Fib
sub r0, r0, #1
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
sub r0, r0, #2
bl Fib
add r0, r4, r0
```

End_Fib:

```
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```





Functions

Fib_Fact:

```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

☞ There could be two further Fib-calls for each call to Fib ...

Fib:

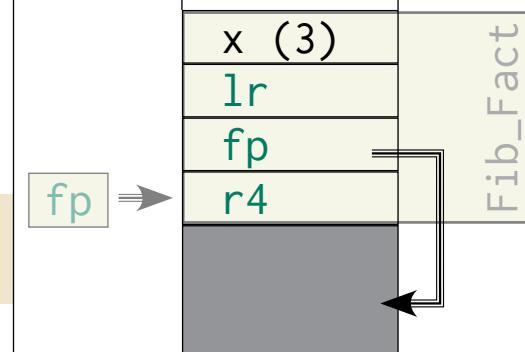
```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
cmp r0, #0
beq End_Fib
cmp r0, #1
beq End_Fib
sub r0, r0, #1
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
sub r0, r0, #2
bl Fib
add r0, r4, r0
```

End_Fib:

```
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

☞ What would be the maximal depth for the stack?

☞ What would the stack look like?





Functions

Components / phases of a function call:

- Values (**parameters**) to be passed to a function.
- **Local variables** inside a function.
- Values (**results**) to be returned from a function.

So far we:

- ☞ ... **passed parameter values** in registers (**r0 - r3**).
- ☞ ... **called the function** (store the return address and jump to the beginning of the function).
- ☞ ... **pushed** the return address, the previous stack frame and used registers (**r4 ...**). Prologue
- ☞ ... **created a new stack frame** (and addressed all local variables relative to this).
- ☞ ... **grew the stack** such that it can hold the local variables.
- ☞ ... **done the calculations/operations** based on the local variables and scratch registers.
- ☞ ... **passed return values** in registers (**r0 - r1**).
- ☞ ... **restored the stack pointer** (and thus de-allocated all local variables). Epilogue
- ☞ ... **popped** the return address, the previous stack frame and used registers (**r4 ...**).
- ☞ ... **jumped back** to the next instruction after the original function call.
- ☞ ... **used the return values** found in **r0 - r1**.



Functions

Components / phases of a function call:

- Values (**parameters**) to be passed to a function.
- **Local variables** inside a function.
- Values (**results**) to be returned from a function.

☞ What happens if the parameters or return values won't fit into registers?

So far we:

- ☞ ... **passed parameter values in registers (r0 - r3).**
- ☞ ... **called the function** (store the return address and jump to the beginning of the function).
- ☞ ... **pushed** the return address, the previous stack frame and used registers (r4 ...).
- ☞ ... **created a new stack frame** (and addressed all local variables relative to this).
- ☞ ... **grew the stack** such that it can hold the local variables.
- ☞ ... **done the calculations/operations** based on the local variables and scratch registers.

Prologue

- ☞ ... **passed return values in registers (r0 - r1).**
- ☞ ... **restored the stack pointer** (and thus de-allocated all local variables).
- ☞ ... **popped** the return address, the previous stack frame and used registers (r4 ...).
- ☞ ... **jumped back** to the next instruction after the original function call.
- ☞ ... **used the return values** found in r0 - r1.

Epilogue



Functions

Conventions

ARM architecture calling practice

- r0-r3 are used for parameters.
 - r0-r1 are used for return values.
 - r0-r3 are not expected to be intact after a function call
... all other registers are expected to be intact!
- ☞ If those registers do not suffice, additional parameters and results are passed via the stack.

☞ There are also memory alignment constraints.
(Mostly due to memory bus constraints)

☞ Conventions are different in other architectures (e.g. x86, where parameters are generally passed via the stack).

☞ Why are these conventions architecture related at all?



Functions

Parameter passing

Call by ...

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



Functions

C

Full control over those three modes.

“by value” parameters are local variables.

“in & out, by reference” syntactically as “by pointer value” parameters.

by copy

by reference

Information flow		in	by value	by reference (immutable)
		out	by result	by reference (mutable, no read)
		in & out	by value result	by reference (mutable)
		Parameter becomes a constant inside the function or is copied into a local variable.	Calling function expects the parameter value to appear in a specific space at return.	Function can read and write at any time. Outside code shall not write.



Functions

Haskell

Only control over information flow – not over access.

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and “in & out” parameters are side-effecting and can therefore not exist in a pure functional language.	by reference (mutable) Function can read and write at any time.



Functions

Python

All parameter access is double-indirect (handles).

... this has an impact on performance.

Information flow	in	by copy	Access	by reference
		by value	by reference (immutable)	
	out	by result	by reference (mutable, no read)	
	in & out	by value result	by reference (mutable)	
Parameter becomes a constant inside the function or is copied into a local variable.		No write access is allowed while the function runs (also from outside the function).		
Calling function expects the parameter value to appear in a specific space at return.		No read access from inside the function, write access on return.		
Parameter is copied to a local variable and copied back at return.		Function can read and write at any time. Outside code shall not write.		



Functions

Ada

Limited control over “by value result”.
“by value” parameters are constants.

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



Functions

Assembly

“By reference” semantics by convention only.

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



Functions

Parameter passing

Call by name

Pops up in programming languages since the 60's.

... is conceptually a call-by-value, where the value has not been calculated yet.

Technically a reference to a function is passed and the evaluation of this parameter (function) is left to the called function.

Features:

- Values are only evaluated if and when they are needed.
- Values can change during the life-time of a function (in case of side-effecting functions).
- Values can be stored once calculated (in case of side-effect-free functions).

While this is possible to a degree in most programming languages ...

(even if there is no specific passing mode, you can still pass a reference to a function)

... it is a core concept for functional, lazy evaluation languages, like e.g. Haskell,

and it does find its way back into mainstream languages like C++, .NET languages or Python as **anonymous functions** (sometimes referred to as λ -functions or λ -expressions).



Functions

👉 How about using call by reference here?

Fib_Fact:

```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
ldr r0, [fp, #-12]
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

Fact:

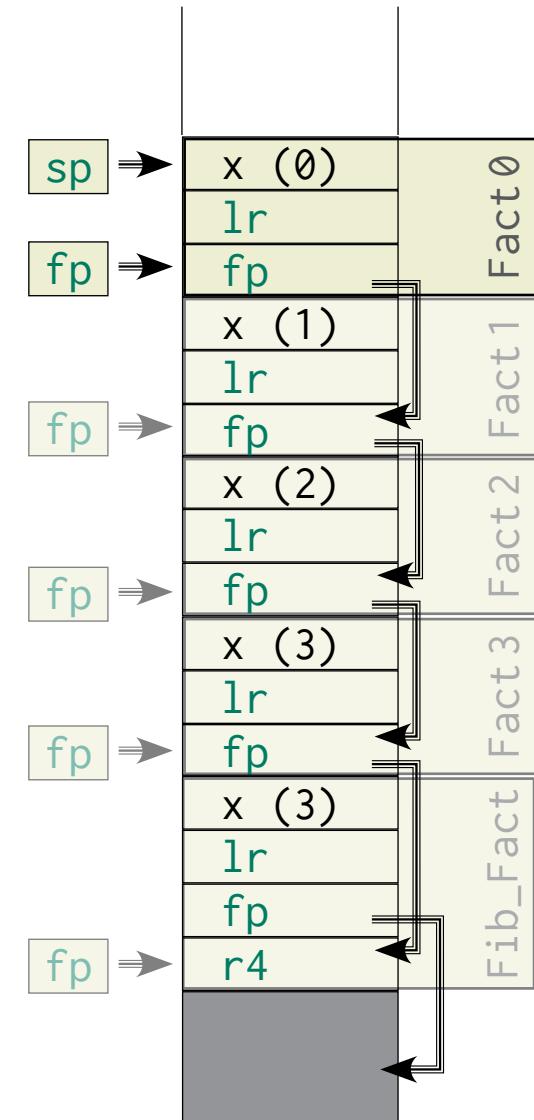
```
stmdb sp!, {fp, lr}
add fp, sp, #4
sub sp, #4
str r0, [fp, #-8]
cmp r0, #0
bne Case_Others
mov r0, #1
b End_Fact
```

Case_Others:

```
sub r0, #1
bl Fact
mov r1, r0
ldr r0, [fp, #-8]
mul r0, r1
```

End_Fact:

```
add sp, #4
ldmia sp!, {fp, lr}
bx lr
```





Functions

Fib_Fact:

```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
sub r0, fp, #12
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

r5 has been nominated to hold the reference to x inside Fact

Fact:

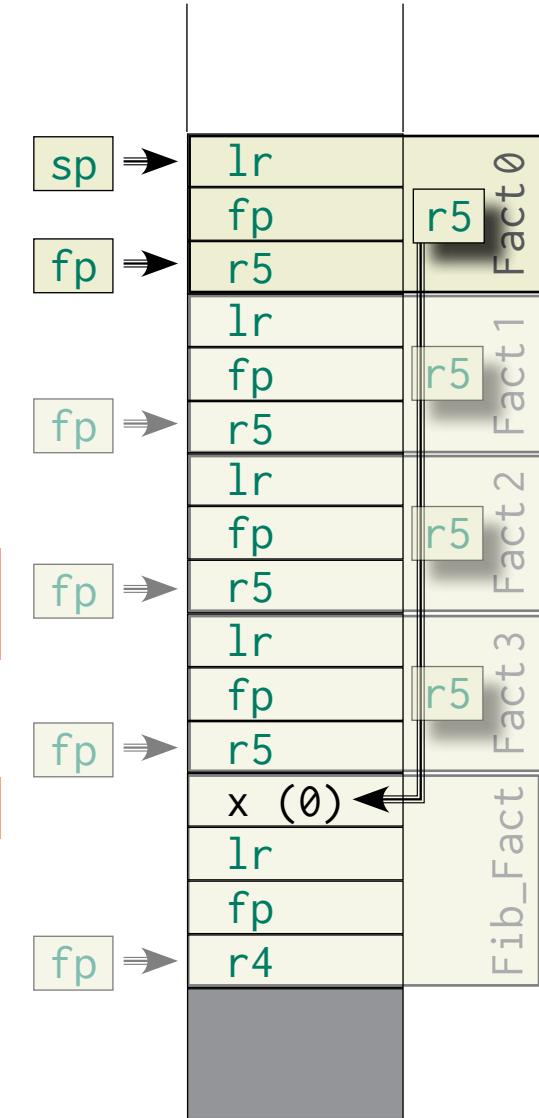
```
stmdb sp!, {r5, fp, lr}
add fp, sp, #4
mov r5, r0
ldr r0, [r5]
cmp r0, #0
bne Case_Others
mov r0, #1
b End_Fact
```

Case_Others:

```
sub r0, #1
str r0, [r5]
mov r0, r5
bl Fact
mov r1, r0
ldr r0, [r5]
mul r0, r1
```

End_Fact:

```
mov sp, fp
ldmia sp!, {r5, fp, lr}
bx lr
```





Functions

Fib_Fact:

```
stmdb sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, #4
str r0, [fp, #-12]
bl Fib
mov r4, r0
sub r0, fp, #12
bl Fact
add r0, r0, r4
add sp, #4
ldmia sp!, {r4, fp, lr}
bx lr
```

👉 We turned Fact into the constant 0.

Fact:

```
stmdb sp!, {r5, fp, lr}
add fp, sp, #4
mov r5, r0
ldr r0, [r5]
cmp r0, #0
bne Case_Others
mov r0, #1
b End_Fact
```

Case_Other:

```
sub r0, r0, r0
str r0, [r5]
mov r0, r5
bl Fact
mov r1, r0
ldr r0, [r5]
mul r0, r1
```

End_Fact:

```
mov fp, sp!
ldmia fp, {r5, fp, lr}
bx lr
```

👉 What did we overlook?

lr	r5	Fact 0
fp		
r5		Fact 1
lr	r5	
fp		
r5		Fact 2
lr	r5	
fp		
r5		Fact 3
lr	r5	
fp		
r5		
x (0)		
lr		
fp		
r4		



Functions

Fib_Fact:

```
stmdb sp!, {r5, fp, lr}
add    fp, sp, #4
mov    r5, r0
ldr    r0, [r5]
cmp    r0, #0
bne   Case_Others
mov    r0, #1
b     End_Fact

Fib
mov    r4, r0
sub    r0, fp, #12
bl    Fact
add    r0, r0, r4
add    sp, #4
ldmia sp!, {r4, fp, lr}
bx    lr
```

What is the value of x during one execution of Fact?

Fact:

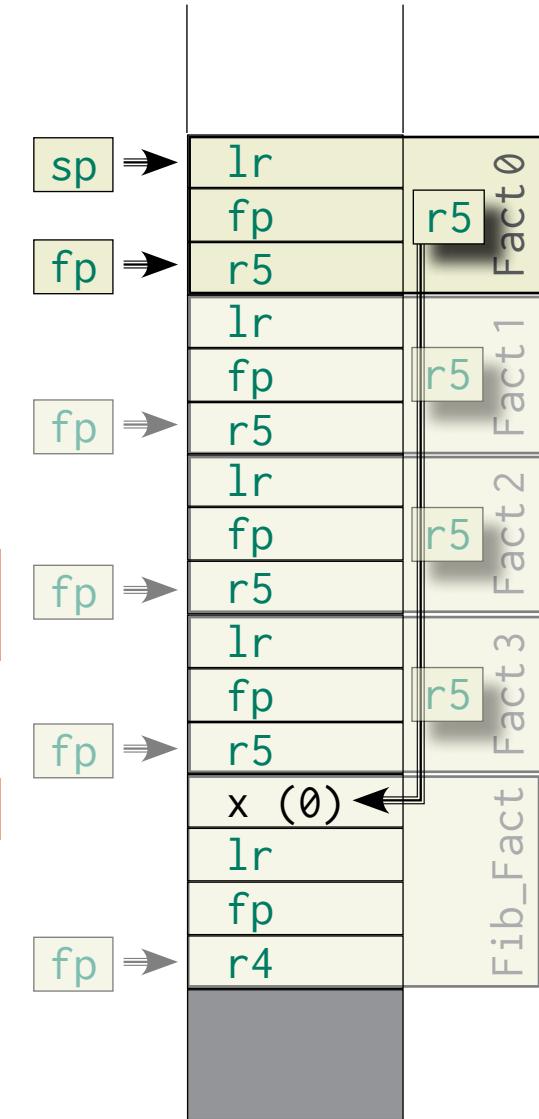
```
stmdb sp!, {r5, fp, lr}
add    fp, sp, #4
mov    r5, r0
ldr    r0, [r5]
cmp    r0, #0
bne   Case_Others
mov    r0, #1
b     End_Fact
```

Case_Others:

```
sub    r0, #1
str    r0, [r5]
mov    r0, r5
bl    Fact
mov    r1, r0
ldr    r0, [r5]
mul    r0, r1
```

End_Fact:

```
mov    sp, fp
ldmia sp!, {r5, fp, lr}
bx    lr
```





Functions

Parameter passing

Call by ...

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	We should have used either of those modes <small>by reference (mutable, no read)</small> No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write. Yet we used this mode



Functions

When to use what?

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



One-way and by-copy

Those are side-effect-free and hence the resulting scenarios are easy to analyse.

Copying large data structures might be time consuming or infeasible.

Values can be passed in registers.

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



Two-way and by-copy

Still side-effect-free within the function (but not on the outside).

Potentially more convenient as memory space can be reused.

Values can be passed in registers.

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



Two-way and by-reference

Side-effecting and particular care is required as multiple entities could write on this.

No data has to be replicated.

Values have to be passed in memory.

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



One-way-Out and by-reference

Side-effect-free, if new memory is allocated on return
– cannot be enforced on assembly level (requires compiler).
Values have to be passed in memory.

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



One-way-In and by-reference

Side-effect-free – cannot be enforced on assembly level (requires compiler).

No data has to be replicated.

Values have to be passed in memory.

		Access	
		by copy	by reference
Information flow	in	by value Parameter becomes a constant inside the function or is copied into a local variable.	by reference (immutable) No write access is allowed while the function runs (also from outside the function).
	out	by result Calling function expects the parameter value to appear in a specific space at return.	by reference (mutable, no read) No read access from inside the function, write access on return.
	in & out	by value result Parameter is copied to a local variable and copied back at return.	by reference (mutable) Function can read and write at any time. Outside code shall not write.



Functions

Generic Stack-Frame

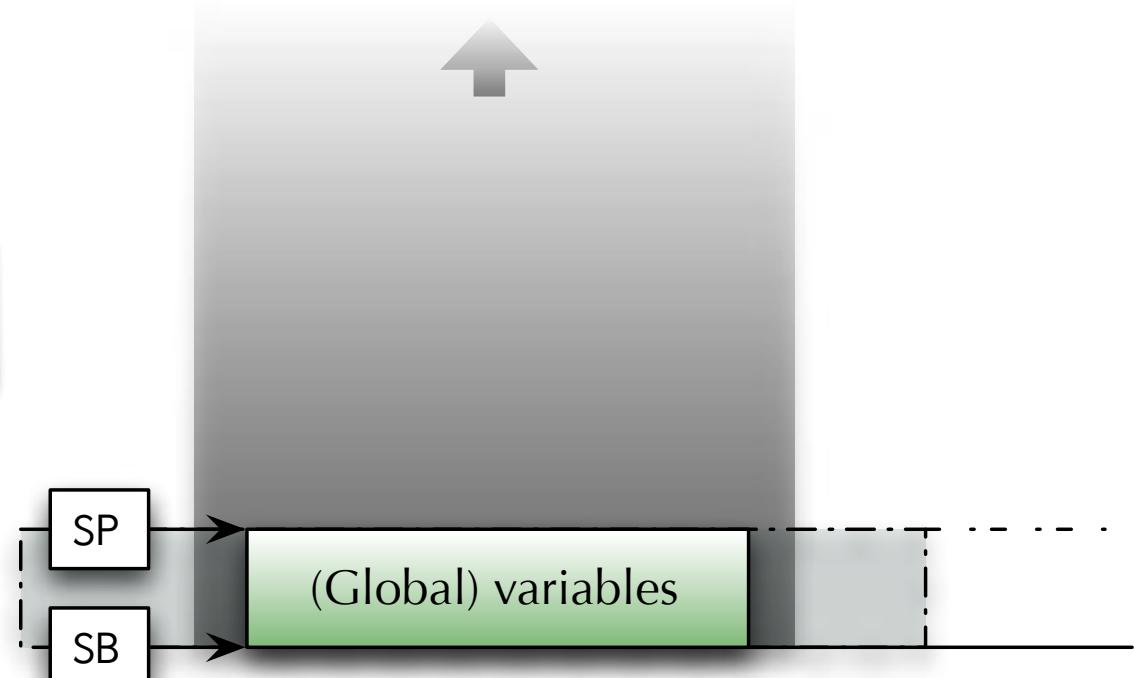
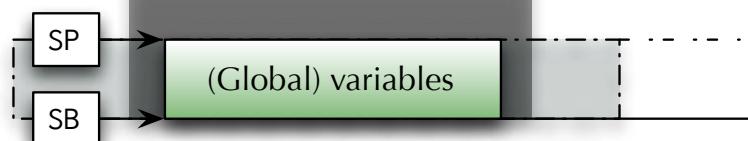


Let there be some (global) data on the stack.

Stack-Base (SB) is a static address,
always pointing to the ... well.

Stack-Pointer (SP) points to the current top of the stack.

Global variables can also
be stored someplace else.



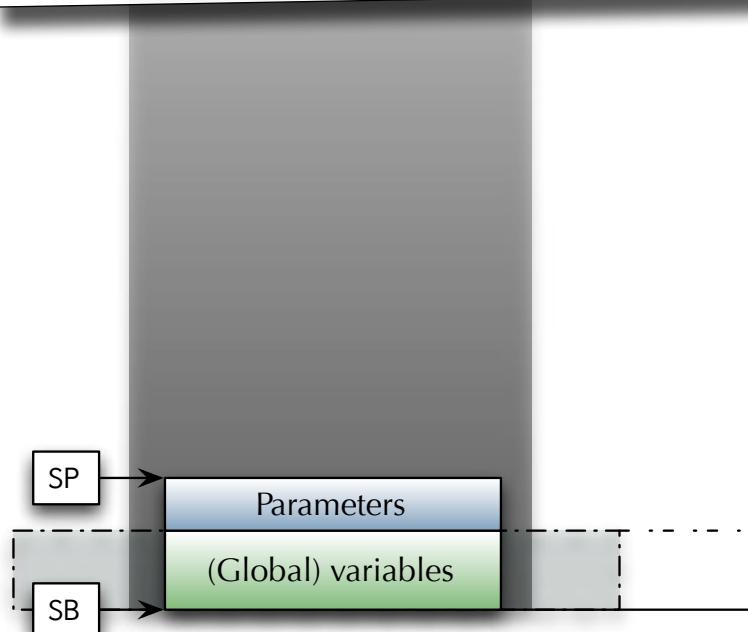


Functions

Generic Stack-Frame



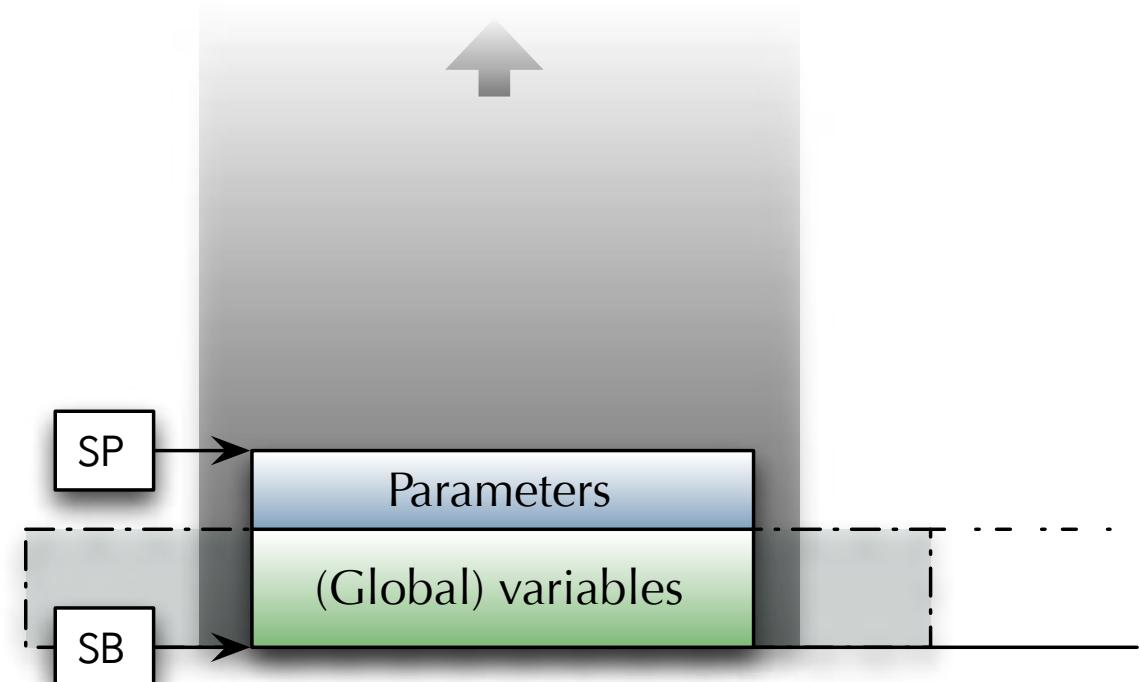
Types, storage structures and passing modes have to be agreed upon between caller and callee.



The current code prepared to call a function:

- ☞ Push parameters on the stack.

Works for any data size (unless the stack overflows) and parameter passing mode.





Functions

Generic Stack-Frame



Types, storage structures and passing modes have to be agreed upon between caller and callee.

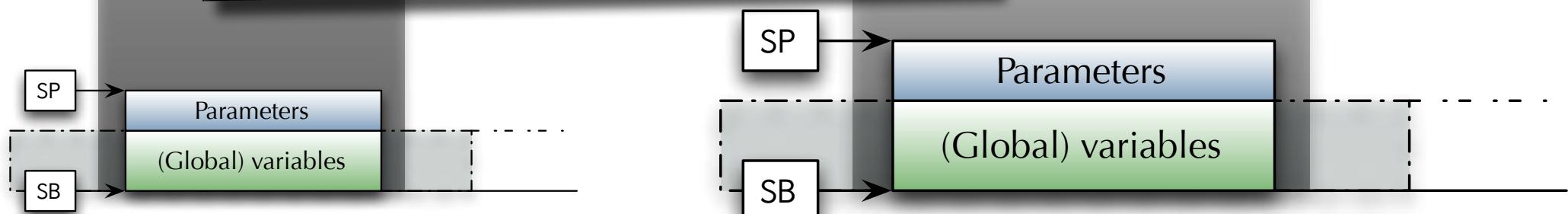
The current code prepared to call a function:

- ☞ Push parameters on the stack.

Works for any data size (unless the stack overflows) and parameter passing mode.

- ☞ Solved if it's the same language, compiler and program.

- ☞ If the languages or the compilers are different, then standards will be required.





Functions

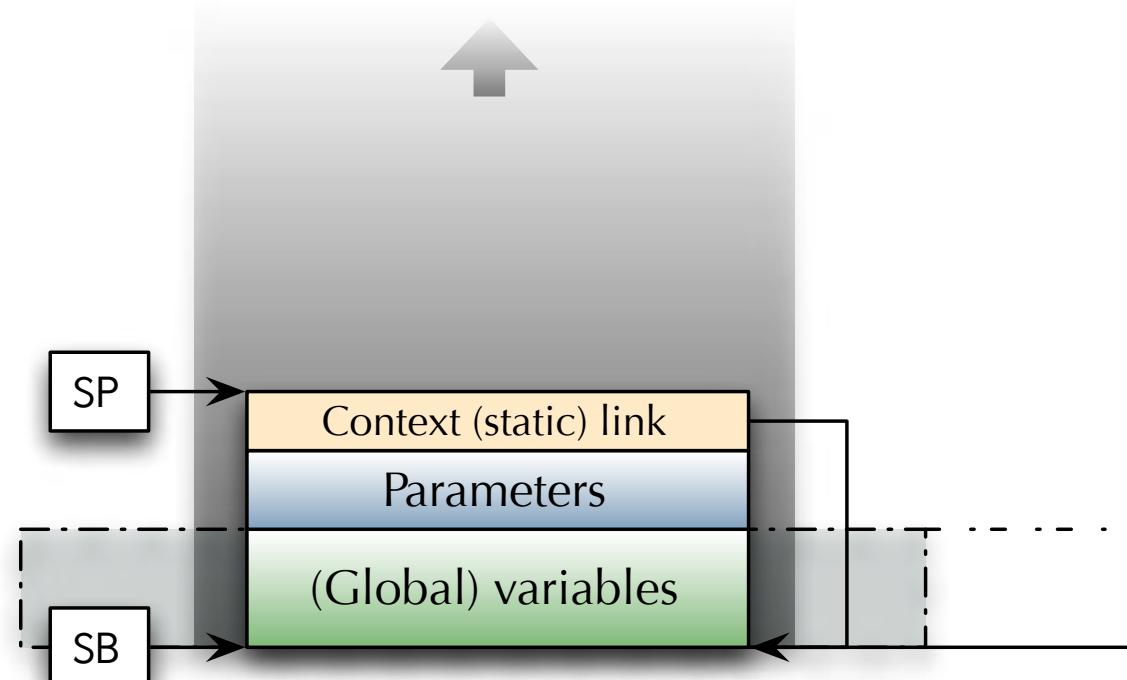
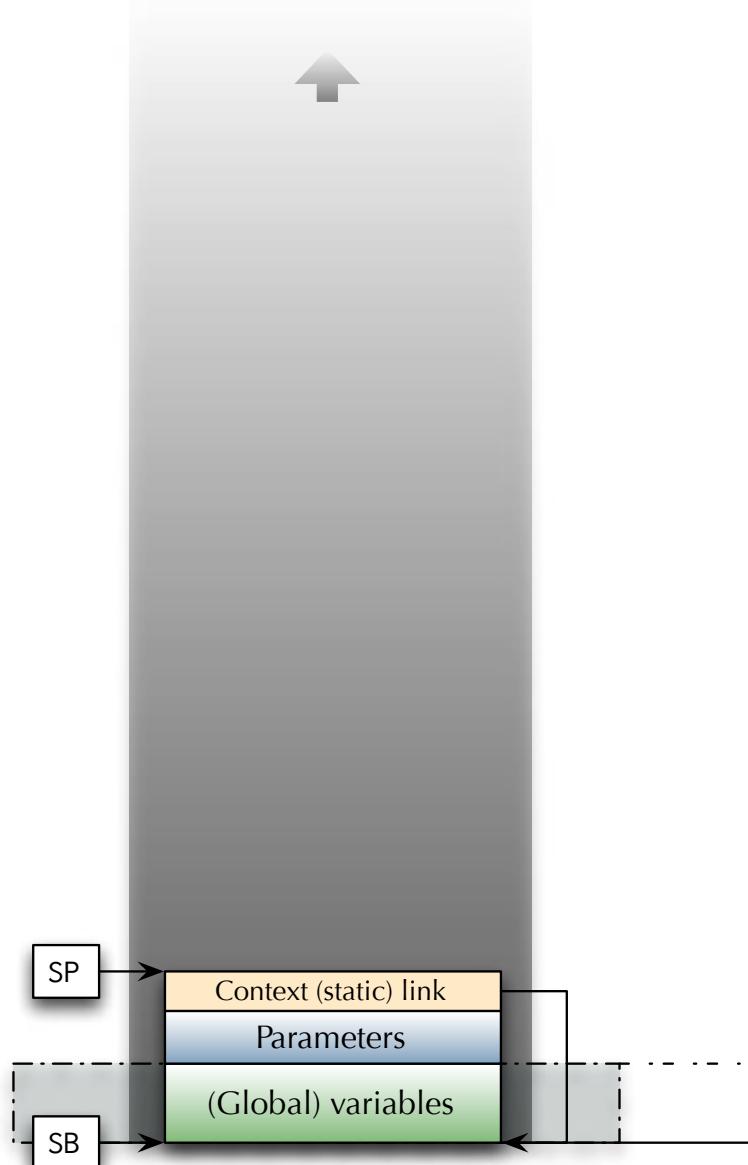
Generic Stack-Frame



Functions (in programming languages) have a context.

☞ E.g. the *surrounding function* or the *hosting object*.

The caller knows this context and provides it.





Functions

Generic Stack-Frame



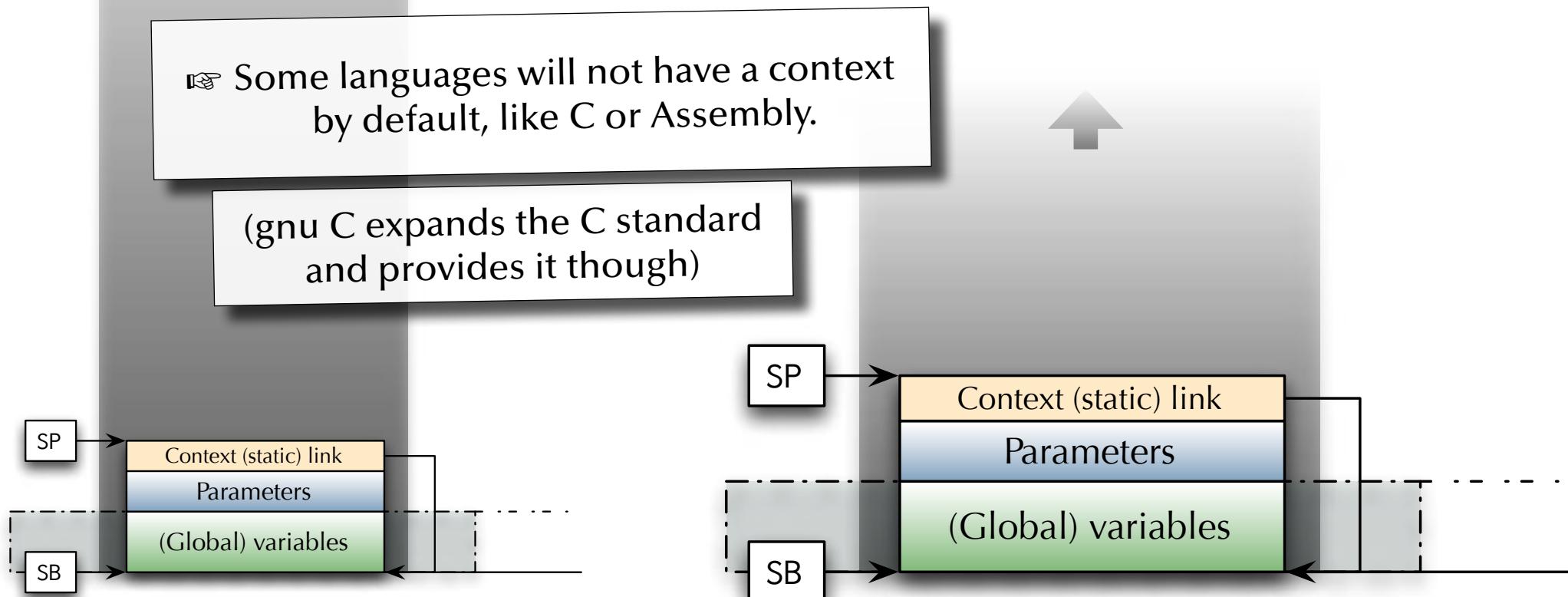
Functions (in programming languages) have a context.

☞ E.g. the *surrounding function* or the *hosting object*.

The caller knows this context and provides it.

☞ Some languages will not have a context by default, like C or Assembly.

(gnu C expands the C standard and provides it though)





Functions

Generic Stack-Frame

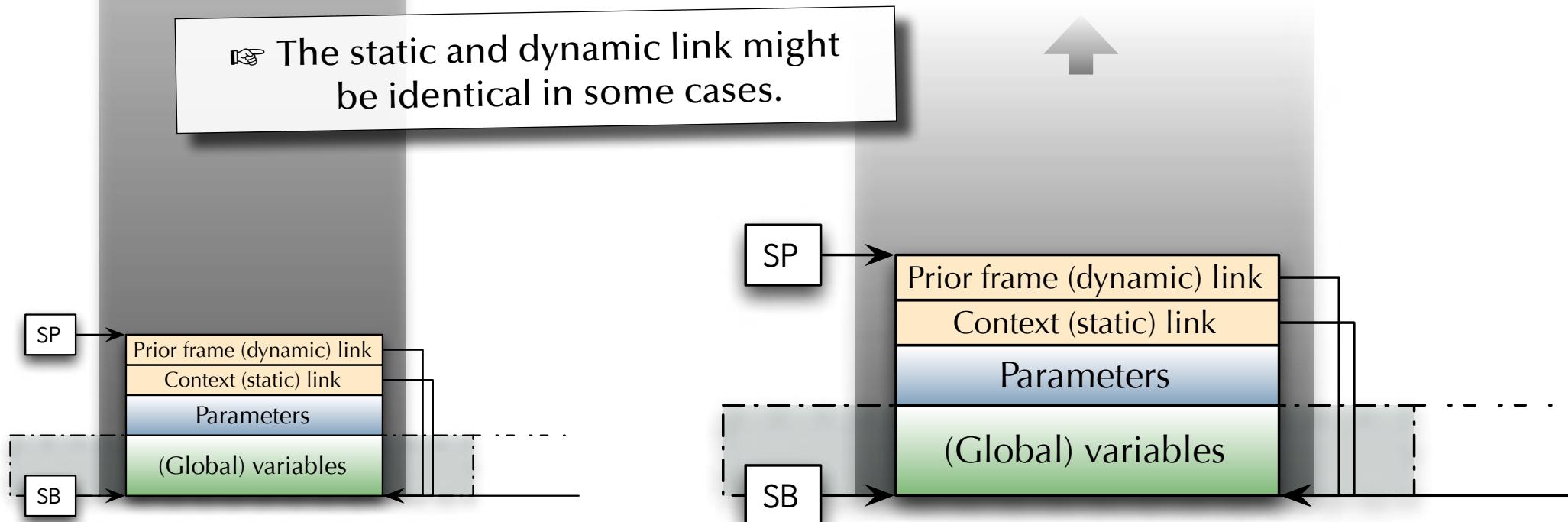


The caller also provides a reference to its own stack frame.

☞ This builds a linear chain of calls through the stack.

Will be used e.g. for debugging (stack trace) and exception propagation.

☞ The static and dynamic link might be identical in some cases.





Functions

Static vs. dynamic links

```
function a (x : Integer) return Integer is
    function b (y : Integer) return Integer is (x + y);
    function c (z : Integer) return Integer is (b (z));
begin
    return c (x);
end a;
```

```
a :: Integer -> Integer
```

```
a x = c x
```

where

```
b :: Integer -> Integer
```

```
b y = x + y
```

```
c :: Integer -> Integer
```

```
c z = b z
```



Functions

Static vs. dynamic links

```
function a (x : Integer) return Integer is
    function b (y : Integer) return Integer is (x + y);
    function c (z : Integer) return Integer is (b (z));
begin
    return c (x);
end a;
```

```
a :: Integer -> Integer
```

```
a x = c x
```

where

```
b :: Integer -> Integer
```

```
b y = x + y
```

```
c :: Integer -> Integer
```

```
c z = b z
```

☞ The **dynamic** and **static link** for function c are both function a.



Functions

Static vs. dynamic links

```
function a (x : Integer) return Integer is
    function b (y : Integer) return Integer is (x + y);
    function c (z : Integer) return Integer is (b (z));
```

```
begin
```

```
    return c (x);
```

```
end a;
```

```
a :: Integer -> Integer
```

```
a x = c x
```

where

```
b :: Integer -> Integer
```

```
b y = x + y
```

```
c :: Integer -> Integer
```

```
c z = b z
```

Dynamic link (prior frame)

☞ The caller of function b is function c.

☞ Hence exceptions raised in b are handled first in b, then in c, and then a.

☞ The **dynamic** and **static link** for function c are both function a.



Functions

Static vs. dynamic links

```
function a (x : Integer) return Integer is
    function b (y : Integer) return Integer is (x + y);
    function c (z : Integer) return Integer is (b (z));
```

begin

```
    return c (x);
```

end a;

```
a :: Integer -> Integer
```

```
a x = c x
```

where

```
b :: Integer -> Integer
```

```
b y = x + y
```

```
c :: Integer -> Integer
```

```
c z = b z
```

☞ The **dynamic** and **static link** for function c are both function a.

Dynamic link (prior frame)

☞ The caller of function b is function c.

☞ Hence exceptions raised in b are handled first in b, then in c, and then a.

Static link (context)

☞ The context for function b is function a.

☞ Hence b can access x (but not z).



Functions

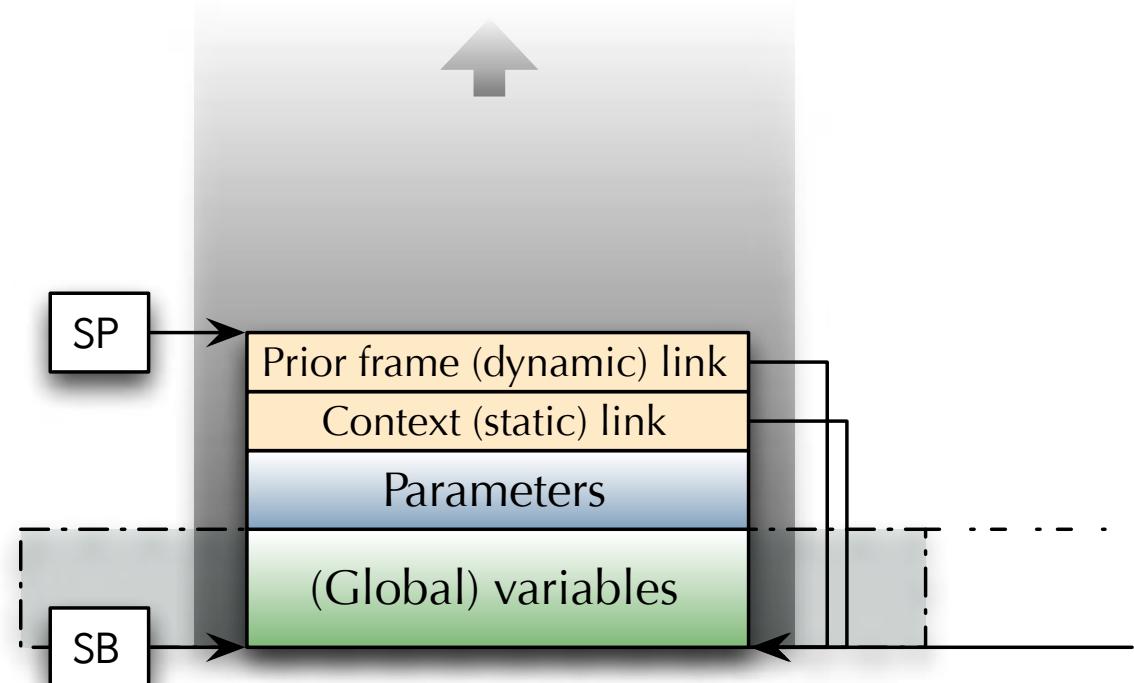
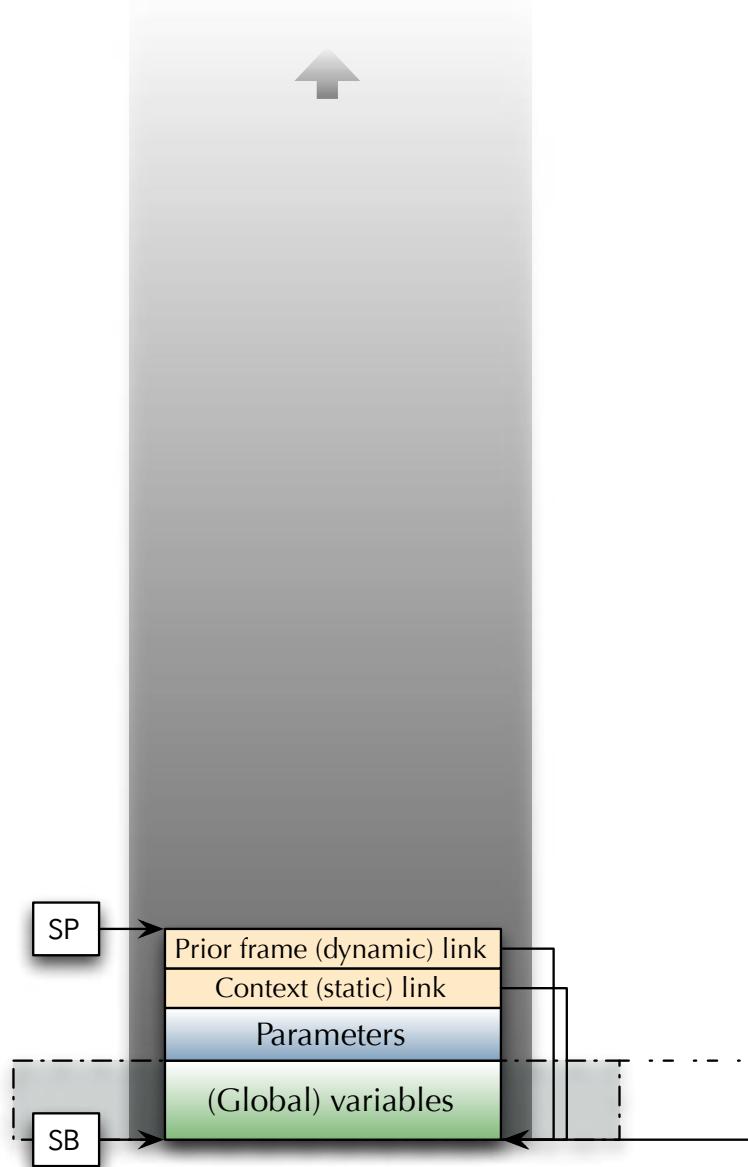
Generic Stack-Frame



The caller also provides a reference to its own stack frame.

☞ This builds a linear chain of calls through the stack.

Will be used e.g. for debugging (stack trace) and exception propagation.





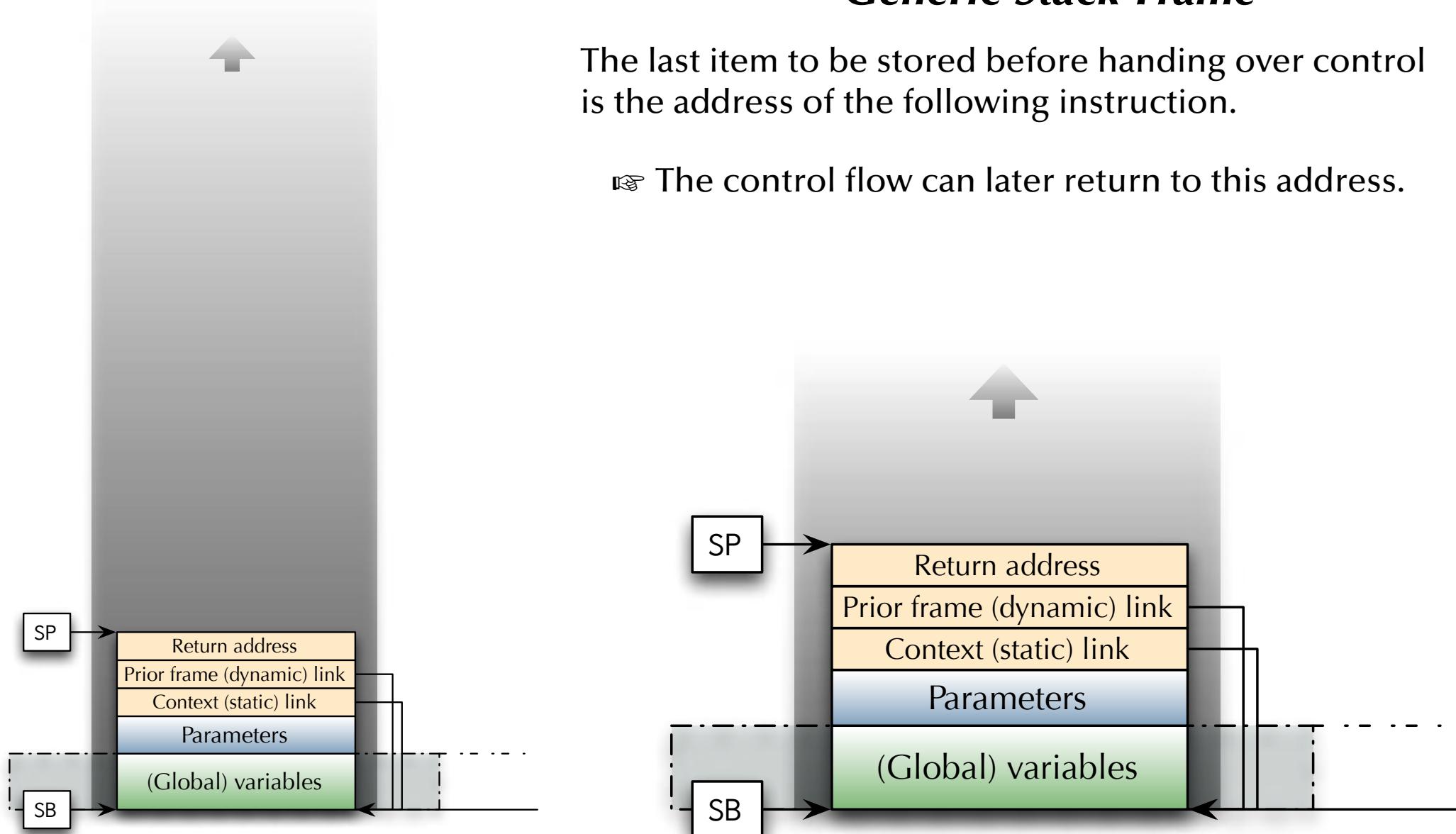
Functions

Generic Stack-Frame



The last item to be stored before handing over control is the address of the following instruction.

☞ The control flow can later return to this address.





Functions

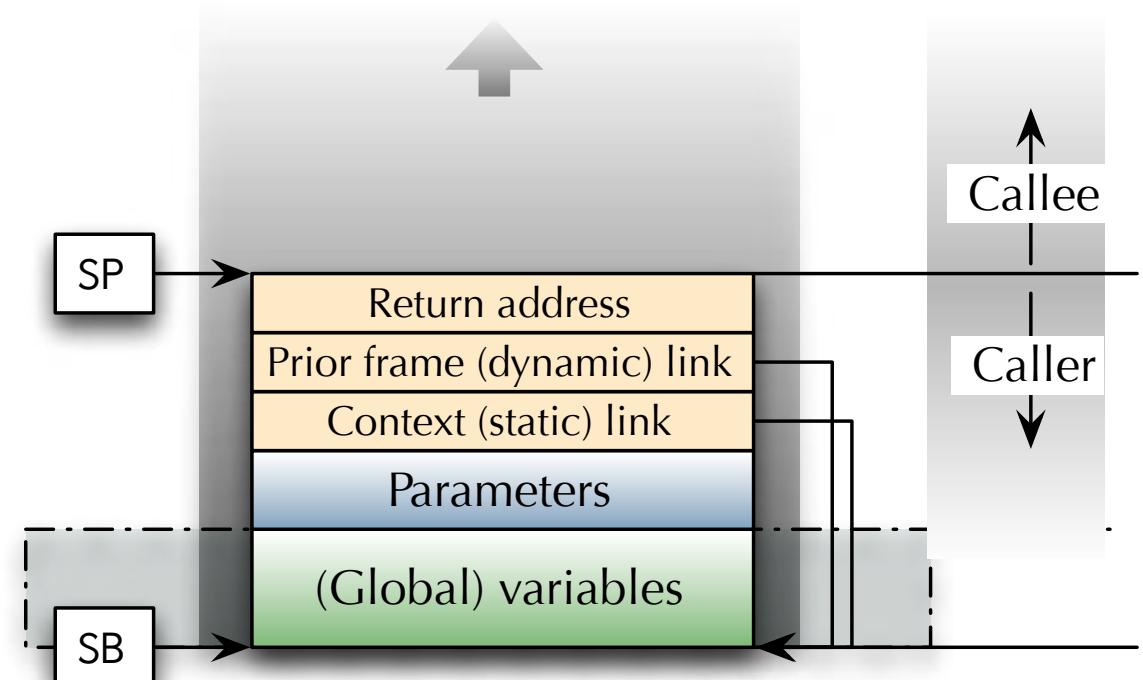
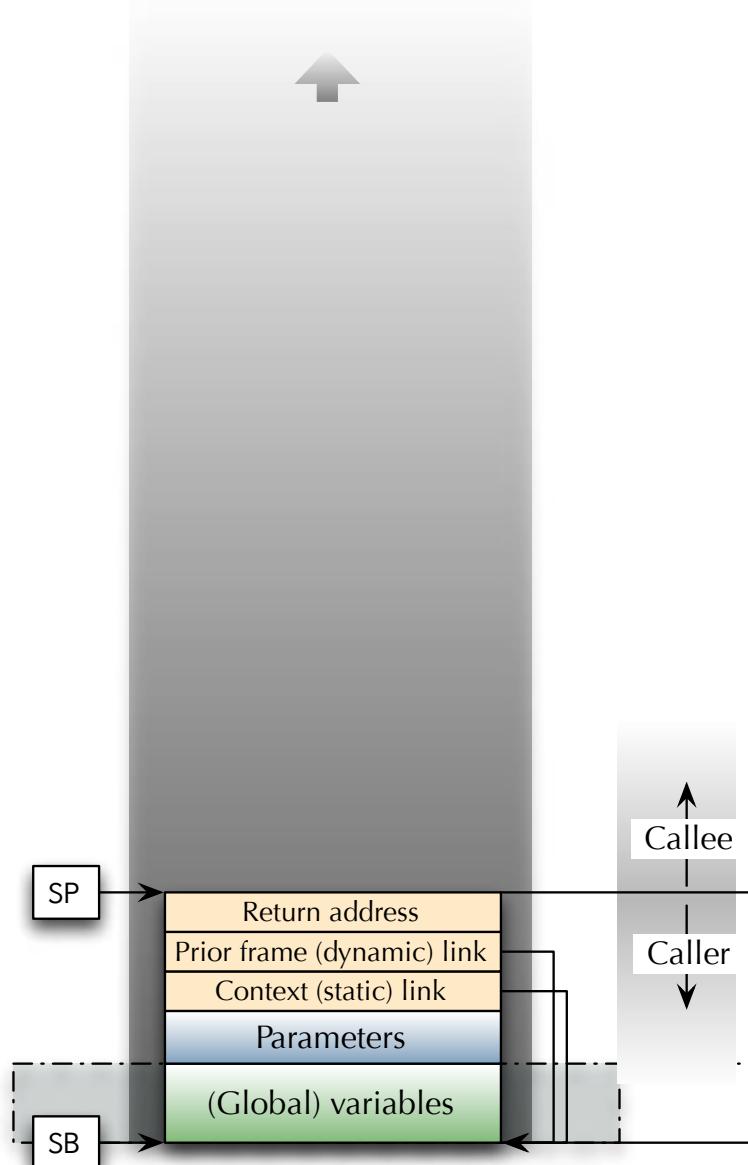
Generic Stack-Frame



Control is handed over to the callee.

- ☞ The Instruction Pointer (IP), sometimes also called Program Counter (PC) is changed to a new address.

Operations from here are in the control of the callee.





Functions

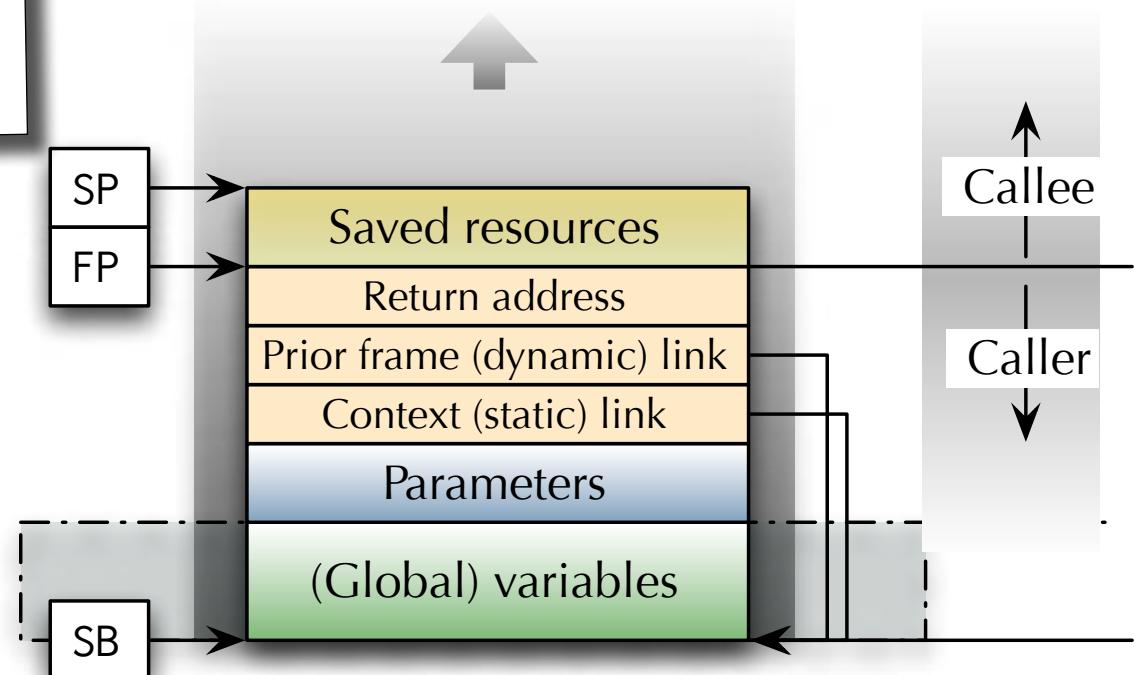
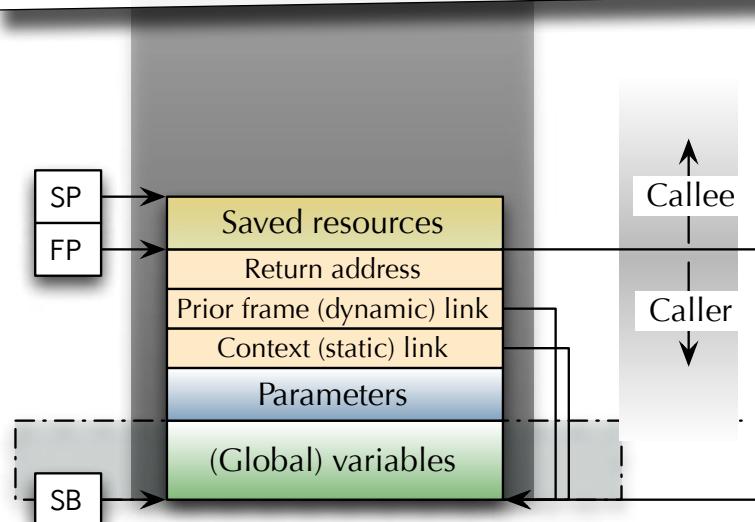
Generic Stack-Frame



A **Frame Pointer** (FP) is established at the boundary between the caller and the callee.

- ☞ Upwards from the FP: data from this function.
- ☞ Downwards from the FP:
data provided by the previous function.

Saved resources are for instance registers which the callee is planning to use.





Functions

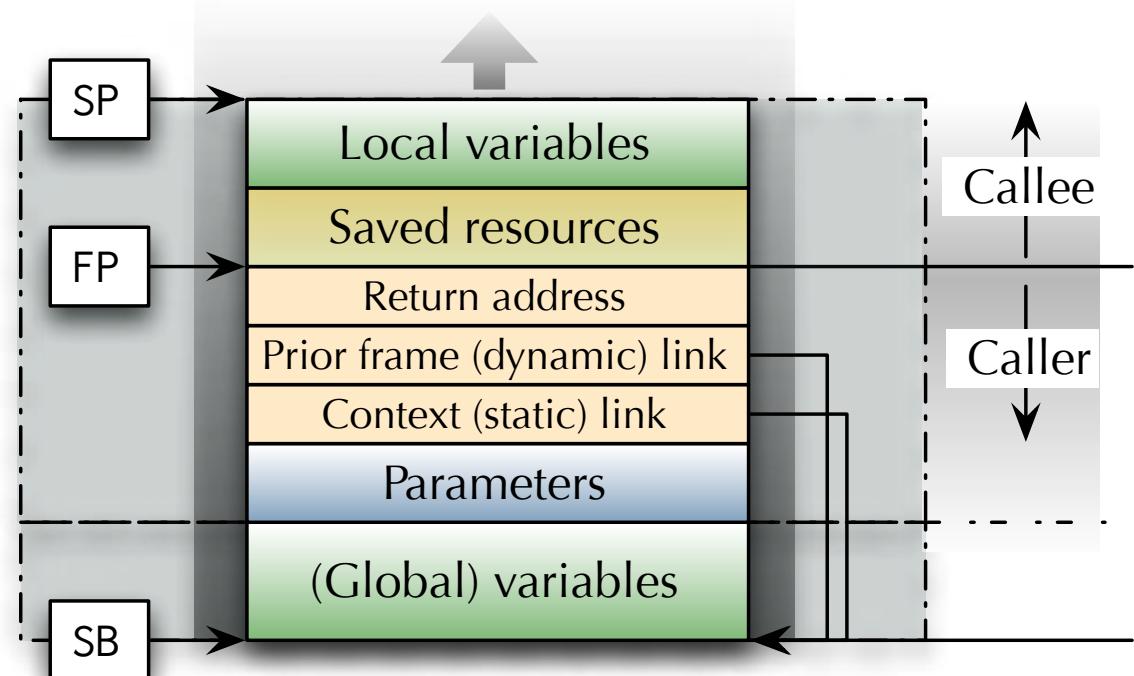
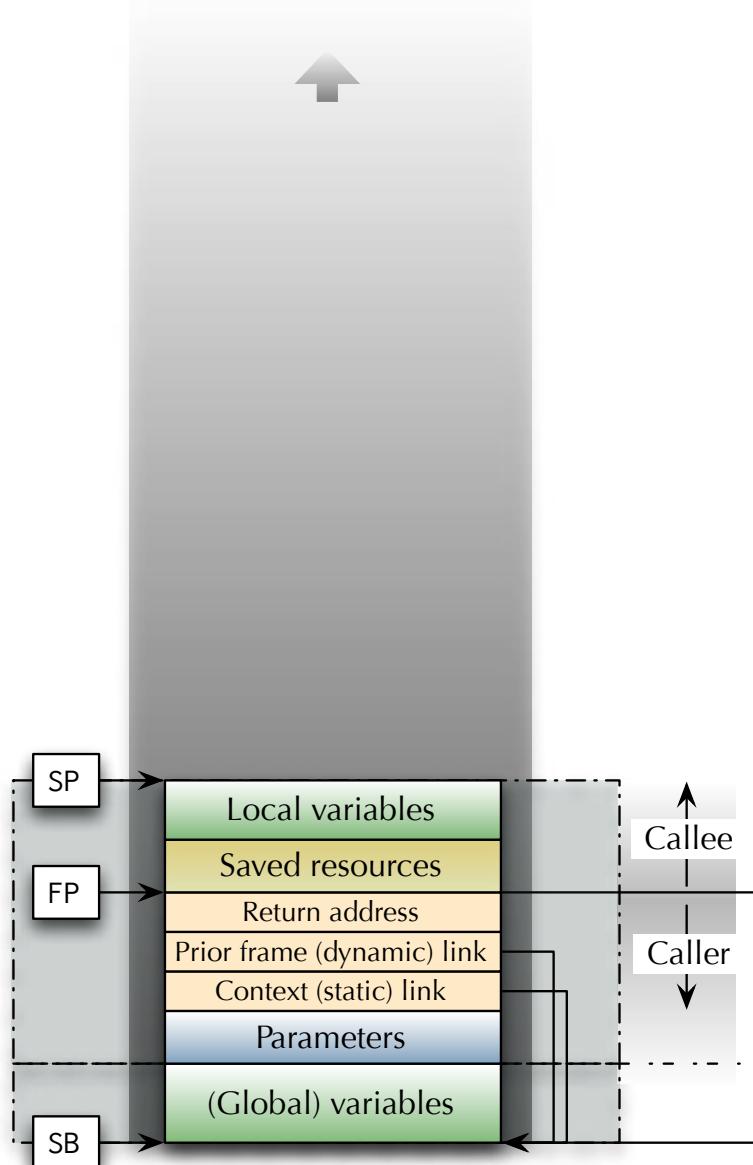
Generic Stack-Frame



Local variables are allocated (by moving the stack pointer).

Local variables can be of any size or structure, unless the stack overflows.

☞ The completes a new **stack frame**.





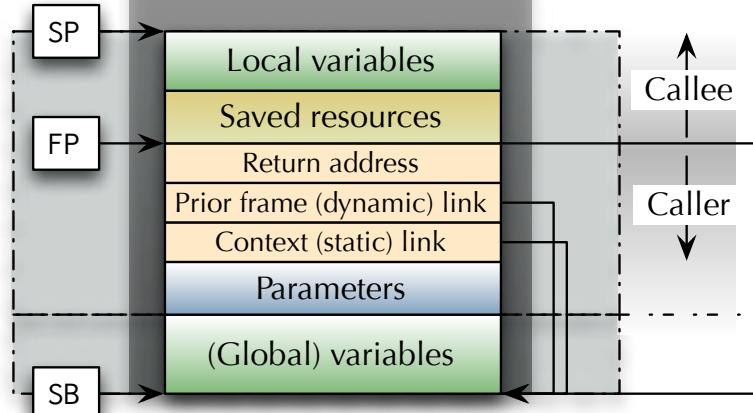
Functions

Generic Stack-Frame



While this function is executing, local variables can still be added.

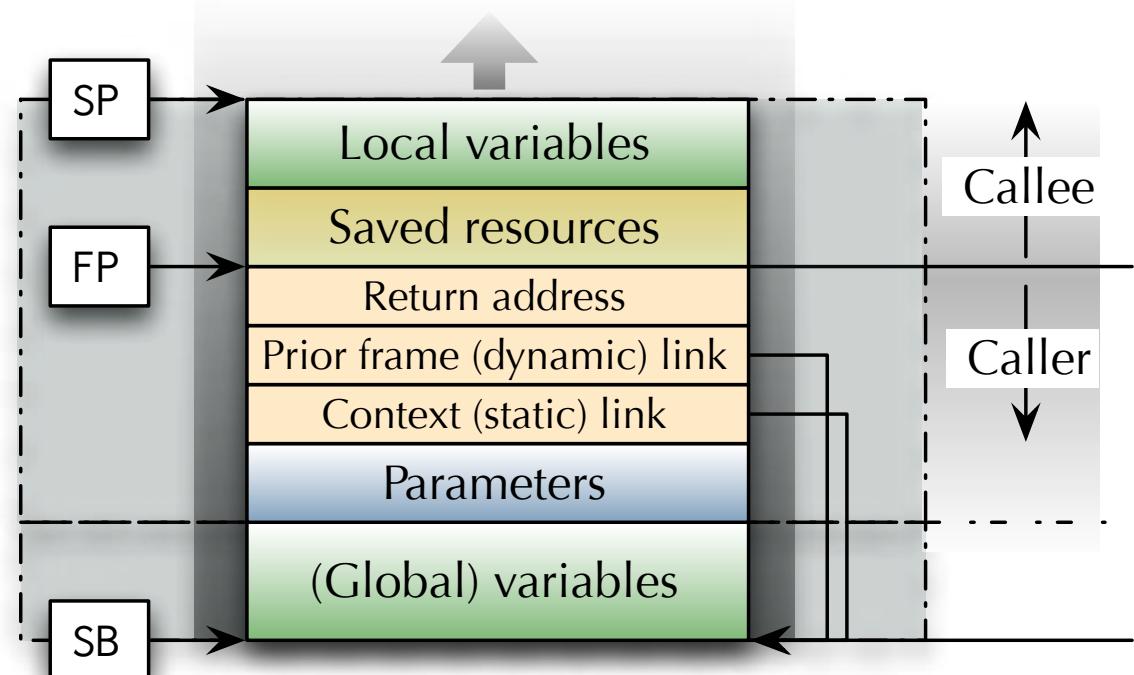
- ☞ Handy, if e.g. the size of a local variable is not yet determined when the function starts.



Local variables are allocated (by moving the stack pointer).

Local variables can be of any size or structure, unless the stack overflows.

- ☞ The completes a new **stack frame**.





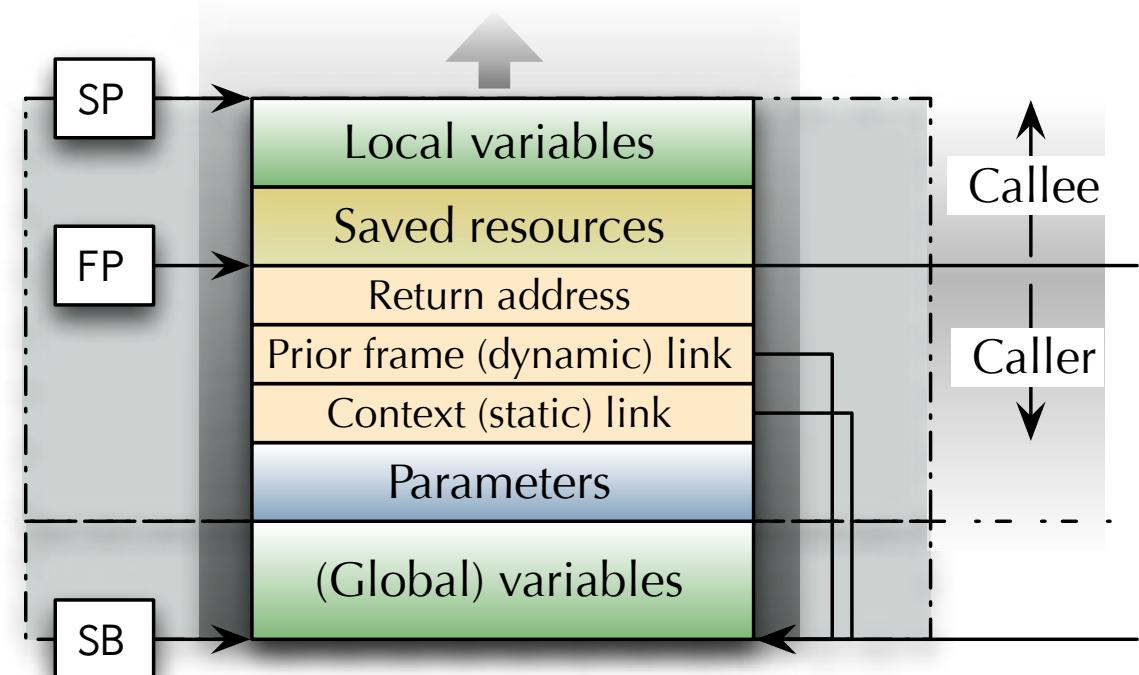
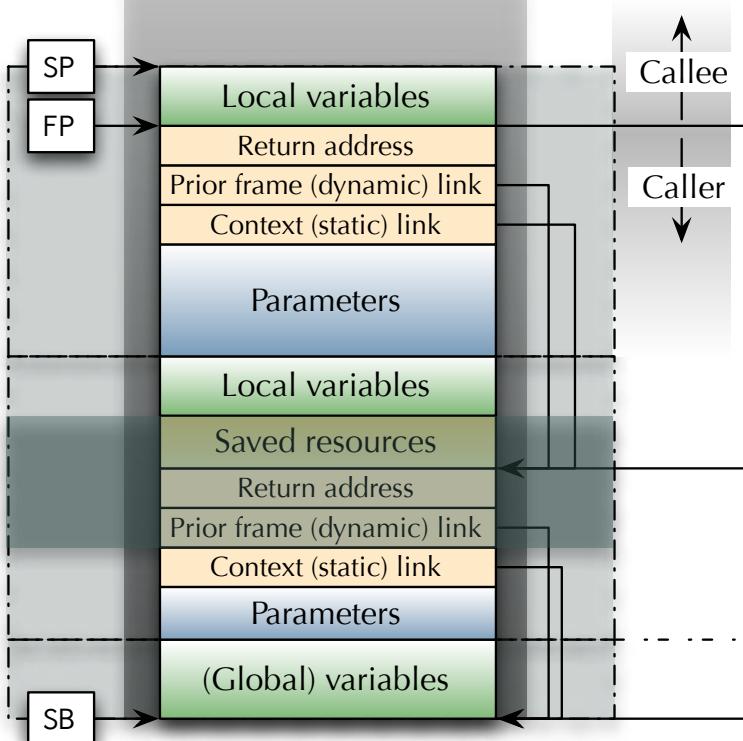
Functions

Generic Stack-Frame



The next function call will produce the next stack frame.

- ☞ Variables and parameters from the context stay visible (via the chain of static links).



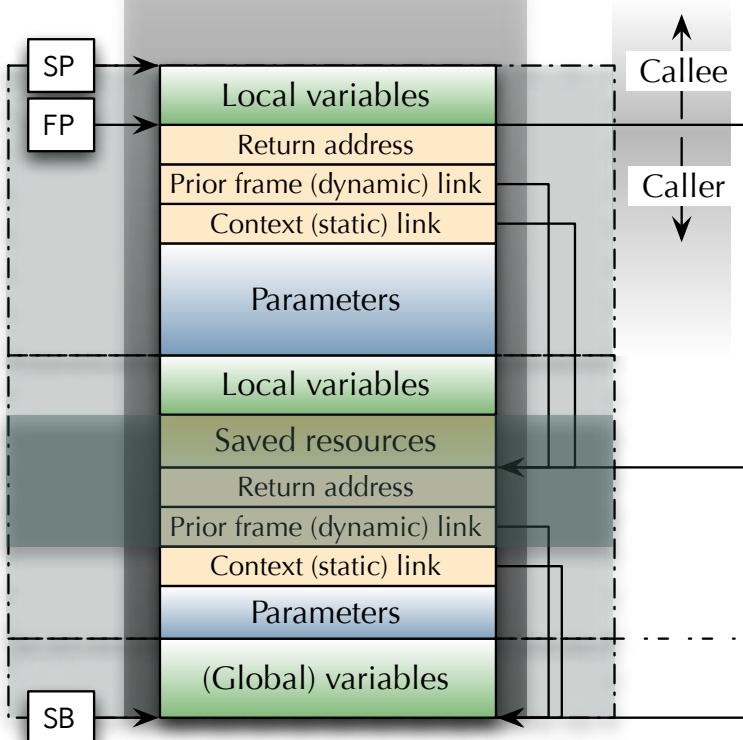


Functions

Generic Stack-Frame

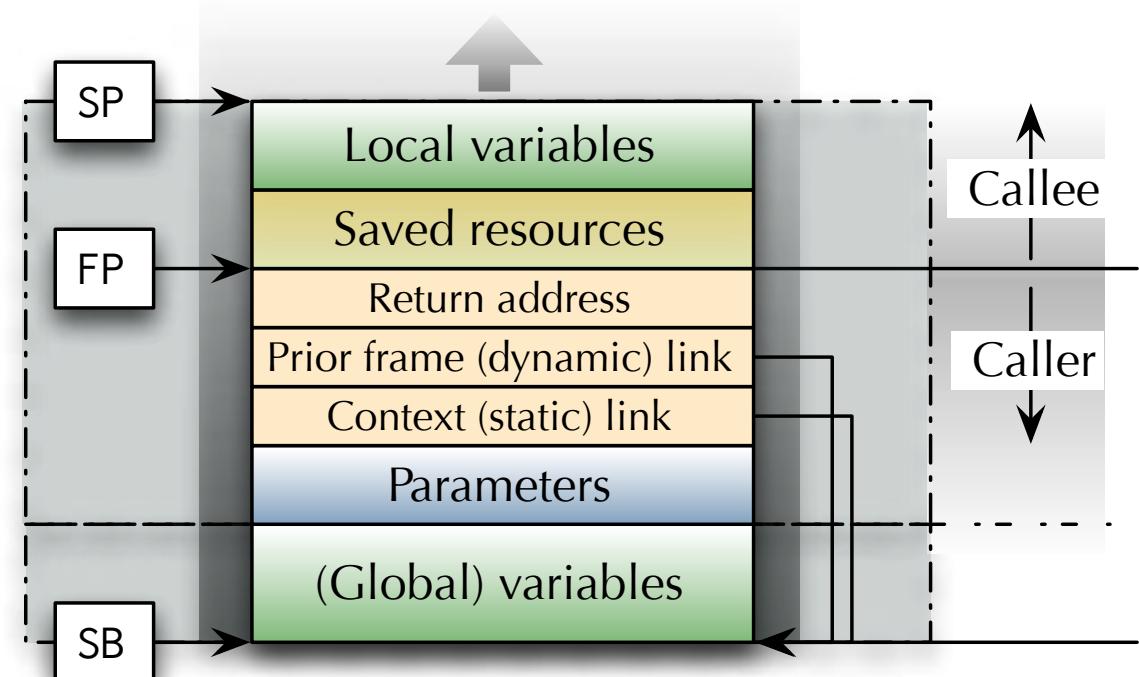


Local variables can only be added to the currently executing function.



The next function call will produce the next stack frame.

- ☞ Variables and parameters from the context stay visible (via the chain of static links).



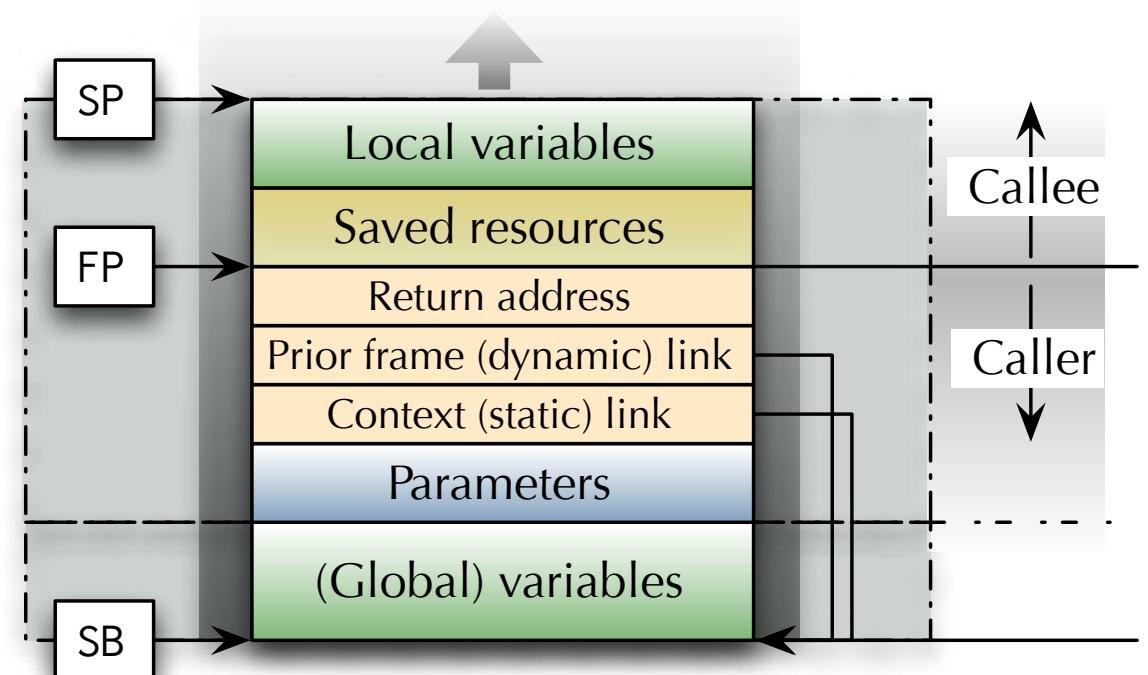
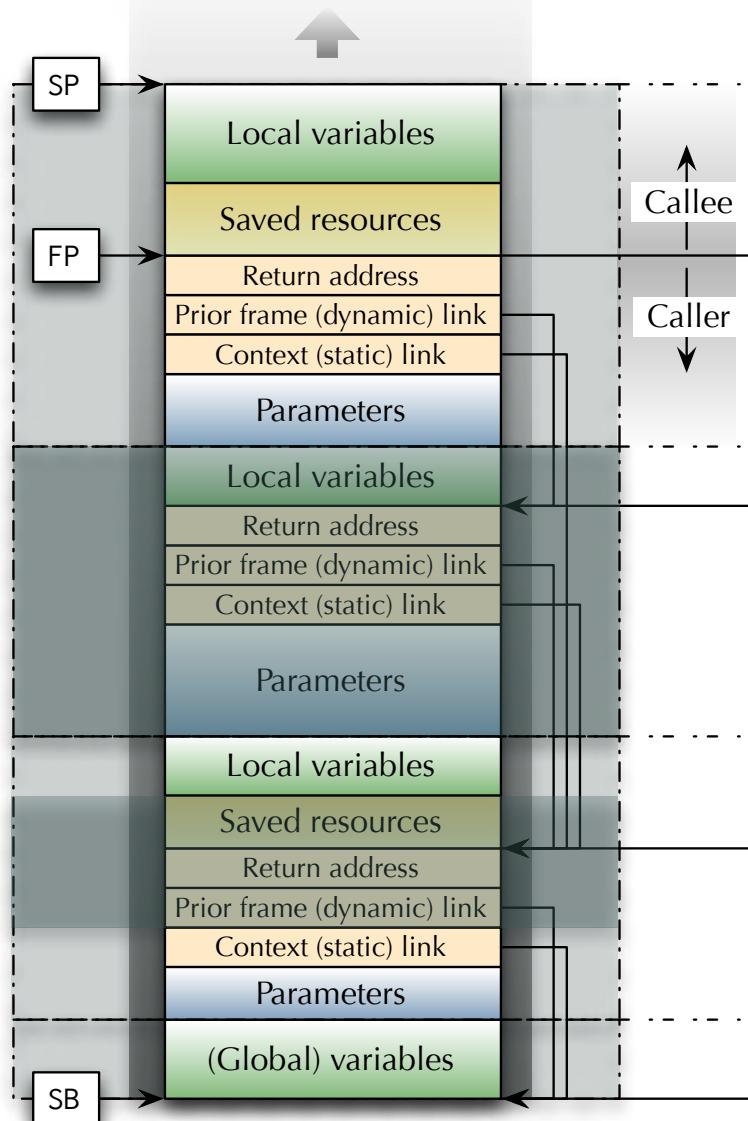


Functions

Generic Stack-Frame

The next function call will produce the next stack frame.

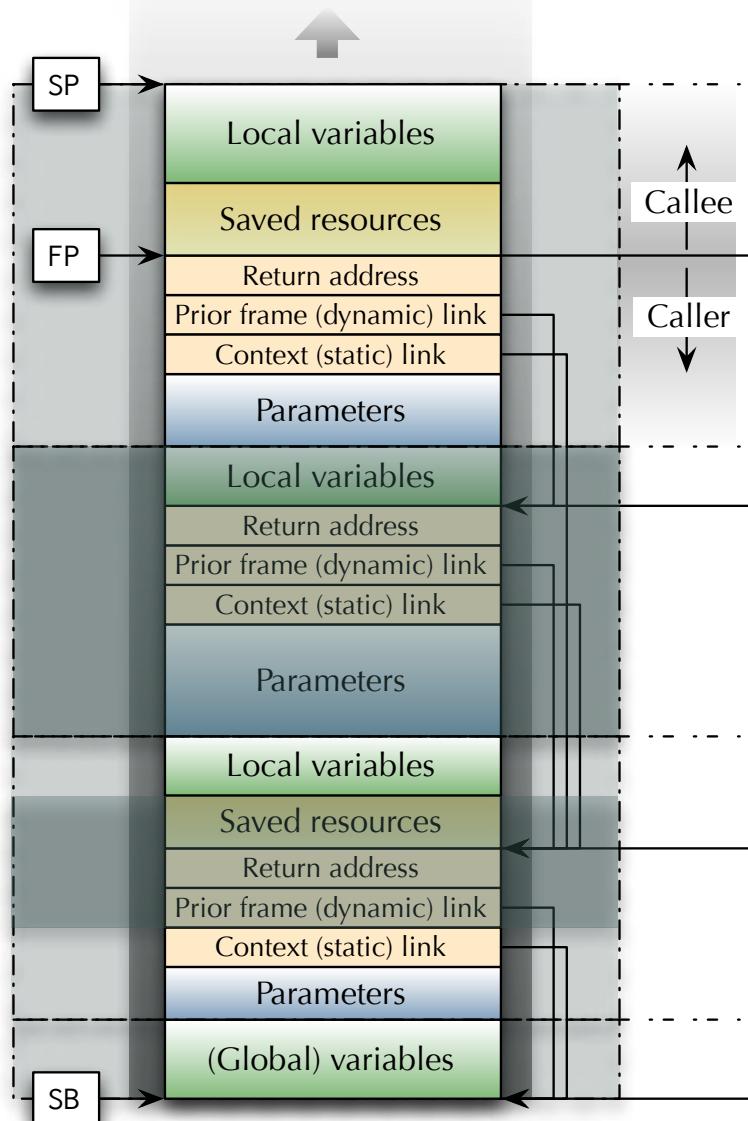
☞ Note which variables and parameters are visible.





Functions

Generic Stack-Frame

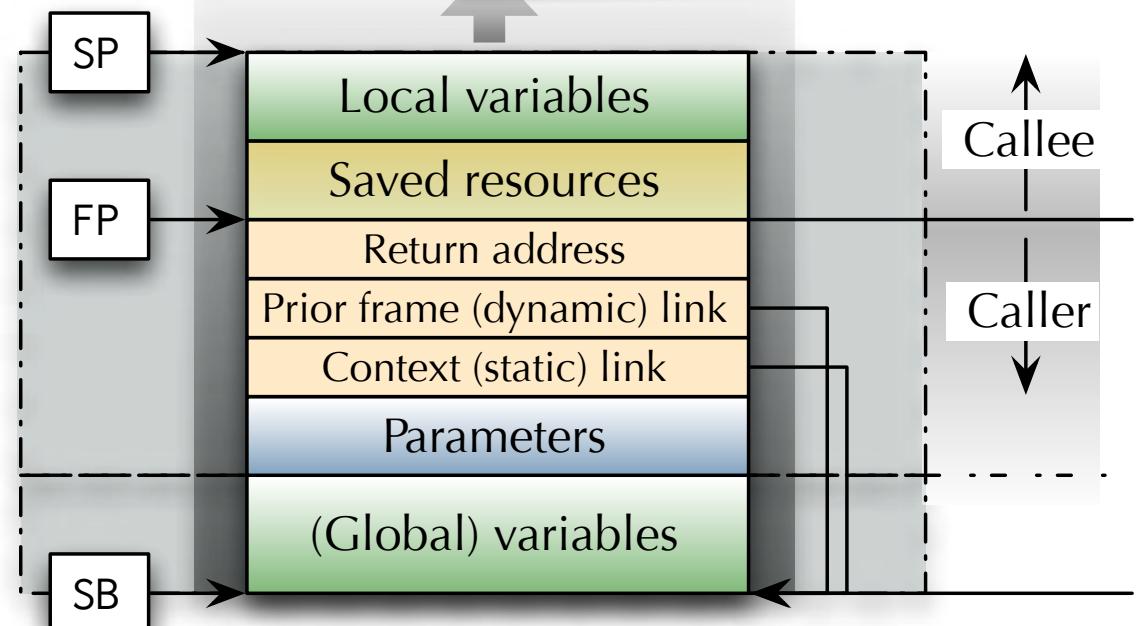


The next function call will produce the next stack frame.

☞ Note which variables and parameters are visible.

Accessing the context like
that can be inefficient!

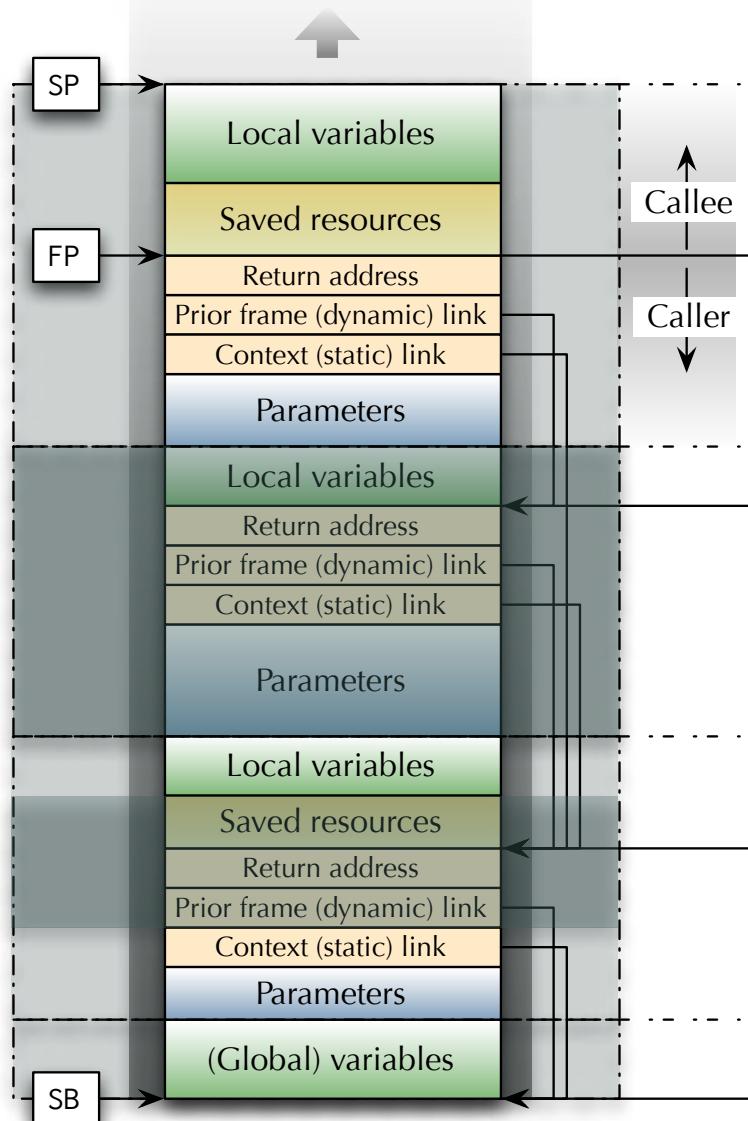
☞ Compilers may choose other
mechanism (e.g. displays,
which make all context
levels accessible at once).



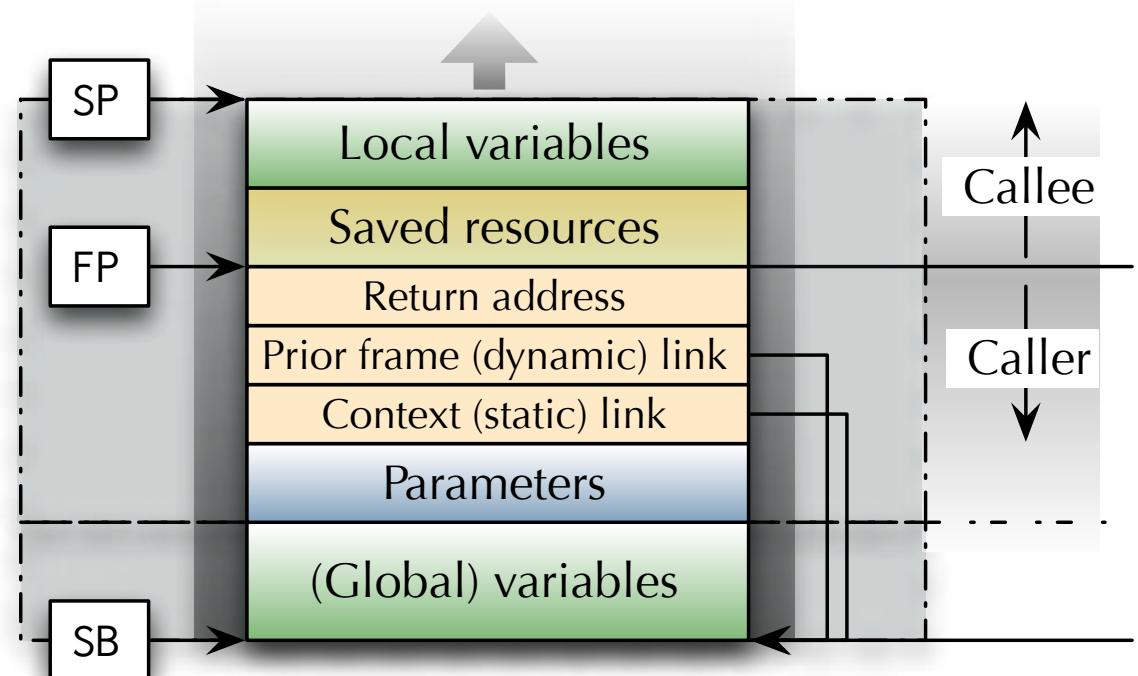


Functions

Generic Stack-Frame



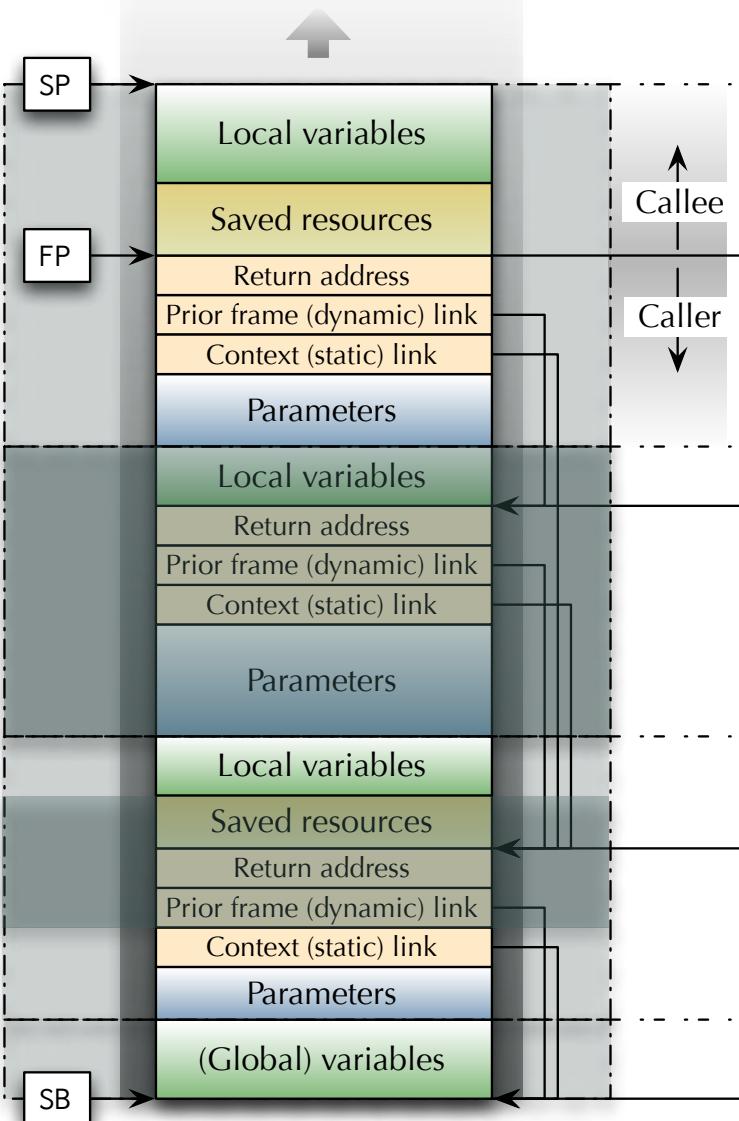
☞ How fast / complex is the **allocation and deallocation** of *local variables* and *parameters* on the stack?





Functions

Generic Stack-Frame – Caller



Pre_Call:

- ... ; Allocate/identify space for the parameters
- ... ; Copy the in and in-out
- ... ; parameters to this space
- ... ; Potentially provide links
- ... ; Provide a return address ("Post_Call")
- ... ; (usually implicit with the call itself)

☞ Call the function

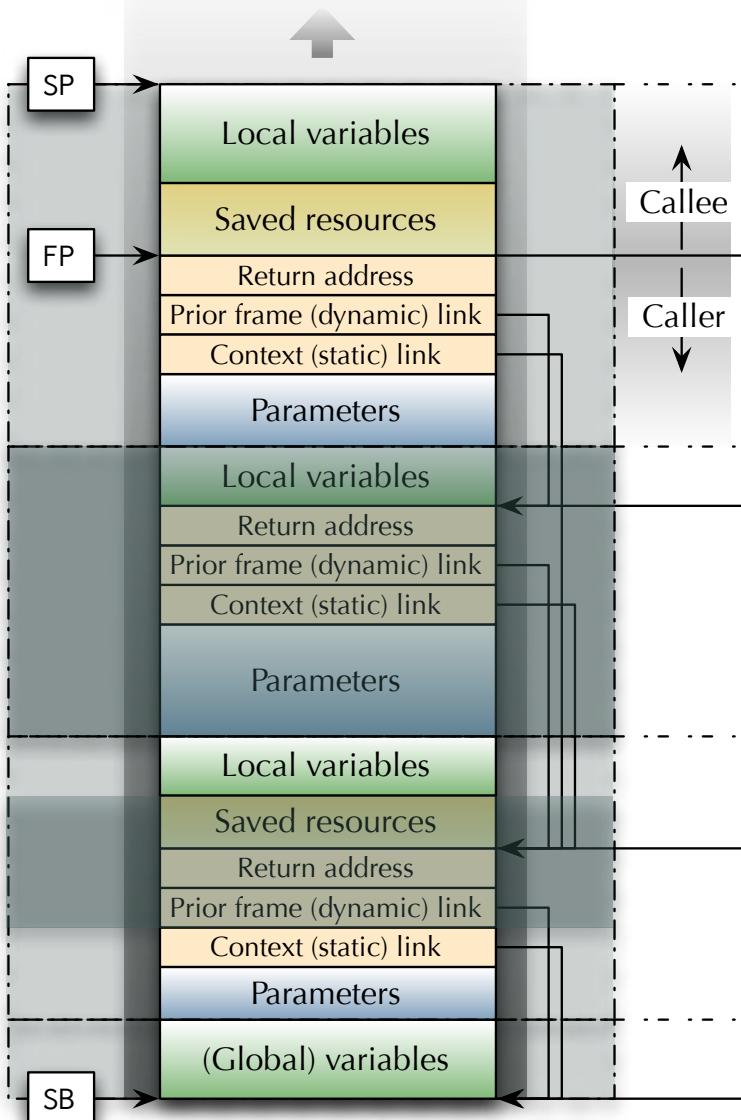
Post_Call:

- ... ; Copy the out and in-out parameters
- ... ; to local variables or registers
- ... ; Potentially restore the frame pointer
- ... ; Restore the stack to its previous state
- ... ; (if the stack has been used)



Functions

Generic Stack-Frame – Callee



Prologue:

... ; Save all registers which are needed
... ; inside this function to the stack
... ; Establish a new frame pointer
... ; while potentially saving the previous fp
... ; Allocate/identify space for local variables
... ; Potentially initialize local variables

👉 Operations, which will use
local and context variables and parameters

(via the FP(s))

Epilogue:

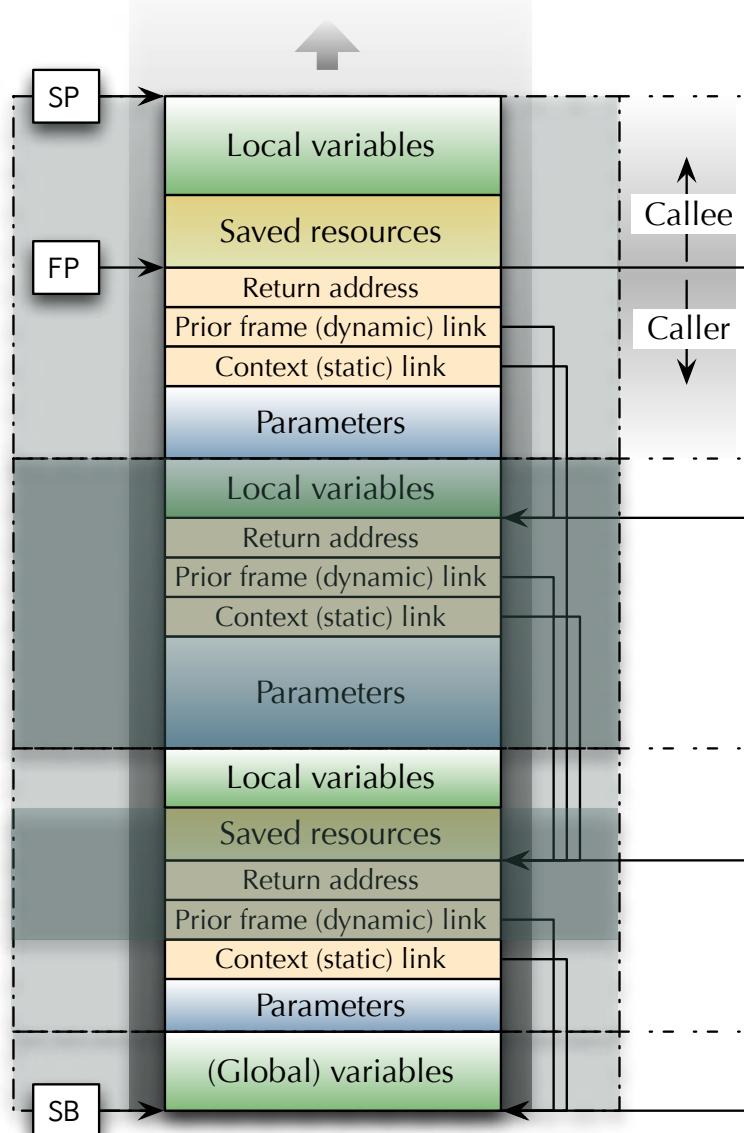
... ; Potentially restore the prior frame pointer
... ; Restore the stack to its state at entry

👉 Return from function



Functions

Generic Stack-Frame – Heap

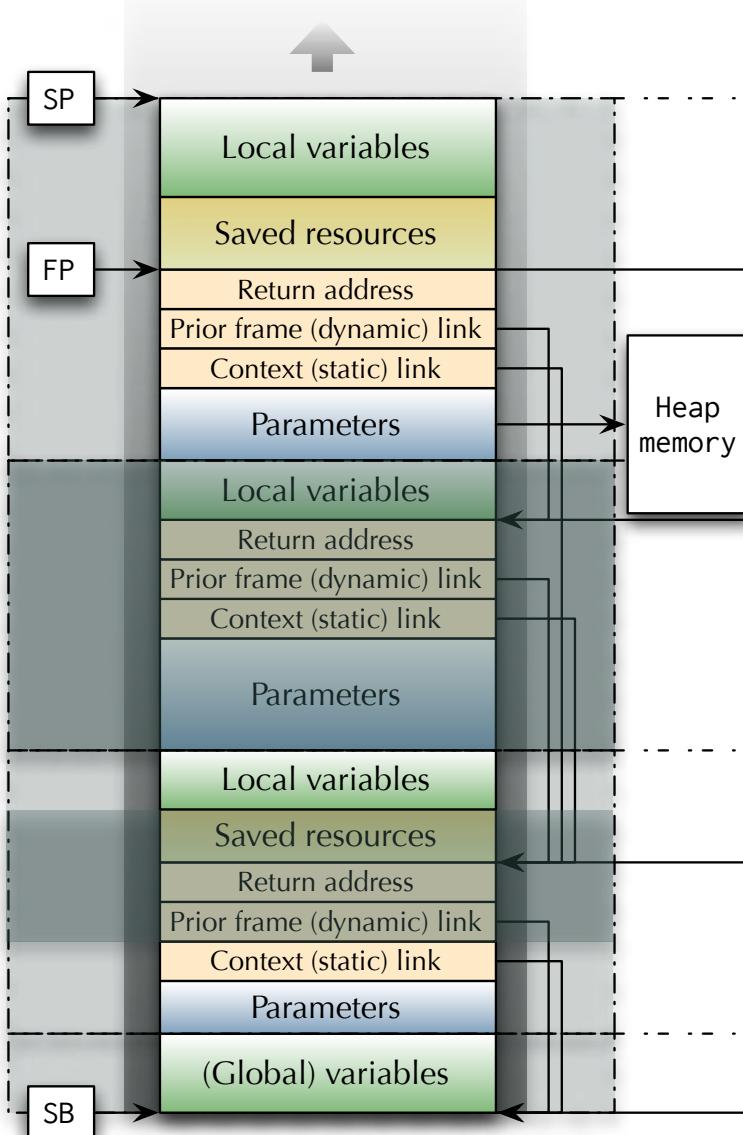


☞ How to keep any memory allocation after function return?



Functions

Generic Stack-Frame – Heap



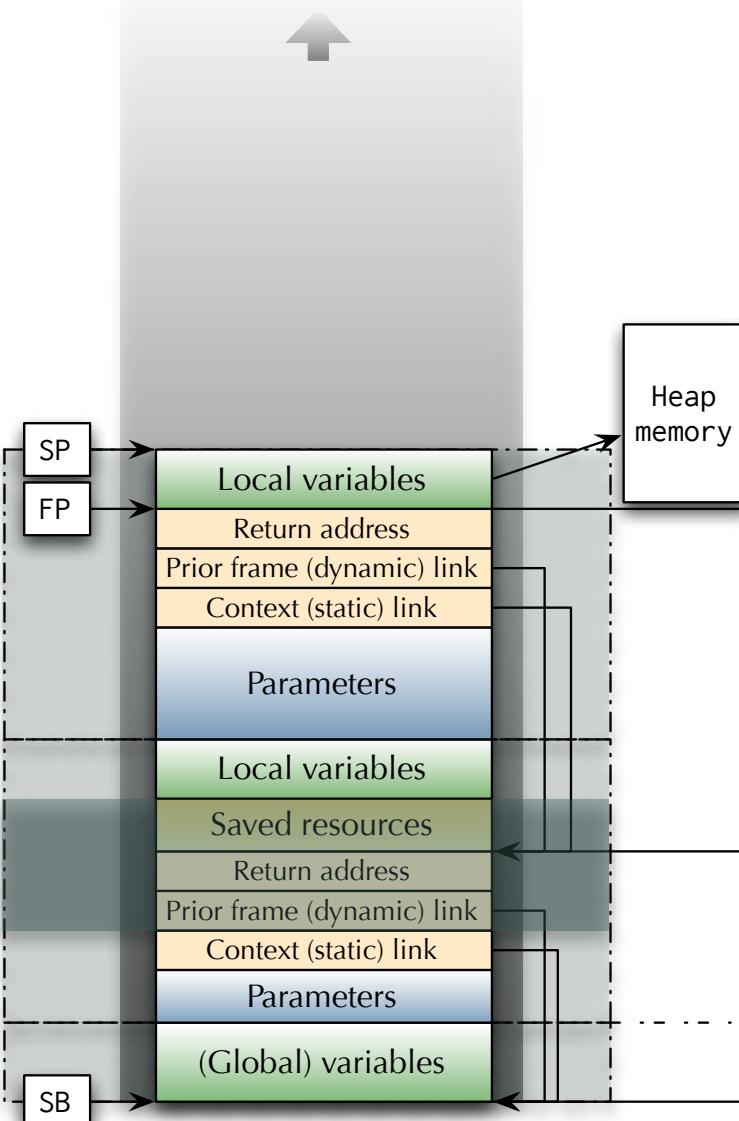
☞ How to keep any memory allocation after function return?

☞ By using an out, by-reference parameter, the link to the newly allocated memory area is kept.



Functions

Generic Stack-Frame – Heap



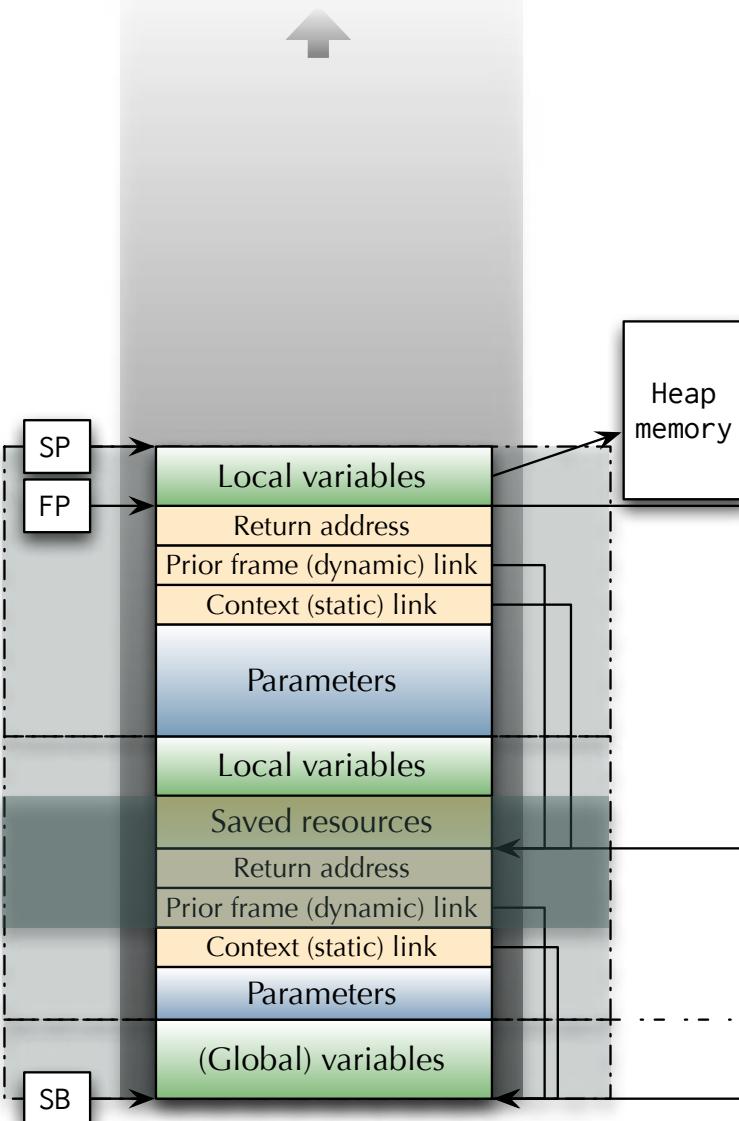
☞ How to keep any memory allocation after function return?

- ☞ By using an out, by-reference parameter, the link to the newly allocated memory area is kept.
- ☞ ... and a local variable in the calling function can keep this link.



Functions

Generic Stack-Frame – Heap



☞ How to keep any memory allocation after function return?

☞ By using an out, by-reference parameter, the link to the newly allocated memory area is kept.

☞ ... and a local variable in the calling function can keep this link.

☞ When to deallocate though?

- Garbage collection (Java)?
- Smart pointers (C++)?
- Reference ownerships (Rust)?
- Scoped pointers / storage pools (Ada)?



Functions

Summary

Functions

- **Framework**

- Return address
- Relative addressing

- **Parameter passing modes and mechanisms**

- Copy versus reference
- Information flow directions
- Late evaluation

- **Stackframes**

- Static and dynamic links
- Parameters
- Local variables

Computer Organisation & Program Execution 2021



Data Structures

Uwe R. Zimmer - The Australian National University



Data Structures

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

Chapter 2 “Instructions: Language of the Computer”,

Chapter 5 “Large and Fast: Exploiting Memory Hierarchy”

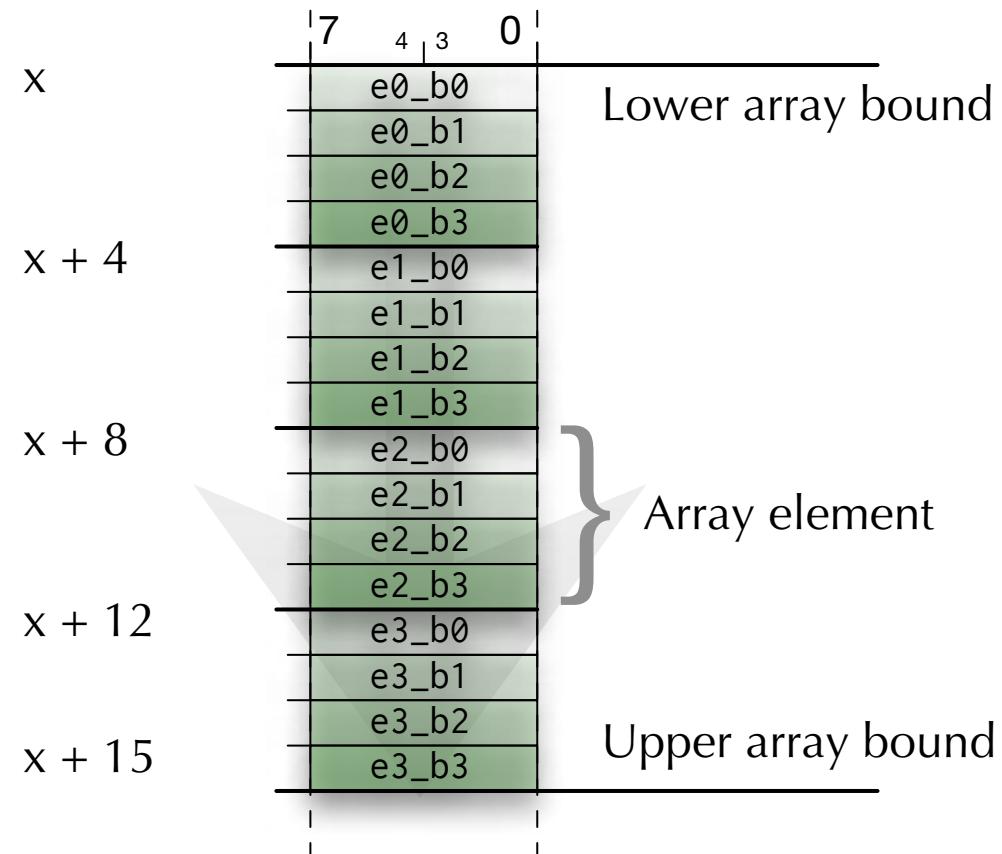
ARM edition, Morgan Kaufmann 2017



Data Structures

Array layout

Elements of equal size sequentially in memory





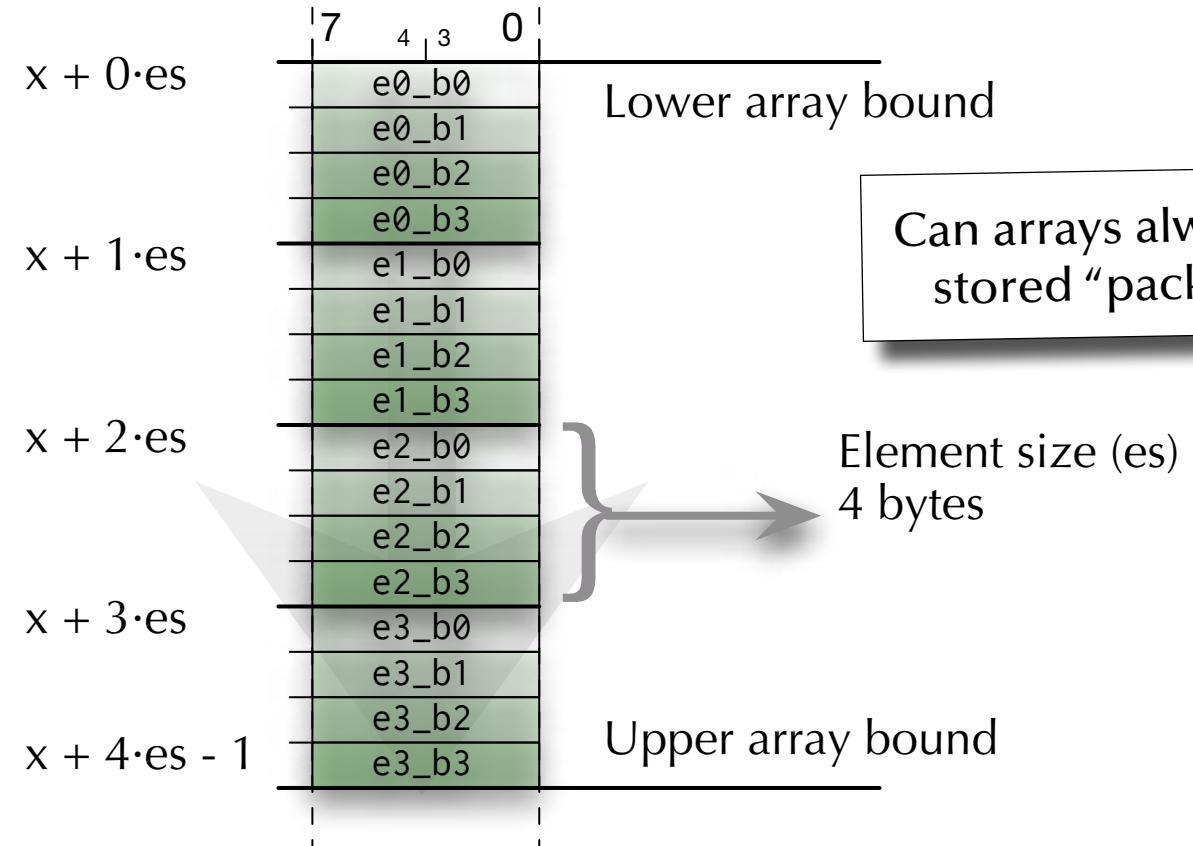
Data Structures

Array addressing

$x + i \cdot es$

Maybe be good to have $es = 2^n$?

What happens if array bounds are violated?

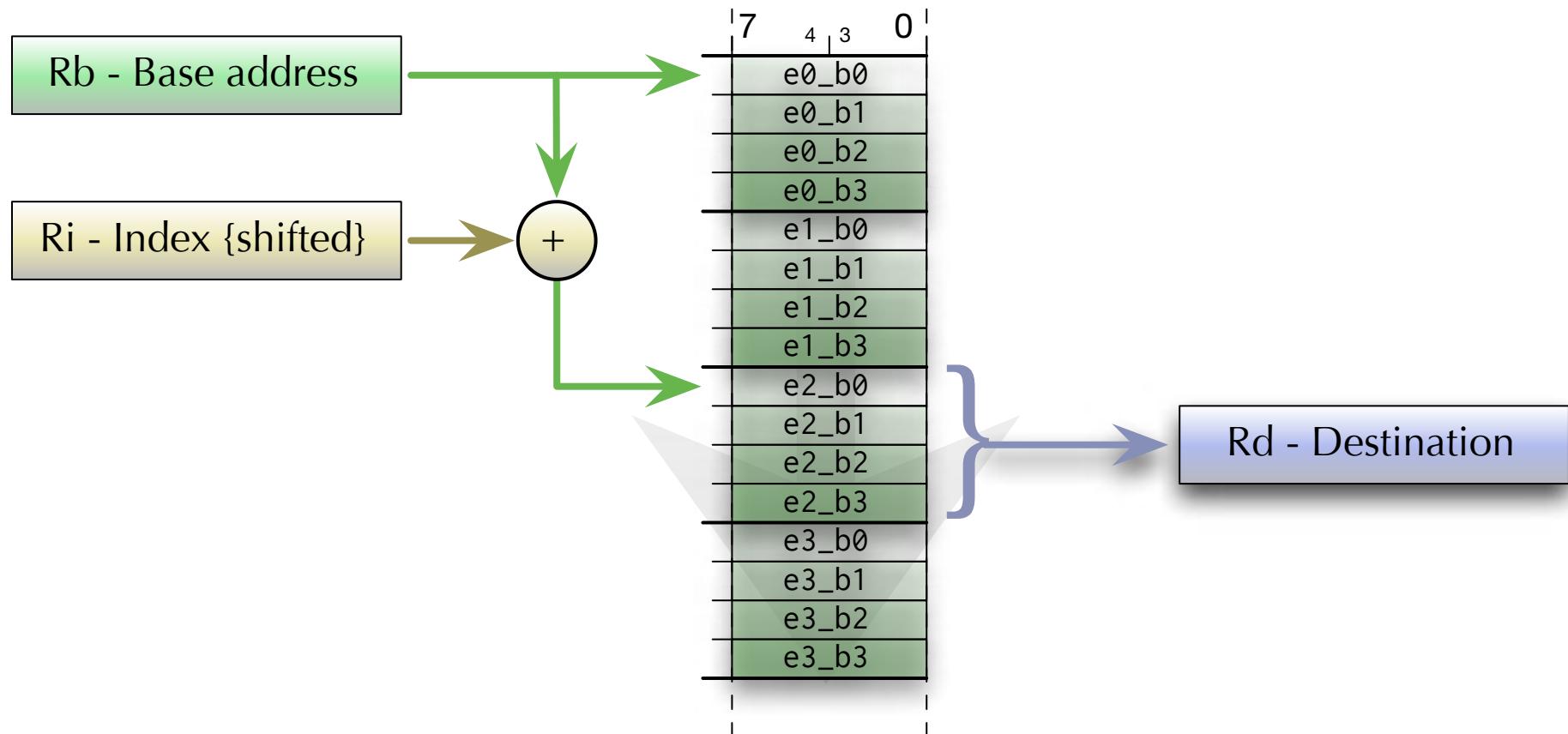


☞ And who is checking that?



Data Structures

Array addressing via index register

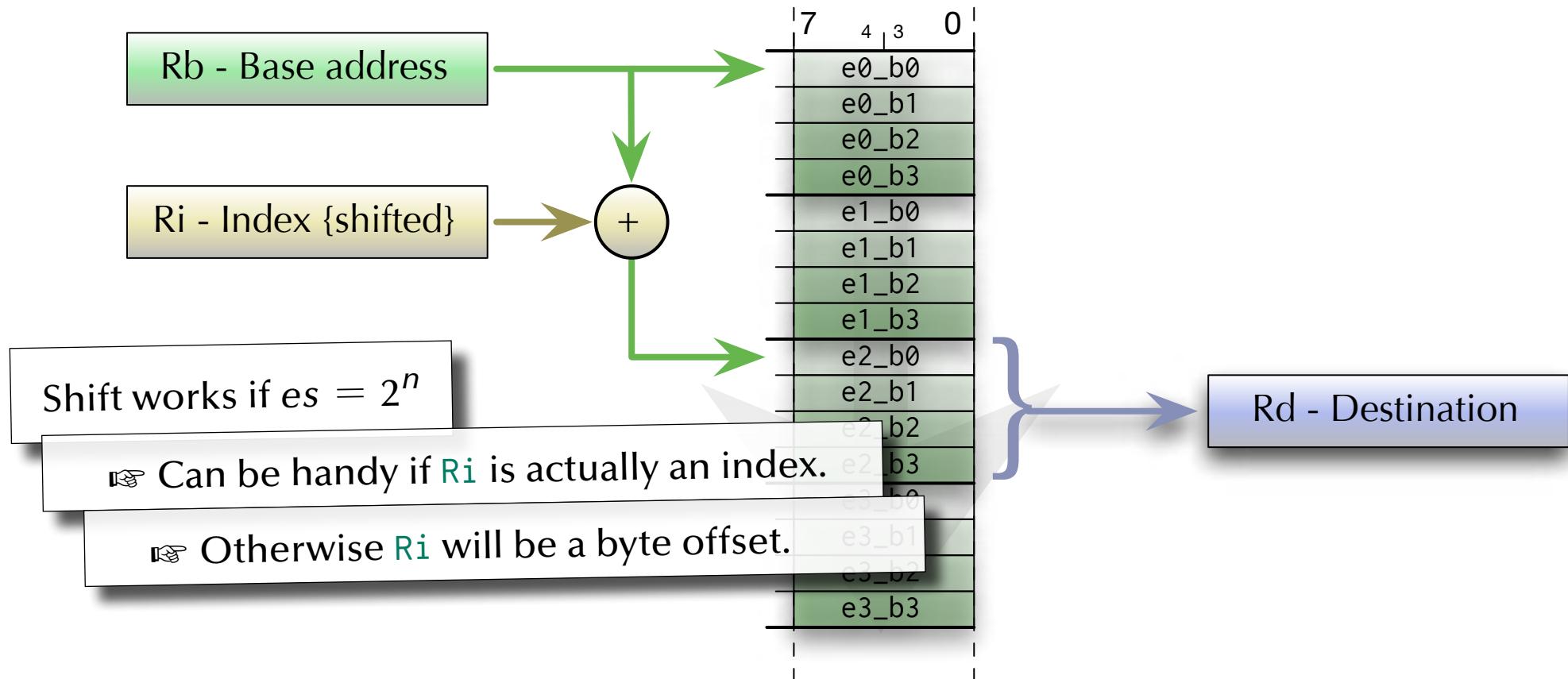


ldr<c><q> <Rd>, [<Rb>, <Ri> {, lsl #<shift>}]



Data Structures

Array addressing via index register

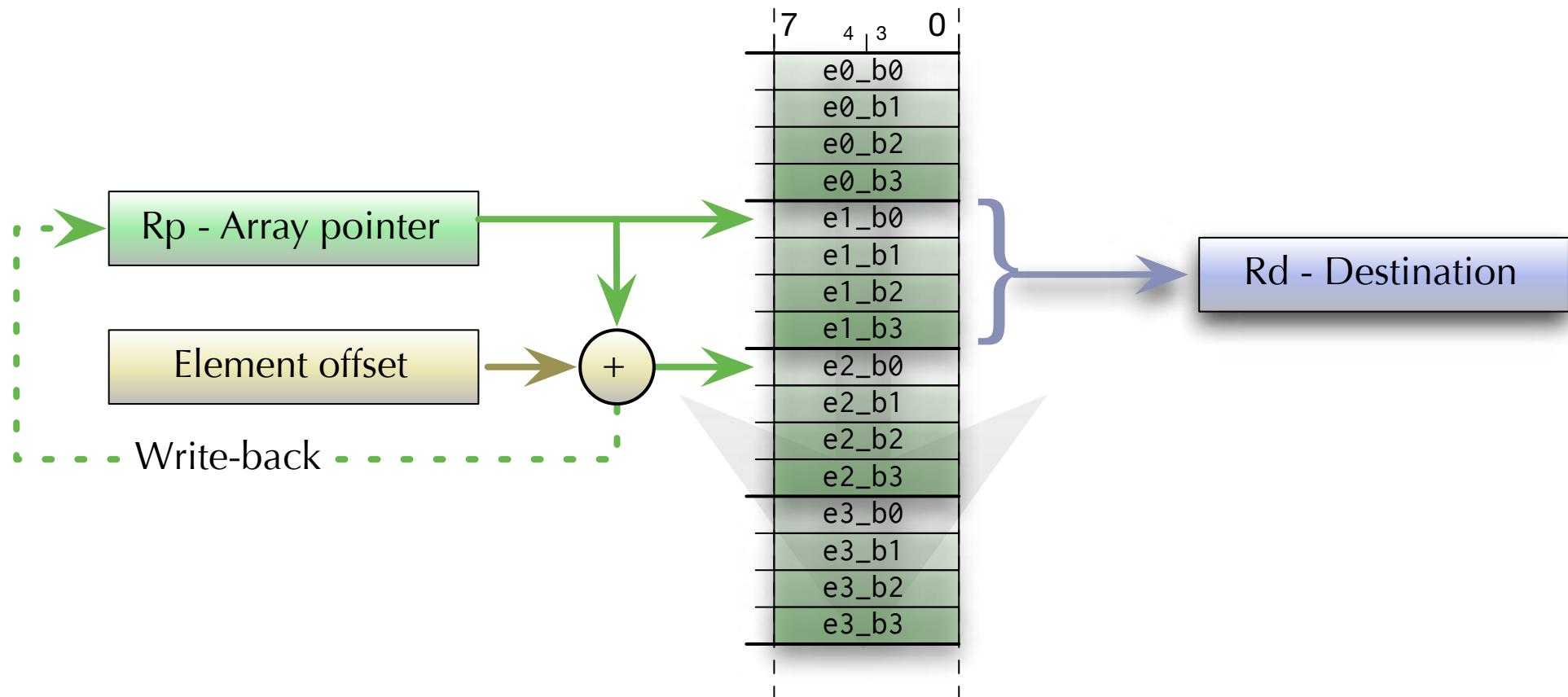


ldr<c><q> <Rd>, [<Rb>, <Ri> {, lsl #<shift>}]



Data Structures

Array addressing via element pointer

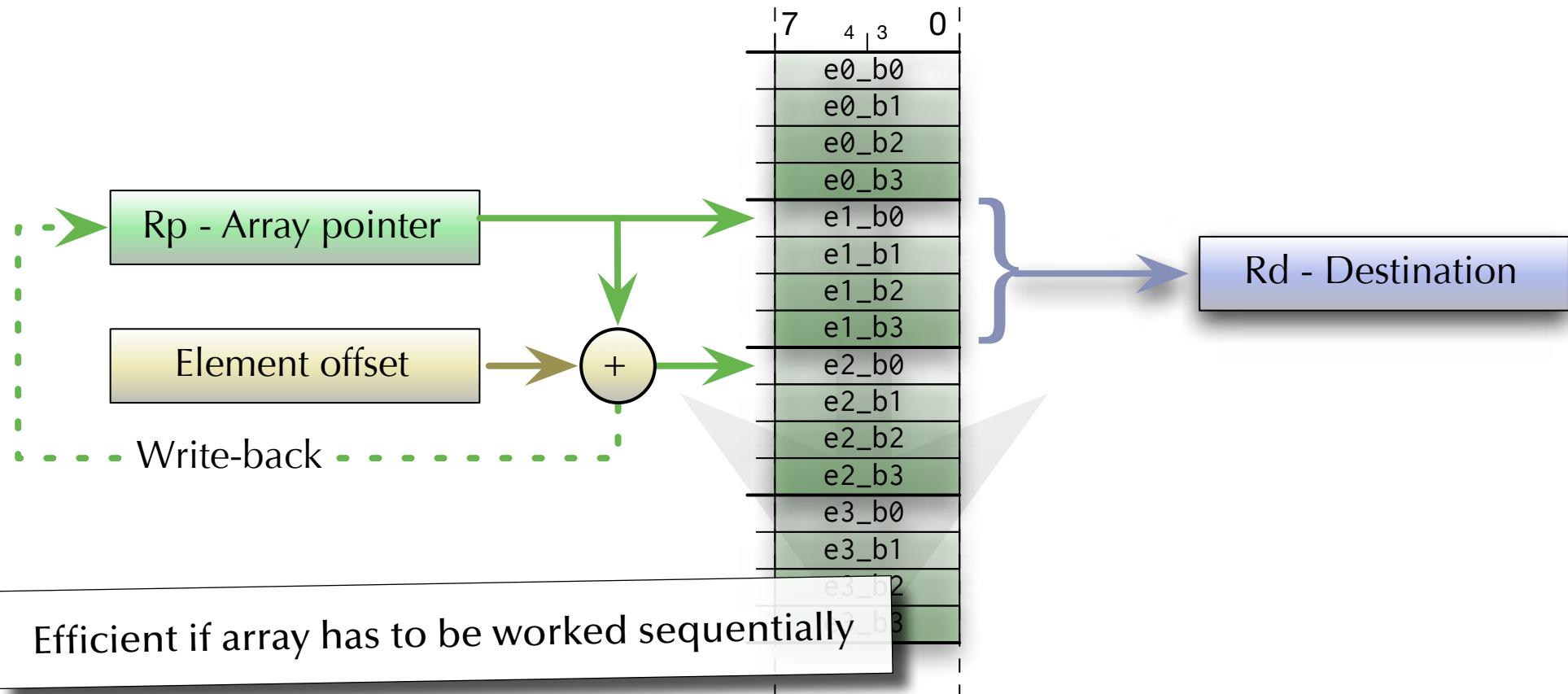


ldr<c><q> <Rd>, [<Rp>], #+/-<offset>



Data Structures

Array addressing via element pointer



ldr<c><q> <Rd>, [<Rp>], #+/-<offset>



Data Structures

Calculate $\sum x_i$

```
int sum (unsigned int uints [], unsigned int from, unsigned int to) {  
    int i;  
    int acc = 0;  
    for (i = from; i <= to; i++) {  
        acc += uints [i];  
    }  
    return acc;  
}
```

```
unsigned int uints [100];  
unsigned int s;  
int i;  
for (i = 0; i <= 99; i++) {  
    uints [i] = rand ();  
}  
s = sum (uints, 0, 99);
```

```
type Naturals is array (Integer range <>) of Natural;  
function Sum (Numbers : Naturals) return Natural is  
    Acc : Natural := 0;  
  
begin  
    for n of Numbers loop  
        Acc := Acc + n;  
    end loop;  
    return Acc;  
end Sum;
```

```
Numbers : constant Naturals (1 .. 100) :=  
    (others => Random (Numbers_Generator));  
Sum_of_Numbers : constant Natural := Sum (Numbers);
```



Data Structures

Calculate $\sum x_i$

```
int sum (unsigned int uints [], unsigned int from, unsigned int to) {  
    int i;  
    int acc = 0;  
    for (i = from; i <= to; i++) {  
        acc += uints [i];  
    }  
    return acc;  
}
```

Best of luck with the array bounds!

```
unsigned int uints [100];  
unsigned int s;  
int i;  
for (i = 0; i <= 99; i++) {  
    uints [i] = rand ();  
}  
s = sum (uints, 0, 99);
```

```
type Naturals is array (Integer range <>) of Natural;  
function Sum (Numbers : Naturals) return Natural is  
    Acc : Natural := 0;  
  
begin  
    for n of Numbers loop  
        Acc := Acc + n;  
    end loop;  
    return Acc;  
end Sum;
```

```
Numbers : constant Naturals (1 .. 100) :=  
(others => Random (Numbers_Generator));  
Sum_of_Numbers : constant Natural := Sum (Numbers);
```



Data Structures

Arbitrary array indexing

```
; r0 base address for array a  
; r1 from array index  
; r2 to array index  
  
mov  r3, #0          ; sum := 0  
mov  r4, #4          ; element size is 4 bytes  
mov  r5, #-1         ; first_element_offset
```

3

for_sum:

```
cmp  r1, r2          ; i > to  
bgt end_for_sum  
mla  r6, r1, r4, r5  ; element_offset := (i * 4) - first_element_offset  
ldr  r7, [r0, r6]    ; a [i] := [base + element_offset]  
add  r3, r7          ; sum := sum + a [i]  
add  r1, #1          ; i := i + 1  
b    for_sum
```

9

end_for_sum:

```
mov  r0, r3          ; r0 sum over all a [from .. to]
```

1



Data Structures

Zero-based array indexing

; r0 base address for array a
; r1 from array index
; r2 to array index

```
mov  r3, #0          ; sum := 0
mov  r4, #4          ; element size is 4 bytes
```

2

for_sum:

```
cmp  r1, r2          ; i > to
bgt end_for_sum
mul r5, r1, r4        ; element_offset := (i * 4)
ldr  r6, [r0, r5]      ; a [i] := [base + element_offset]
add r3, r6            ; sum := sum + a [i]
add r1, #1            ; i := i + 1
b    for_sum
```

8

end_for_sum:

```
mov  r0, r3          ; r0 sum over all a [from .. to]
```

1



Data Structures

Replacing multiplication with shifted index register

; r0 base address for array a
; r1 from array index
; r2 to array index

```
mov r3, #0 ; sum := 0
```

1

for_sum:

```
cmp r1, r2 ; i > to
bgt end_for_sum
ldr r4, [r0, r1, lsl #2] ; a [i] := [base + element_offset]
add r3, r4 ; sum := sum + a [i]
add r1, #1 ; i := i + 1
b for_sum
```

7

end_for_sum:

```
mov r0, r3 ; r0 sum over all a [from .. to]
```

1



Data Structures

Replacing indices with offsets

```
; r0 base address for array a  
; r1 from array index  
; r2 to array index  
lsl    r1, r1, #2          ; translate from index to offset  
lsl    r2, r2, #2          ; translate to index to offset  
mov    r3, #0              ; sum := 0
```

3

for_sum:

```
cmp    r1, r2            ; i > to  
bgt   end_for_sum  
ldr    r4, [r0, r1]        ; a [i] := [base + offset]  
add    r3, r4            ; sum := sum + a [i]  
add    r1, #4             ; offset := offset + 4  
b      for_sum
```

7

end_for_sum:

```
mov    r0, r3            ; r0 sum over all a [from .. to]
```

1



Data Structures

Assuming non-empty arrays

; r0 base address for array a
; r1 from array index
; r2 to array index \geq from index

```
lsl    r1, r1, #2          ; translate from index to offset
lsl    r2, r2, #2          ; translate to index to offset
mov    r3, #0              ; sum := 0
```

3

for_sum:

```
ldr    r4, [r0, r1]        ; a [i] := [base + offset]
add    r3, r4              ; sum := sum + a [i]
add    r1, #4              ; offset := offset + 4
cmp    r1, r2              ; i <= to
ble    for_sum
```

6

end_for_sum:

```
mov    r0, r3              ; r0 sum over all a [from .. to]
```

1



Data Structures

Replacing offsets with addresses

```
; r0 base address for array a  
; r1 from array index  
; r2 to array index >= from index  
  
lsl    r1, r1, #2          ; translate from index to offset  
lsl    r2, r2, #2          ; translate to index to offset  
add   r1, r0              ; translate from index to address -> i_addr  
add   r2, r0              ; translate to index to address -> to_addr  
mov   r0, #0              ; sum := 0
```

5

for_sum:

```
ldr    r3, [r1], #4        ; a [i] := [i_addr]; i_addr += 4  
add   r0, r3              ; sum := sum + a [i]  
cmp   r1, r2              ; i_addr <= to_addr  
ble   for_sum
```

5

end_for_sum:

```
; r0 sum over all a [from .. to]
```



Data Structures

Array Slices

```
numbers      = [0, 1, 2, 3, 4, 5]
numbersSlice = numbers [1:3]
# numbersSlice equals [1, 2, 3]

numbers      := []int {0, 1, 2, 3, 4, 5}
numbersSlice := numbers [1:3]
```

```
type Naturals is array (Integer range <>) of Natural;
Numbers       : constant Naturals (-50 .. 50) := (others => Random (Generator));
Numbers_Slice_1 : constant Naturals           := Numbers (-10 .. 10);
Numbers_Slice_2 : constant Naturals           := Numbers ( 1 .. 10);
Numbers_Slice_3 :           Naturals          := Numbers (-20 .. 50);

begin
  for n of Numbers_Slice_3 loop
    n := n + 1;
  end loop;
end;
```



Data Structures

Array Slices

```
numbers      = [0, 1, 2, 3, 4, 5]
numbersSlice = numbers [1:3]
# numbersSlice equals [1, 2, 3]

numbers      := []int {0, 1, 2, 3, 4, 5}
numbersSlice := numbers [1:3]
```

Are those copy or reference affairs?

```
type Naturals is array (Integer range <>) of Natural;
Numbers       : constant Naturals (-50 .. 50) := (others => Random (Generator));
Numbers_Slice_1 : constant Naturals           := Numbers (-10 .. 10);
Numbers_Slice_2 : constant Naturals          ( 1 .. 10) := Numbers ( 11 .. 20);
Numbers_Slice_3 :               Naturals        := Numbers (-20 .. 50);

begin
  for n of Numbers_Slice_3 loop
    n := n + 1;
  end loop;
end;
```



Data Structures

Copy array slice

```
; r0 base address for array a  
; r1 from array index  
; r2 to array index >= from index  
; r3 base address for array b  
  
lsl    r1, r1, #2          ; translate from index to offset  
lsl    r2, r2, #2          ; translate to index to offset  
add    r1, r0              ; translate from index to address -> i_addr  
add    r2, r0              ; translate to index to address -> to_addr
```

4

for_copy:

```
ldr    r4, [r1], #4        ; a [i] := [i_addr]; i_addr += 4  
str    r4, [r3], #4        ; [j_addr] := a [i] ; j_addr += 4  
cmp    r1, r2              ; i_addr <= to_addr  
ble    for_copy
```

6

end_for_copy:

```
; b [] := a [from .. to]
```



Data Structures

Copy array slice

```
; r0 base address for array a  
; r1 from array index  
; r2 to array index >= from index  
; r3 base address for array b  
  
lsl    r1, r1, #2          ; translate from index to offset  
lsl    r2, r2, #2          ; translate to index to offset  
add    r1, r0              ; translate from index to address -> i_addr  
add    r2, r0              ; translate to index to address -> to_addr
```

4

for_copy:

```
ldr    r4, [r1], #4        ; a [i] := [i_addr]; i_addr += 4  
str    r4, [r3], #4        ; [j_addr] := a [i] ; j_addr += 4  
cmp    r1, r2              ; i_addr <= to_addr  
ble    for_copy
```

6

end_for_copy:

```
; b [] := a [from .. to]
```

Moving blocks of memory can be done even/much faster with special hardware.

➡ DMA controllers



Data Structures

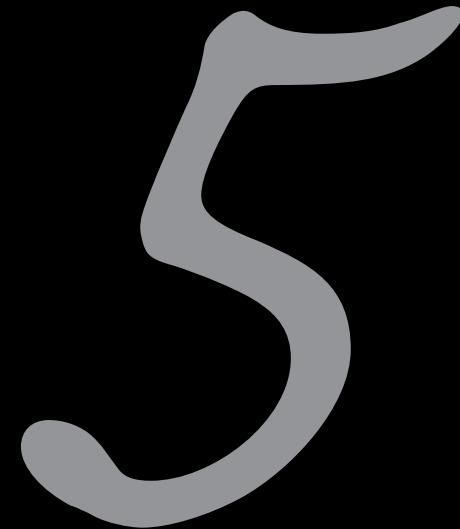
Summary

Data Structures

- **Arrays**

- Structure
- Alignment
- Addressing
- Iterators
- Copy procedures

Computer Organisation & Program Execution 2021



Asynchronism

Uwe R. Zimmer - The Australian National University



Asynchronism

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

Chapter 4 “The Processor”,

Chapter 6 “Parallel Processors from Client to Cloud”

ARM edition, Morgan Kaufmann 2017



Asynchronism

Why?

How do you handle your communication flow?



Asynchronism

Why?

How do you handle your communication flow?

- ☞ Do you have times when you check certain communication?
- ☞ Is certain communication interrupting you? – at any time?
 - ☞ Do you assign “importance levels” to your communication channels/sources?

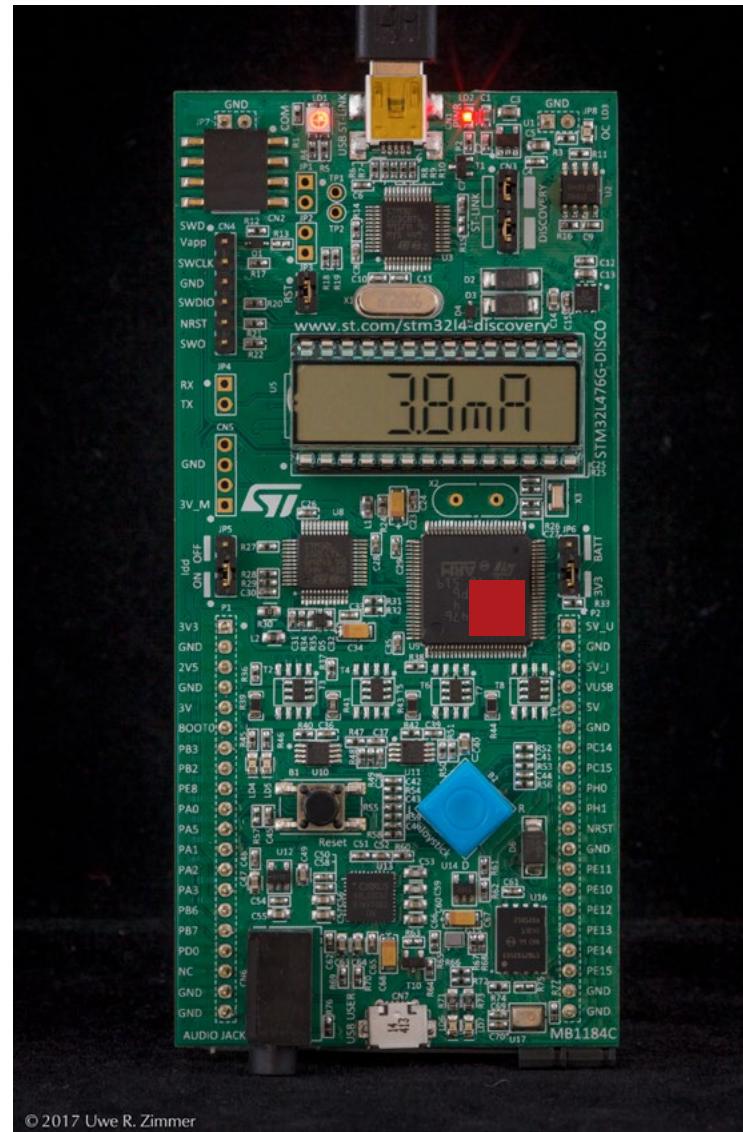


Asynchronism

STM32L476 Discovery

CPU

... running its sequence
of machine instructions.





Asynchronism

STM32L476 Discovery

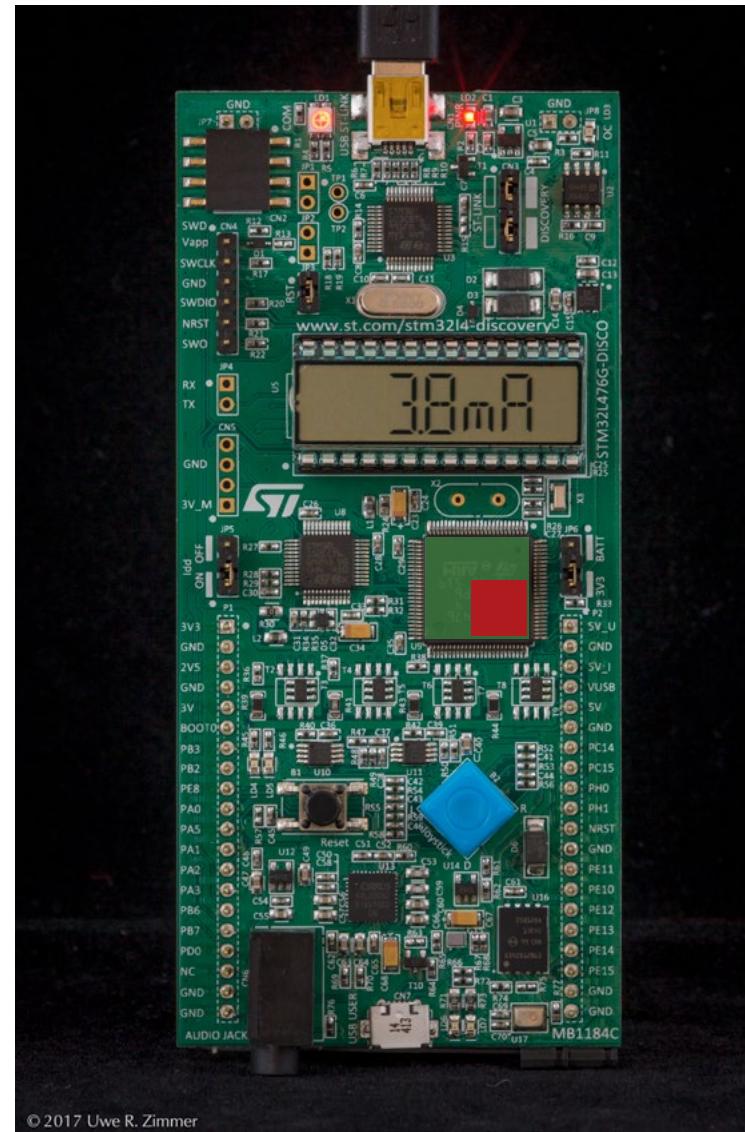
CPU

... running its sequence
of machine instructions.

- ☛ How to interact with
all the other devices
inside the

MCU

?

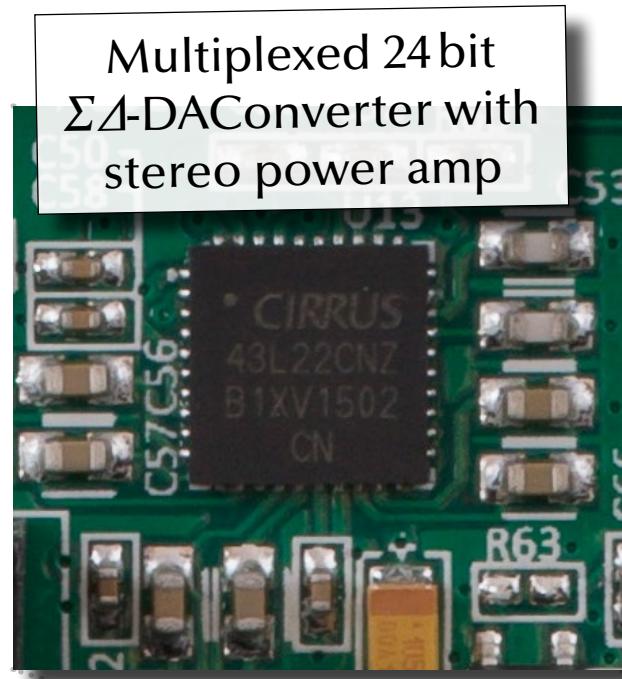


© 2017 Uwe R. Zimmer

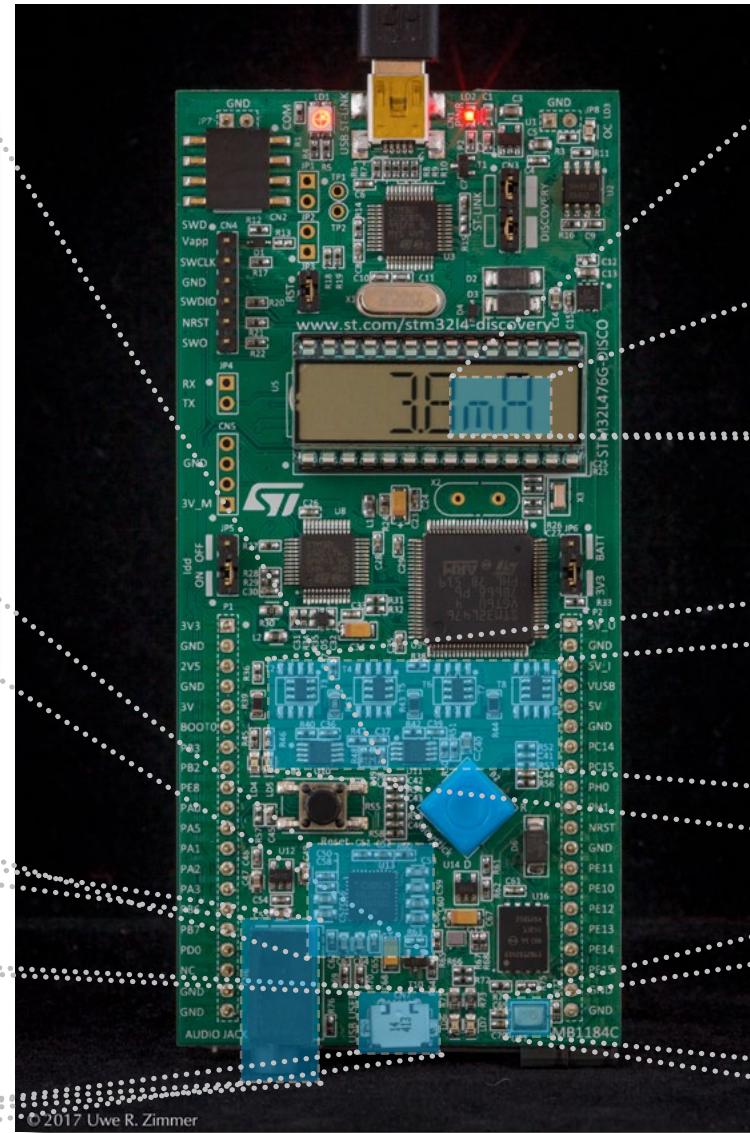


Asynchronism

STM32L476 Discovery



Multiplexed 24 bit
 $\Sigma\Delta$ -DAConverter with
stereo power amp



Headphone jack



USB OTG

“9 axis” motion sensor
(underneath display):
3 axis accelerometer
3 axis gyroscope
3 axis magnetometer

Current meter to MCU
60 nA ... 50 mA

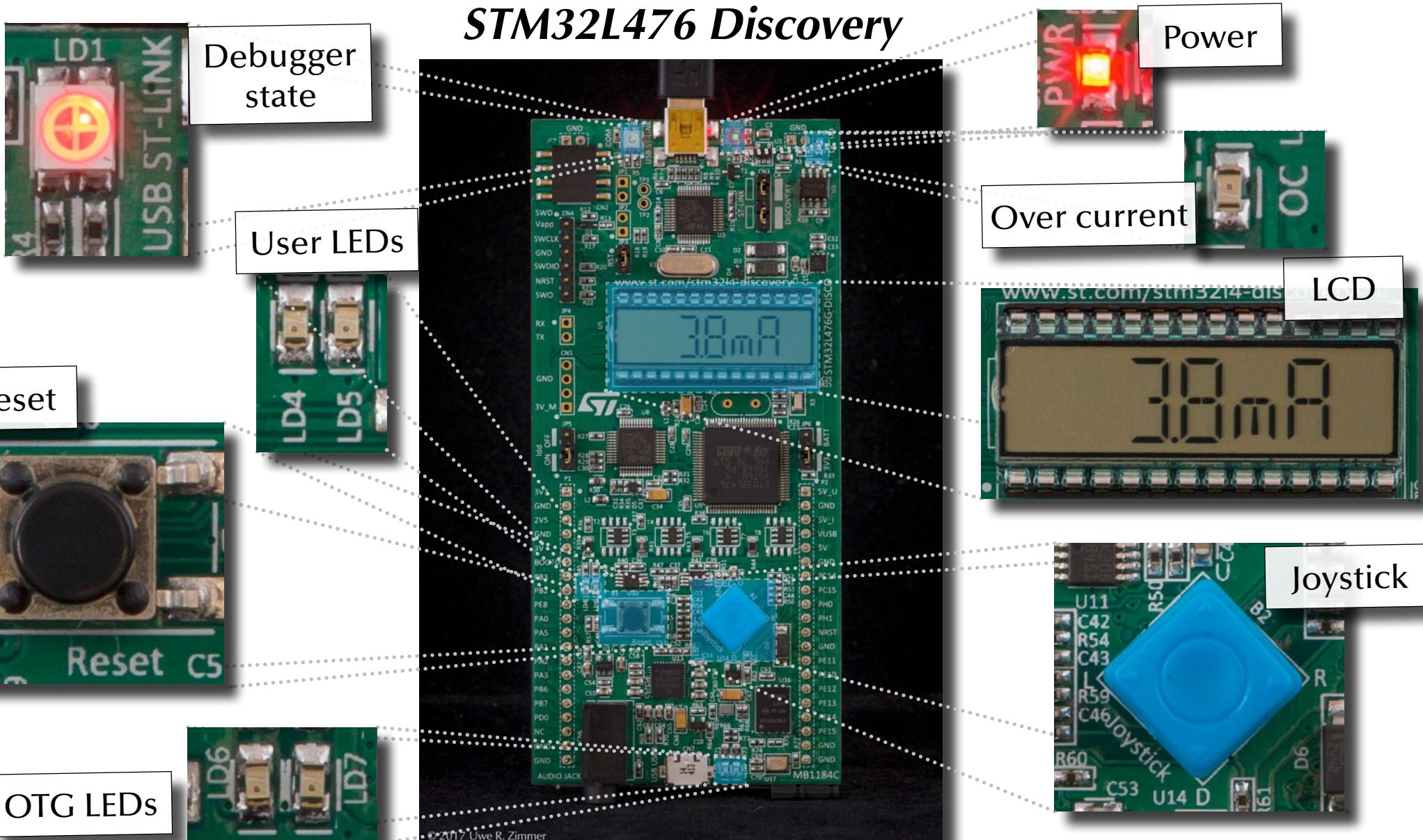


Microphone





Asynchronism





Asynchronism

STM32L476 Discovery

CPU

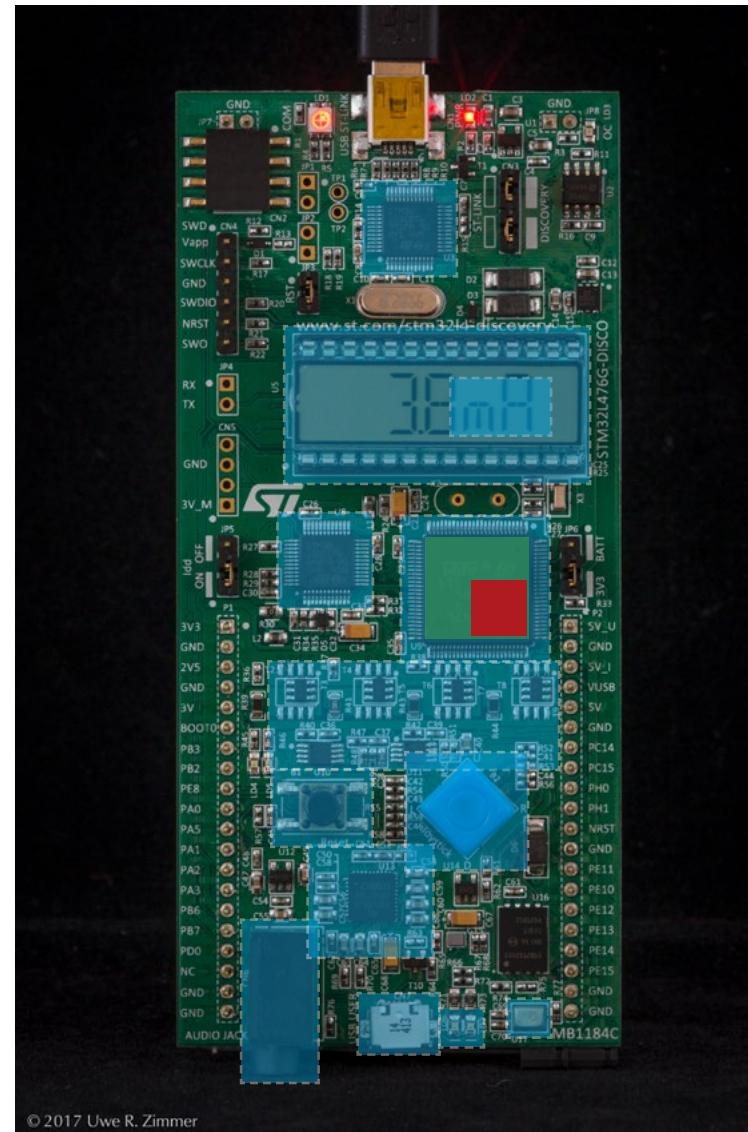
... running its sequence
of machine instructions.

- ☞ How to interact with
all the other devices
inside the

MCU

?

- ☞ ... and then with all the
devices on the board?



© 2017 Uwe R. Zimmer



Asynchronism

STM32L476 Discovery

CPU

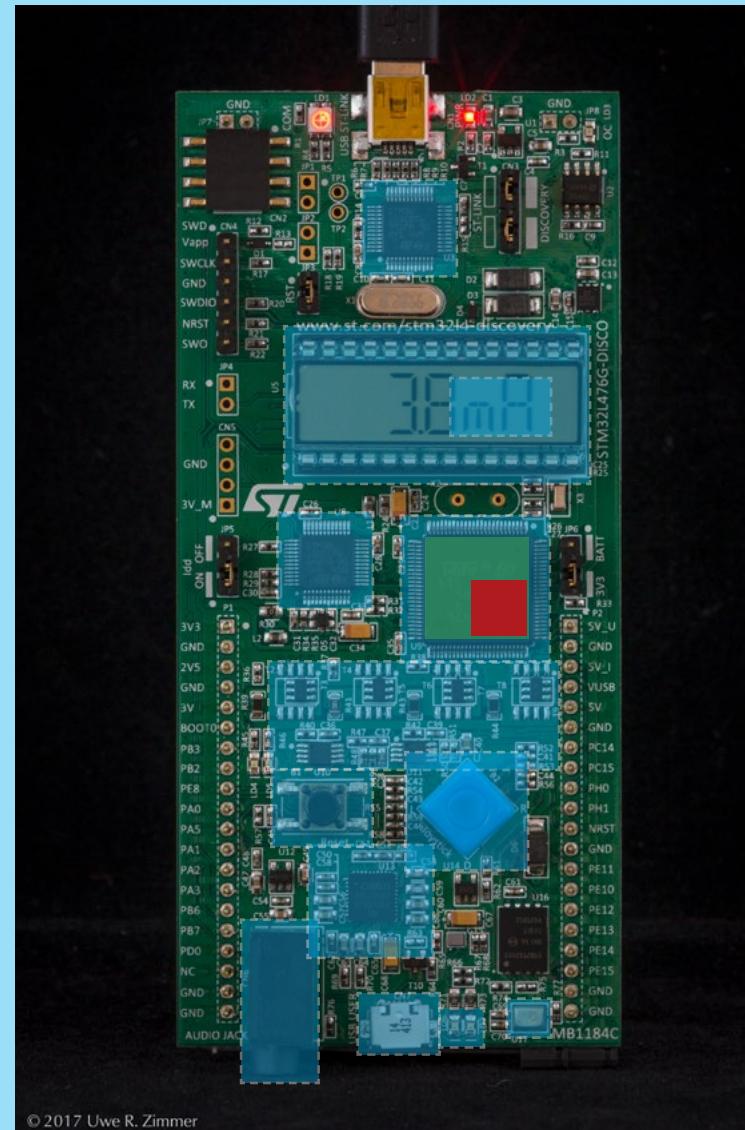
... running its sequence
of machine instructions.

☞ How to interact with
all the other devices
inside the

MCU

?

☞ ... and then with all the
devices on the board?



☞ and then with the

rest of the world

... which is
connected to the board?



Asynchronism

Polling

Sequential machine instructions

- ☞ All external devices need to be “checked” by asking for their status.
- ☞ This should usually happen (semi-) regularly.



Asynchronism

Polling

Sequential machine instructions

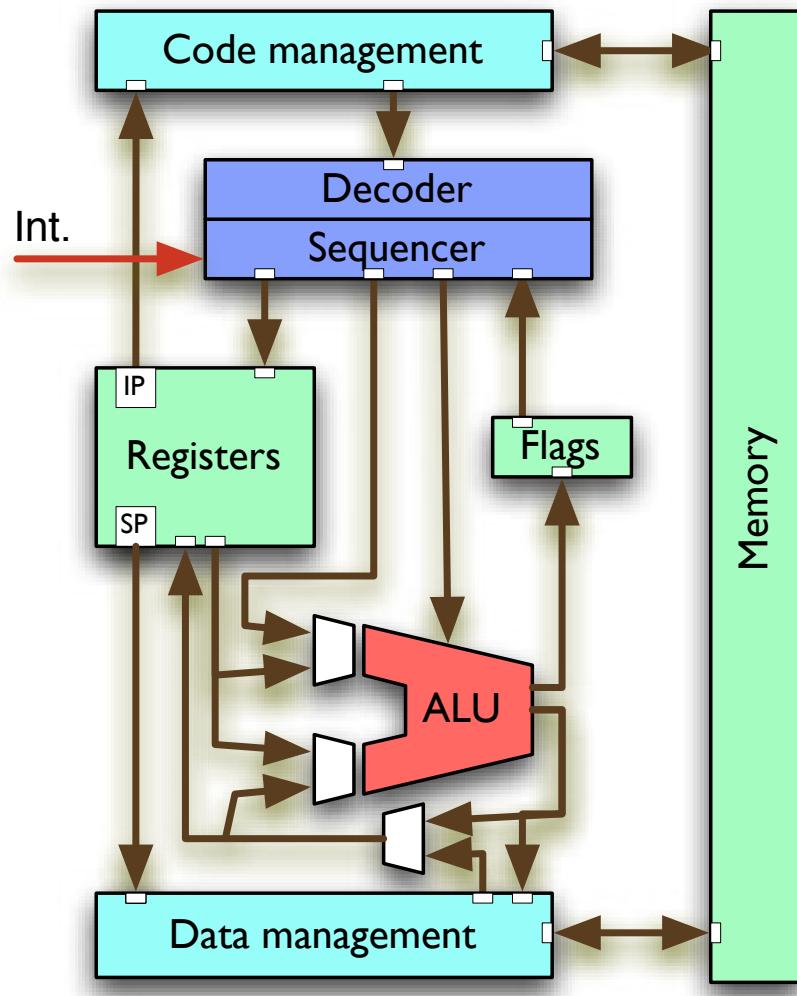
- ☞ All external devices need to be “checked” by asking for their status.
- ☞ This should usually happen (semi-) regularly.
 - ☞ This will lead to a loop of **polling** requests.
- ↗ Maximal latencies can be calculated straight forward.
- ↗ Simplicity of design (with small number of devices).
- ↗ Fastest option with small number of devices (like: one).
- ↘ All devices will need to wait their turn
 - ... even if this device is the only one with new data!
- ↘ The “main” program transforms into one large loop which can be hard to handle in terms of scalable program design.
- ↘ Events or data can be missed.



Asynchronism

Processor Architectures

Interrupts



- One or multiple lines wired directly into the sequencer
 - ☞ Required for:
Pre-emptive scheduling, Timer driven actions, Transient hardware interactions, ...
 - ☞ Usually preceded by an external logic ("interrupt controller") which accumulates and encodes all external requests.

On interrupt (if unmasked):

- CPU stops normal sequencer flow.
- Lookup of interrupt handler's address
- Current IP and state pushed onto stack.
- IP set to interrupt handler.

We successfully interrupted
a sequence of operations ...



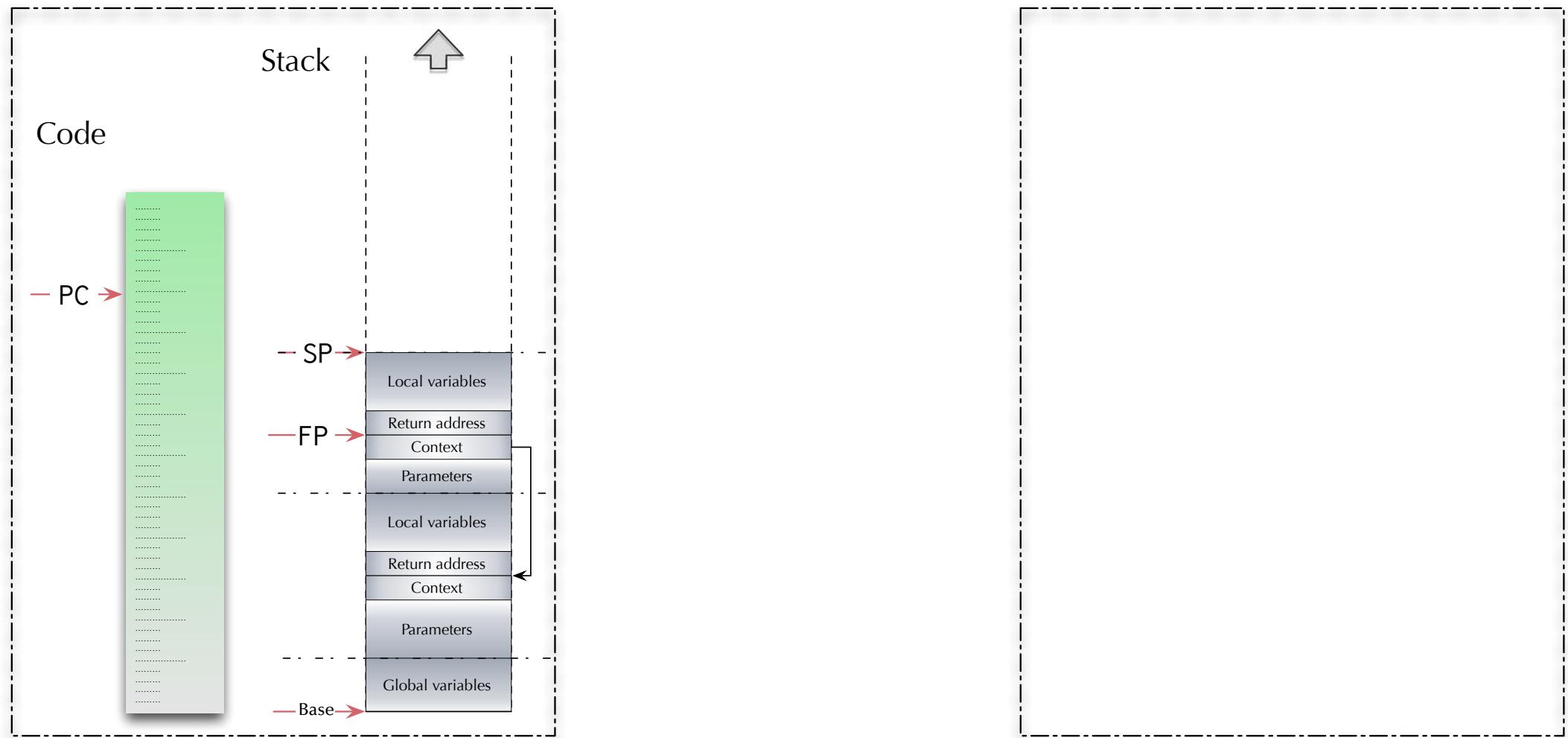


Asynchronism

Interrupt processing

Interrupt handler

Program





Asynchronism

Interrupt processing

Interrupt handler

Program

Stack

Code

— PC →

...

— SP →

Local variables

— FP →

Return address

Context

Parameters

— Base →

Local variables

Return address

Context

Parameters

Global variables

↓

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑

↑



Asynchronism

Interrupt processing

Interrupt handler

Program

Stack

Code



- PC →

Push registers
Declare local variables

- SP →

Local
variables

Registers

Local variables

Return address

Context

Parameters

Local variables

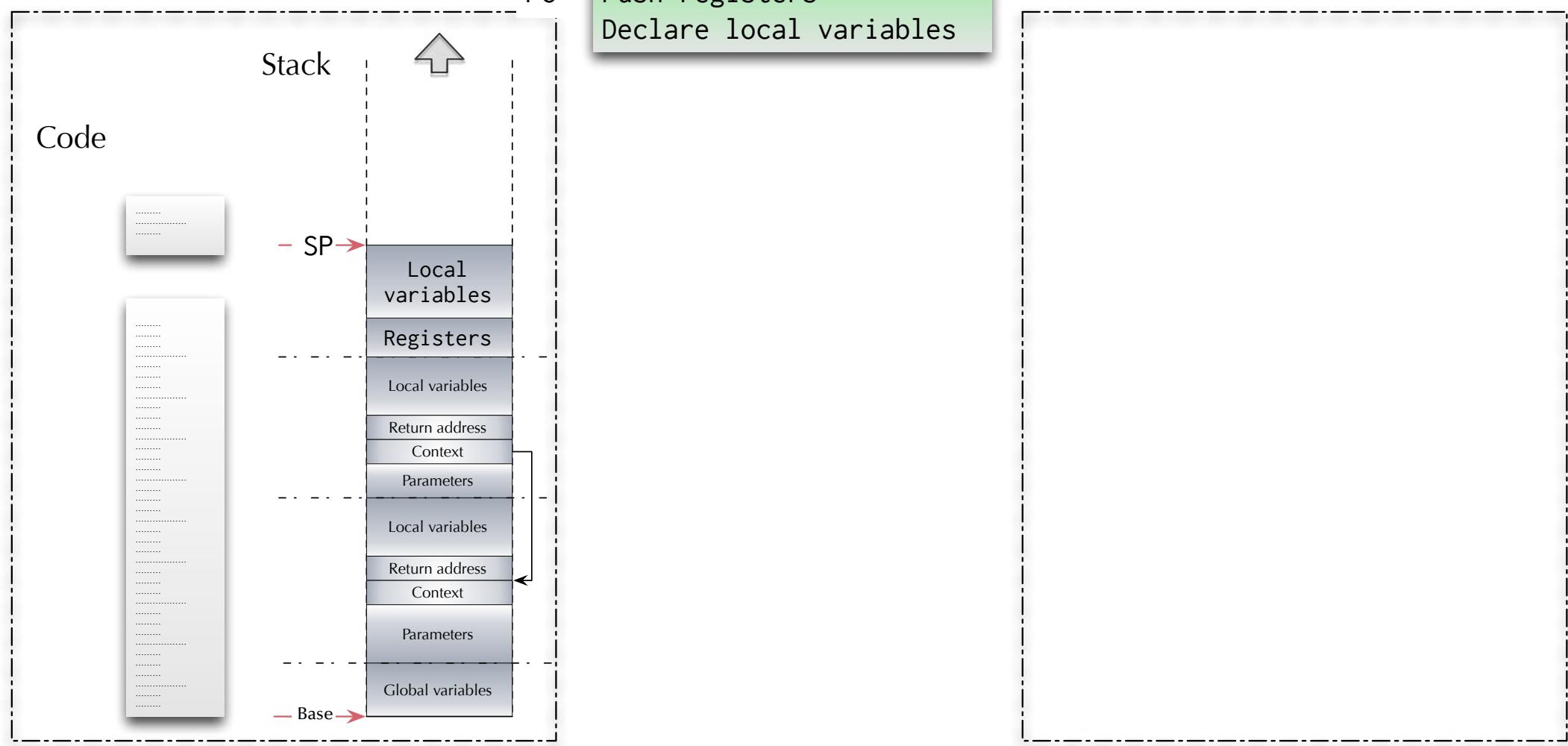
Return address

Context

Parameters

Global variables

- Base →



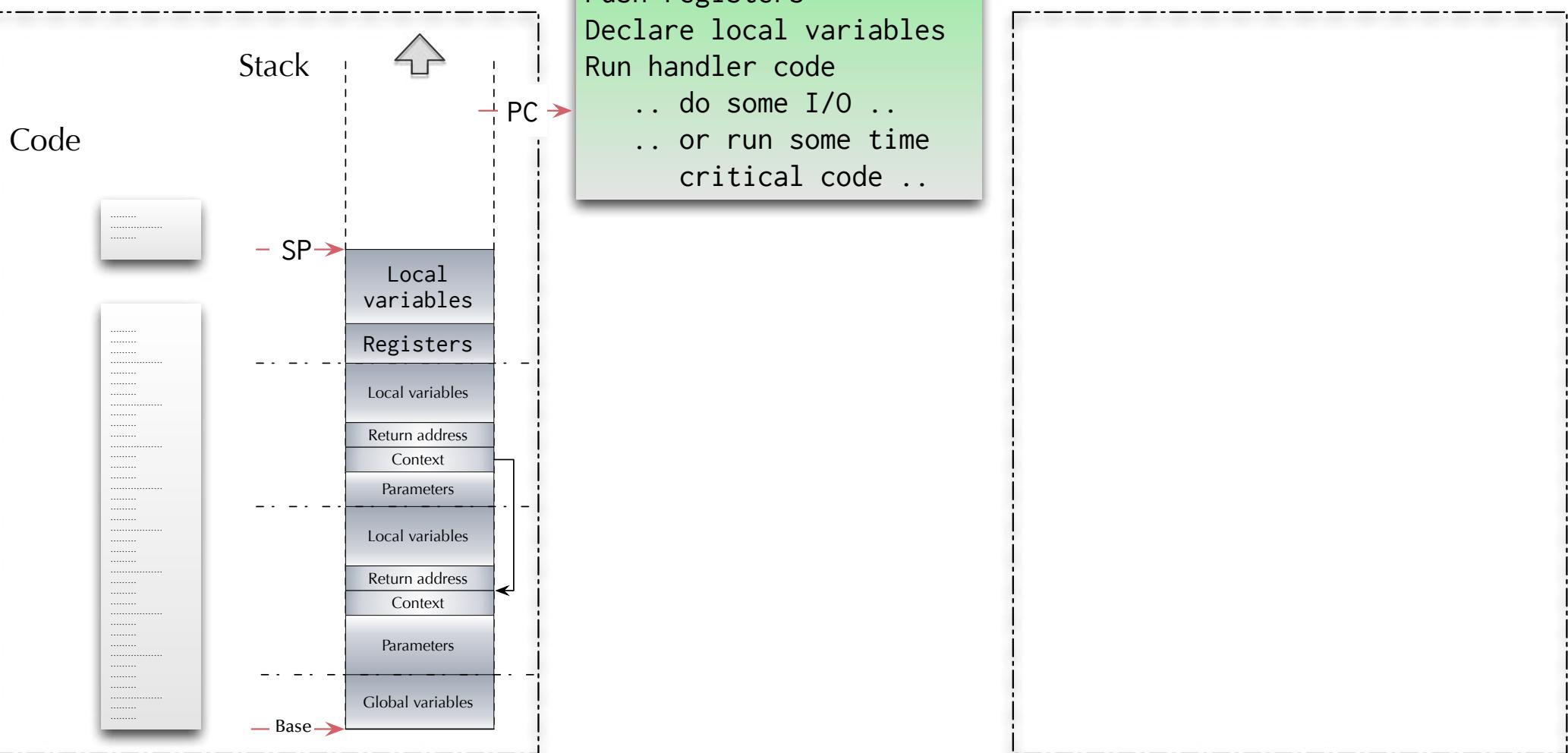


Asynchronism

Interrupt processing

Interrupt handler

Program



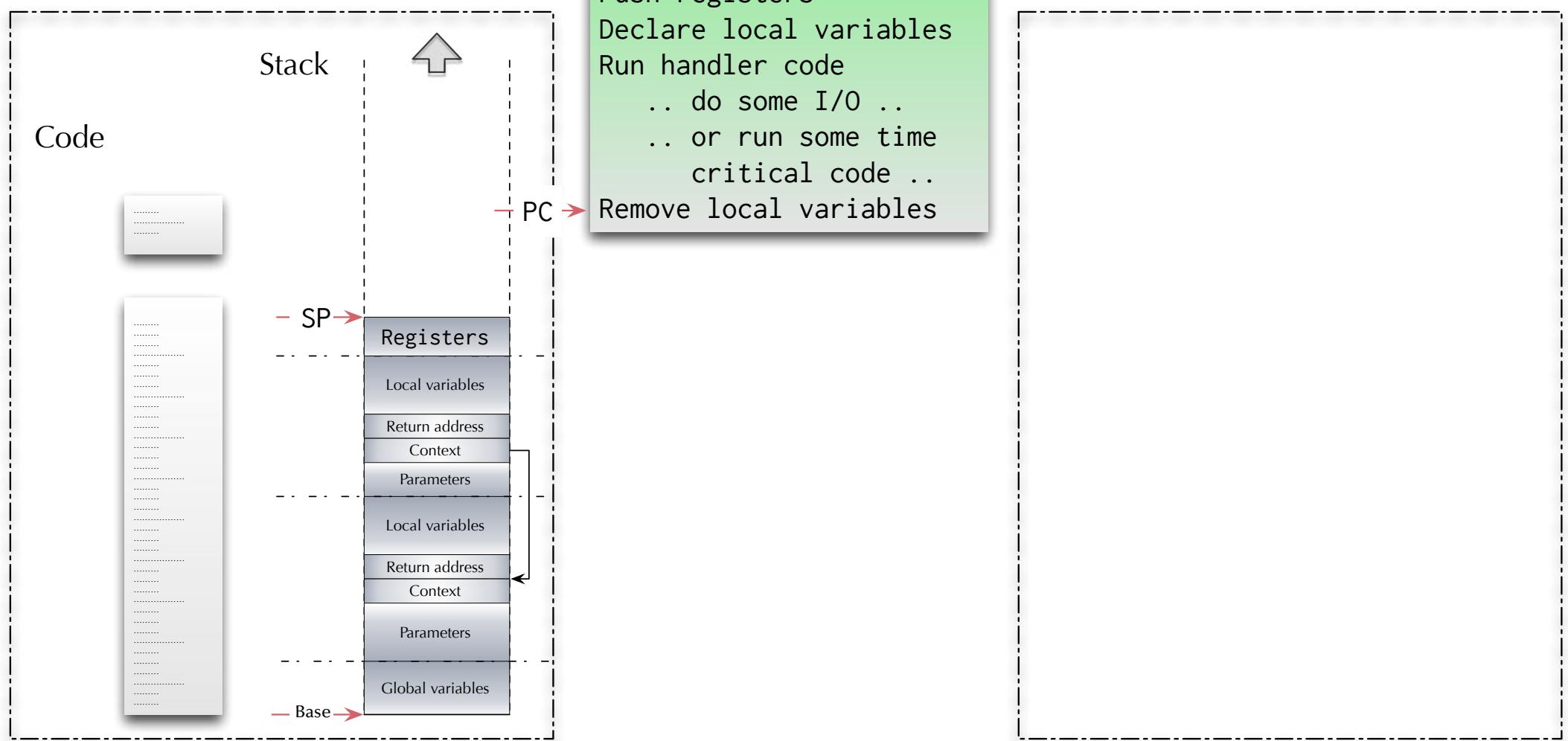


Asynchronism

Interrupt processing

Interrupt handler

Program



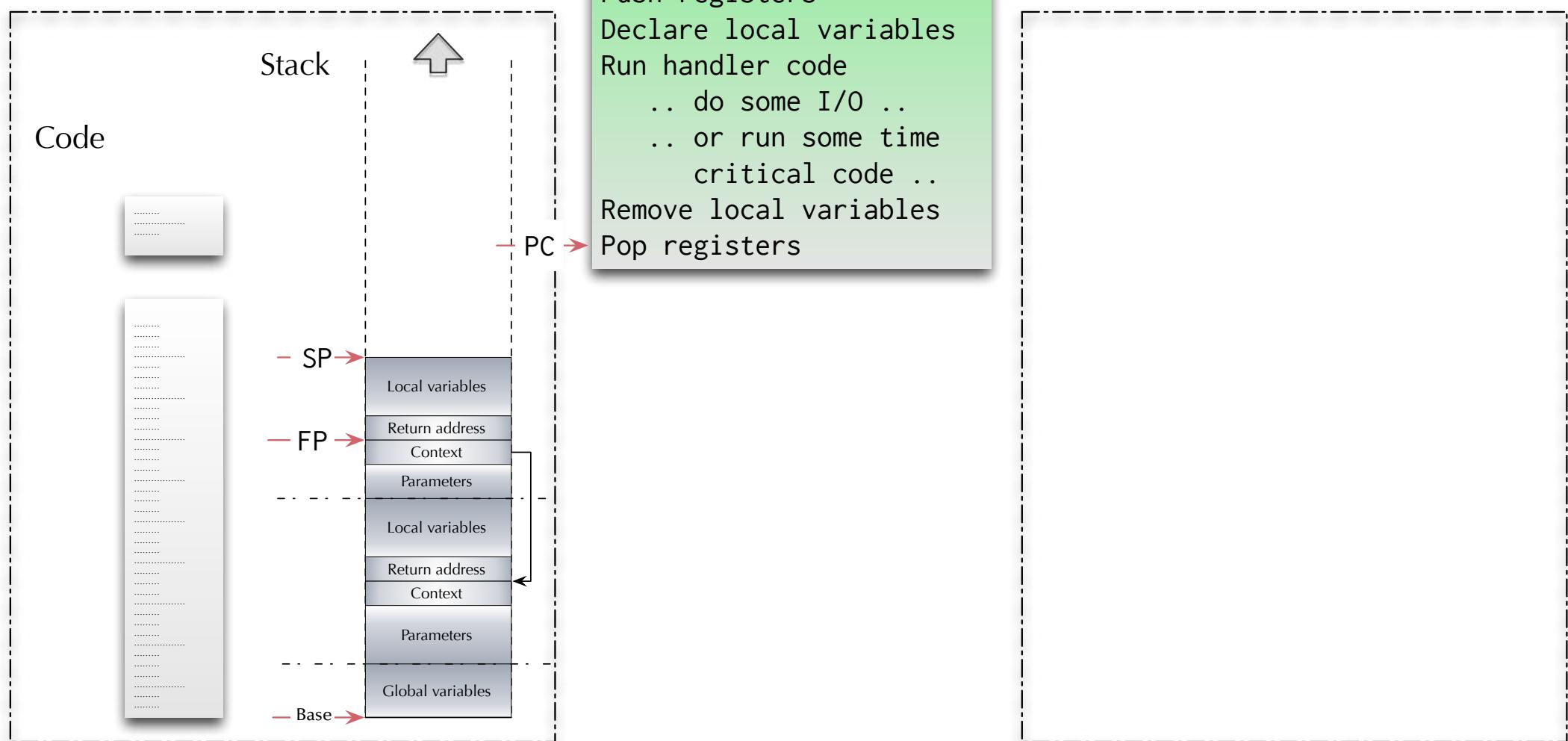


Asynchronism

Interrupt processing

Interrupt handler

Program



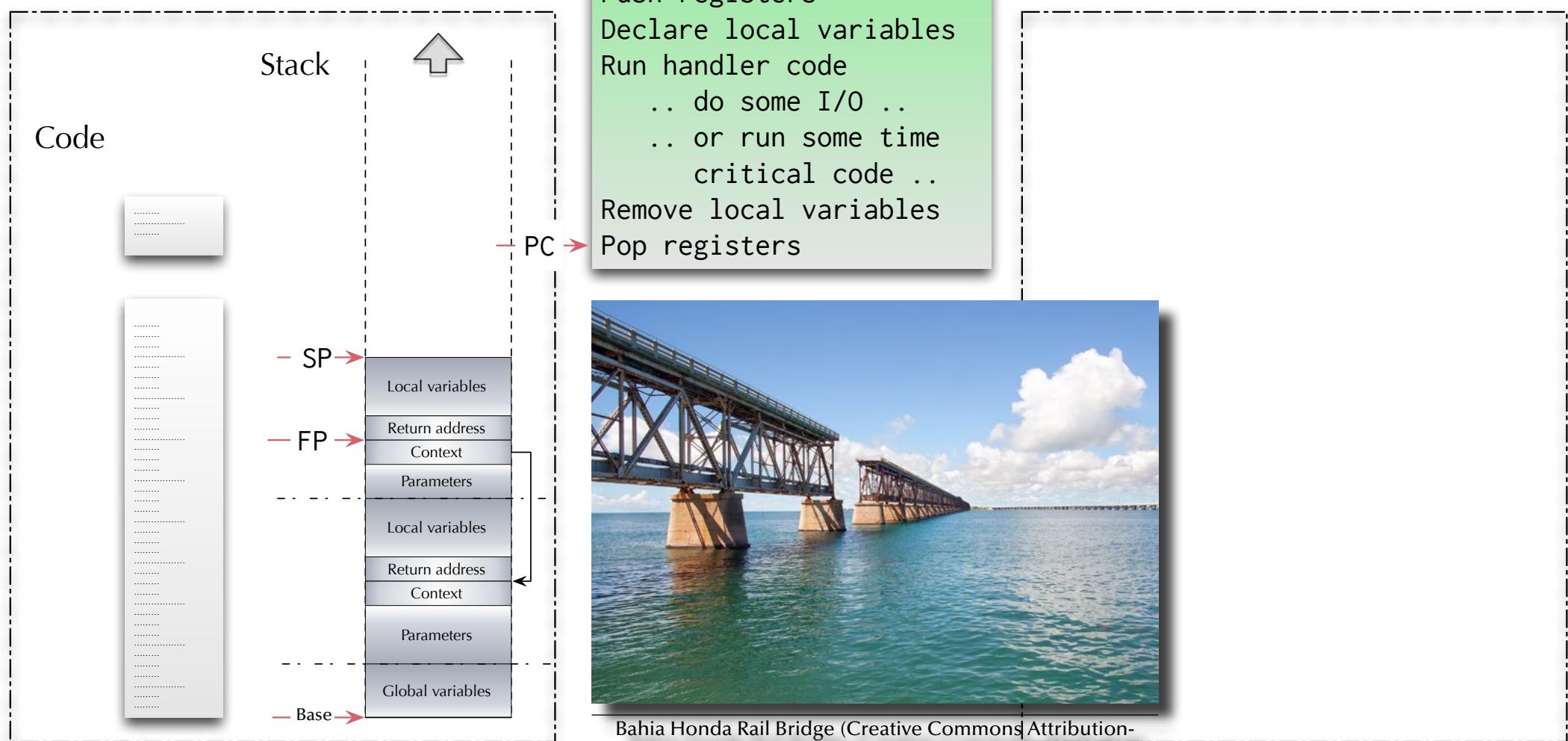


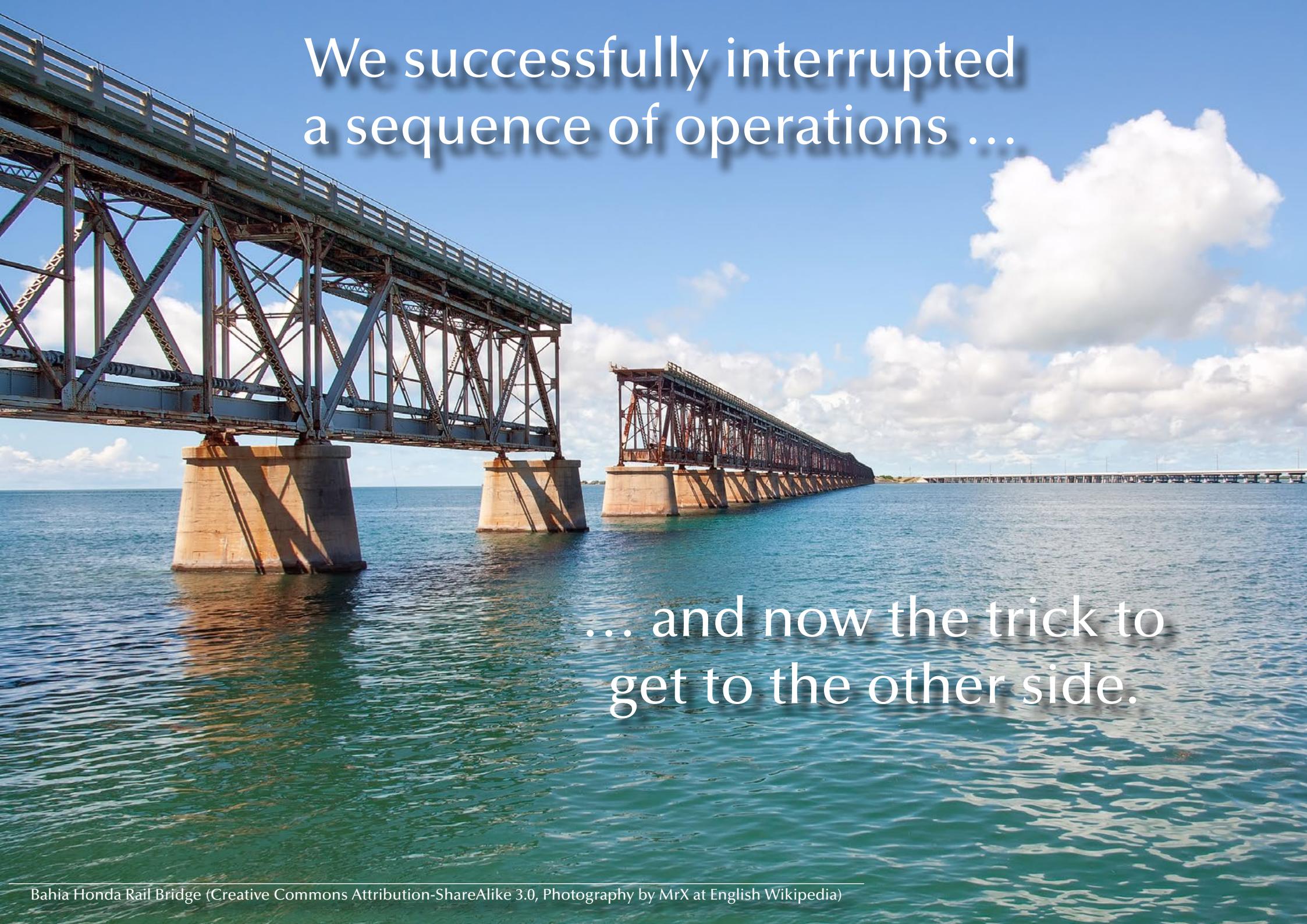
Asynchronism

Interrupt processing

Interrupt handler

Program





We successfully interrupted
a sequence of operations ...

... and now the trick to
get to the other side.

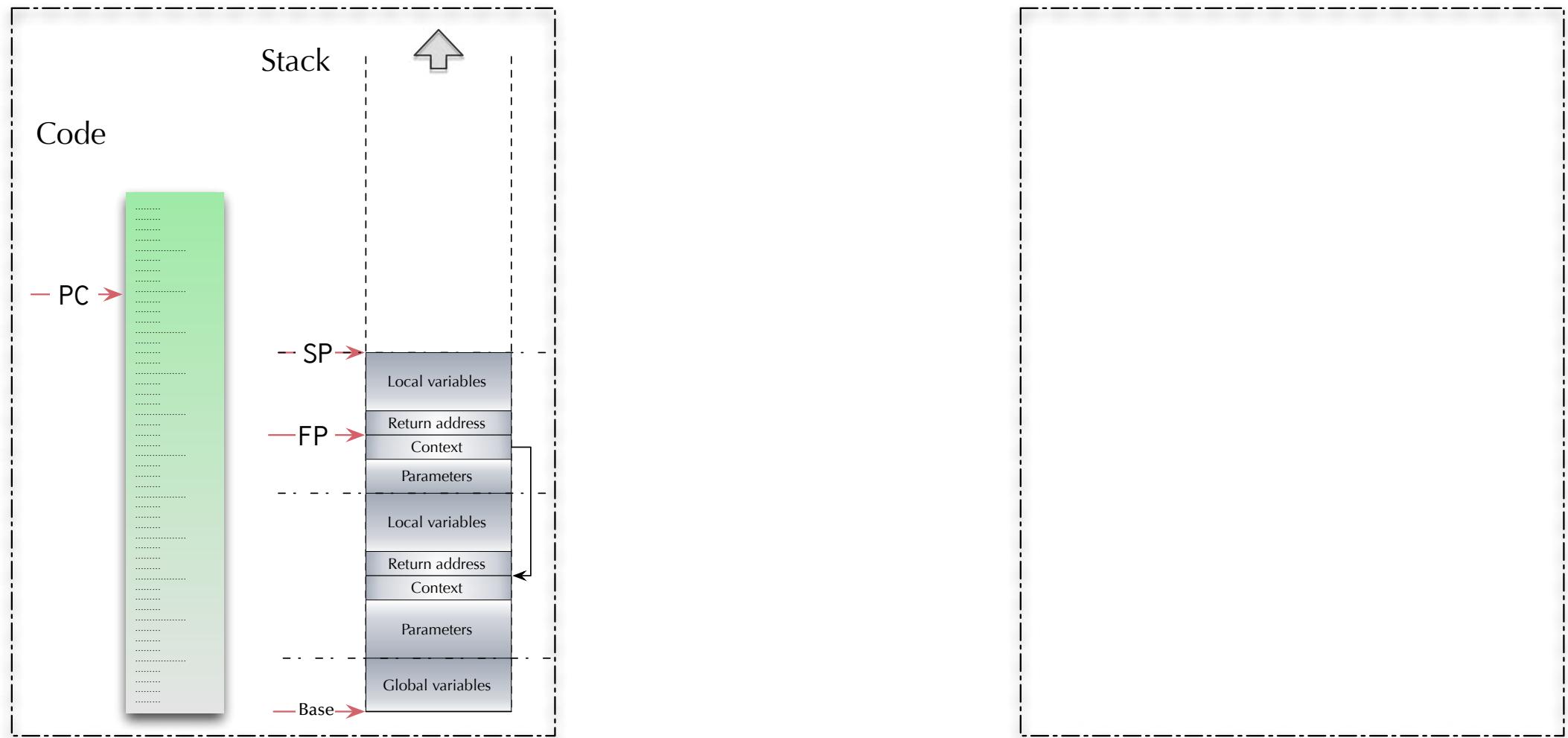


Asynchronism

Interrupt processing

Interrupt handler

Program





Asynchronism

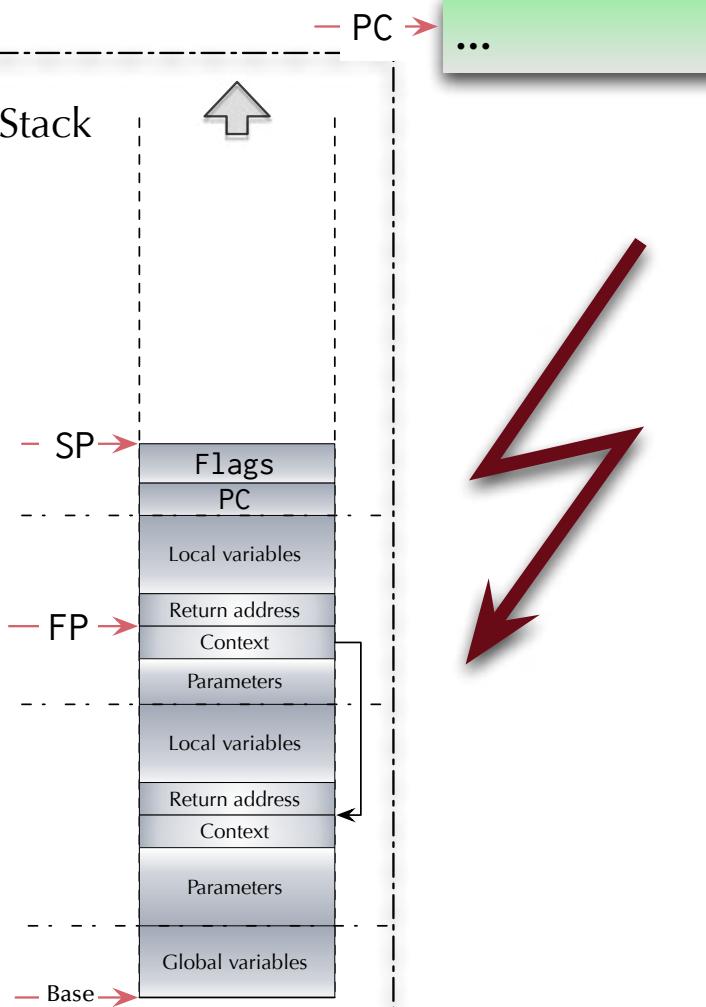
Interrupt processing

Interrupt handler

Program

Stack

Code





Asynchronism

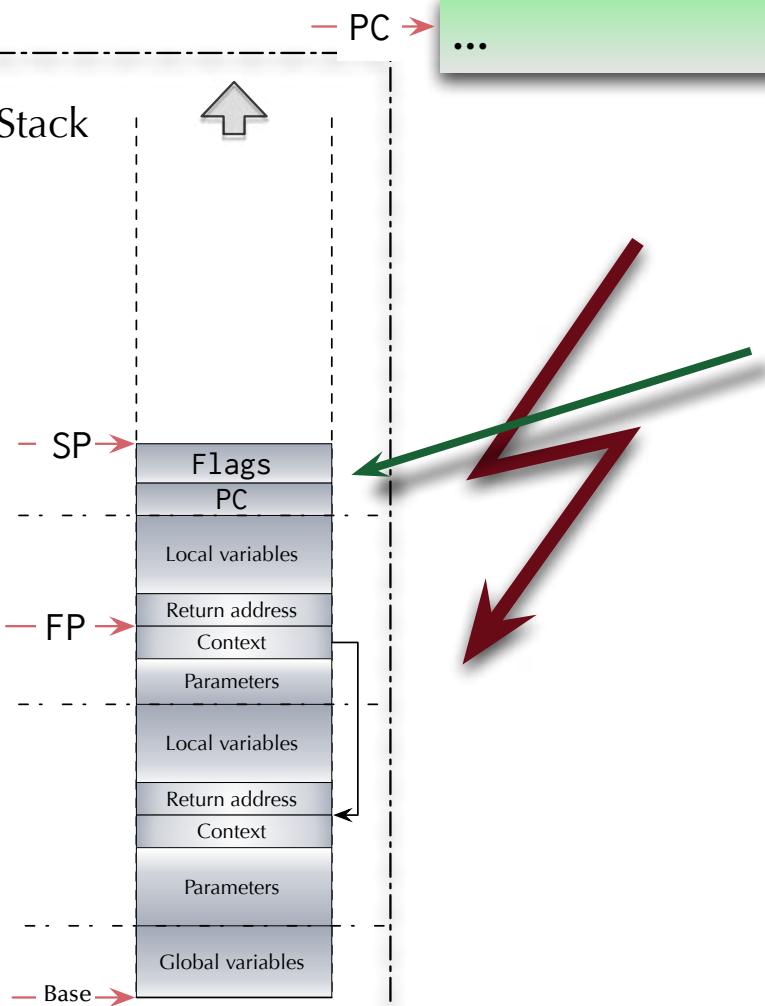
Interrupt processing

Interrupt handler

Program

Stack

Code



The CPU hardware (!)
did that,
before anything
was changed



Asynchronism

Interrupt processing

Interrupt handler

Program

Stack

Code



- PC →

Push registers
Declare local variables

- SP →

Local
variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

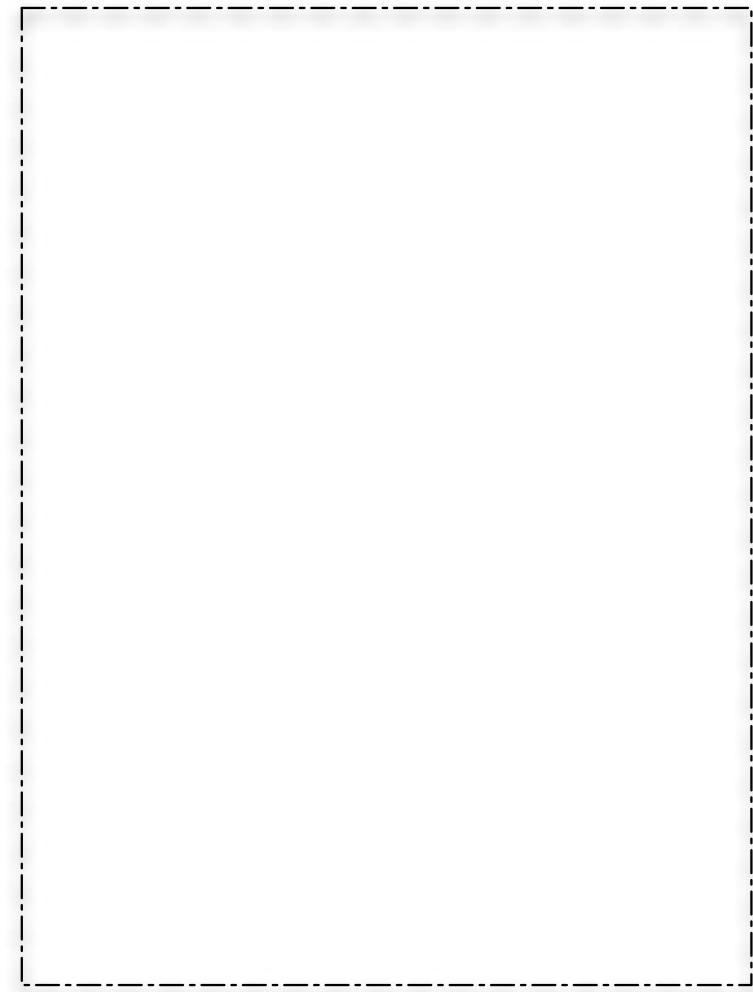
Return address

Context

Parameters

Global variables

- Base →



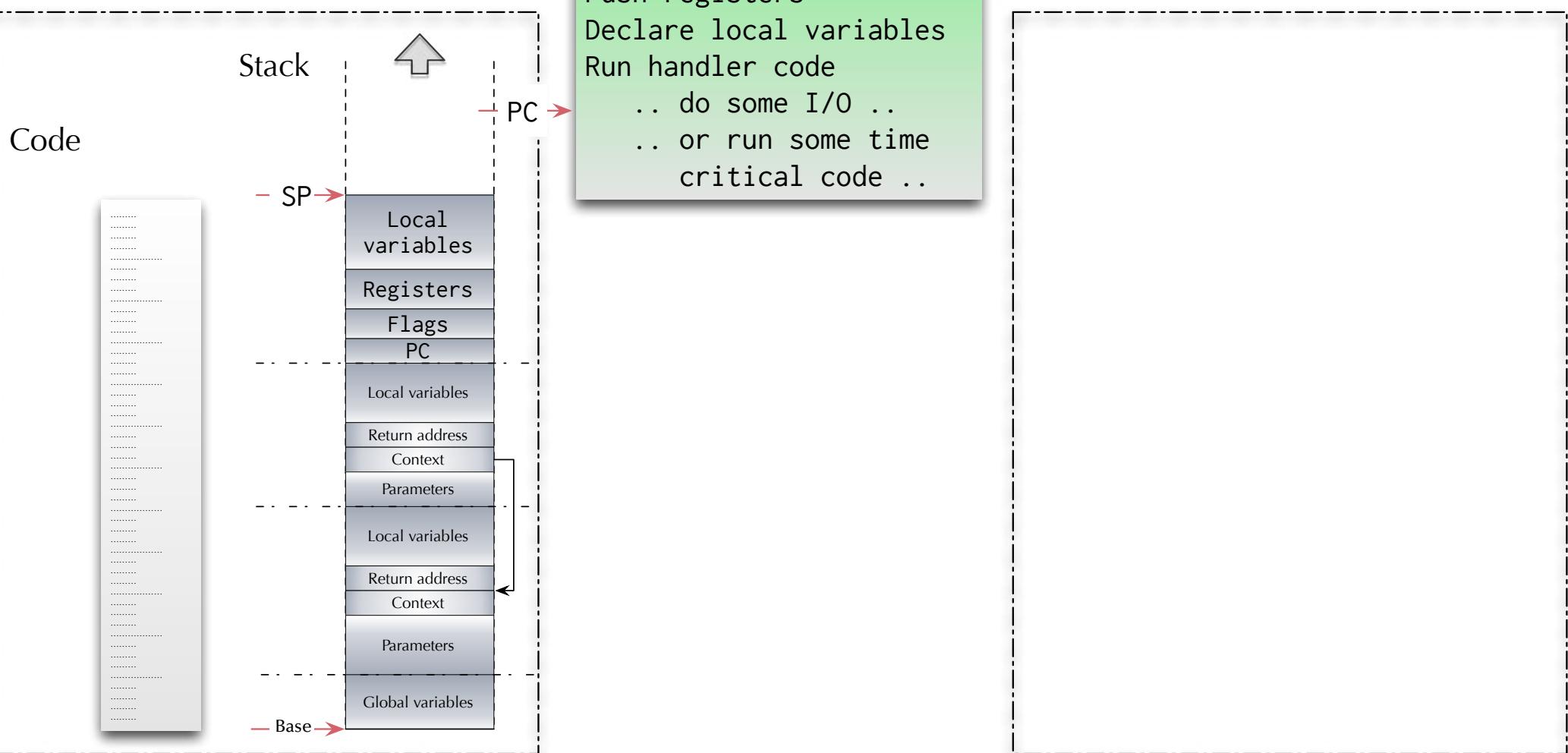


Asynchronism

Interrupt processing

Interrupt handler

Program



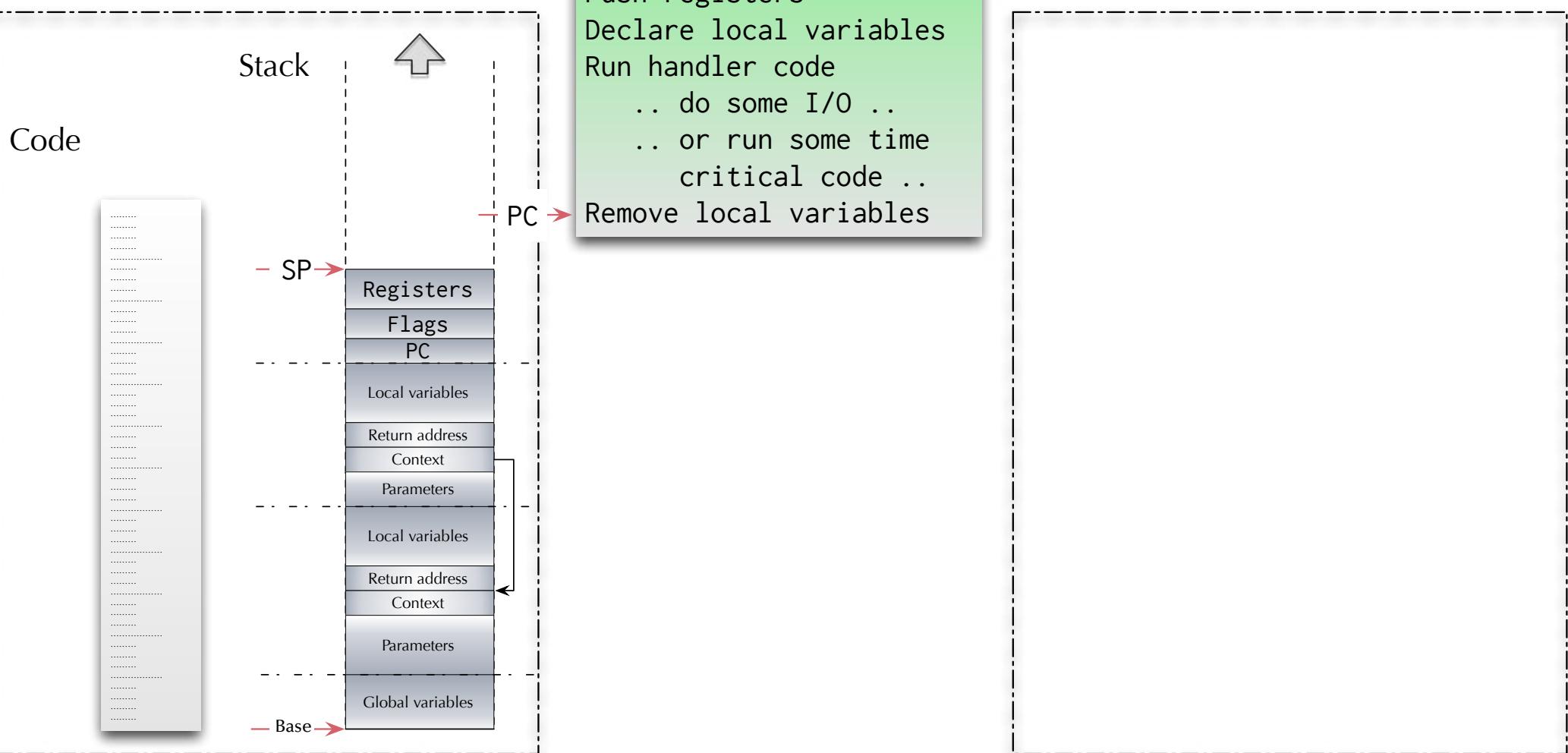


Asynchronism

Interrupt processing

Interrupt handler

Program



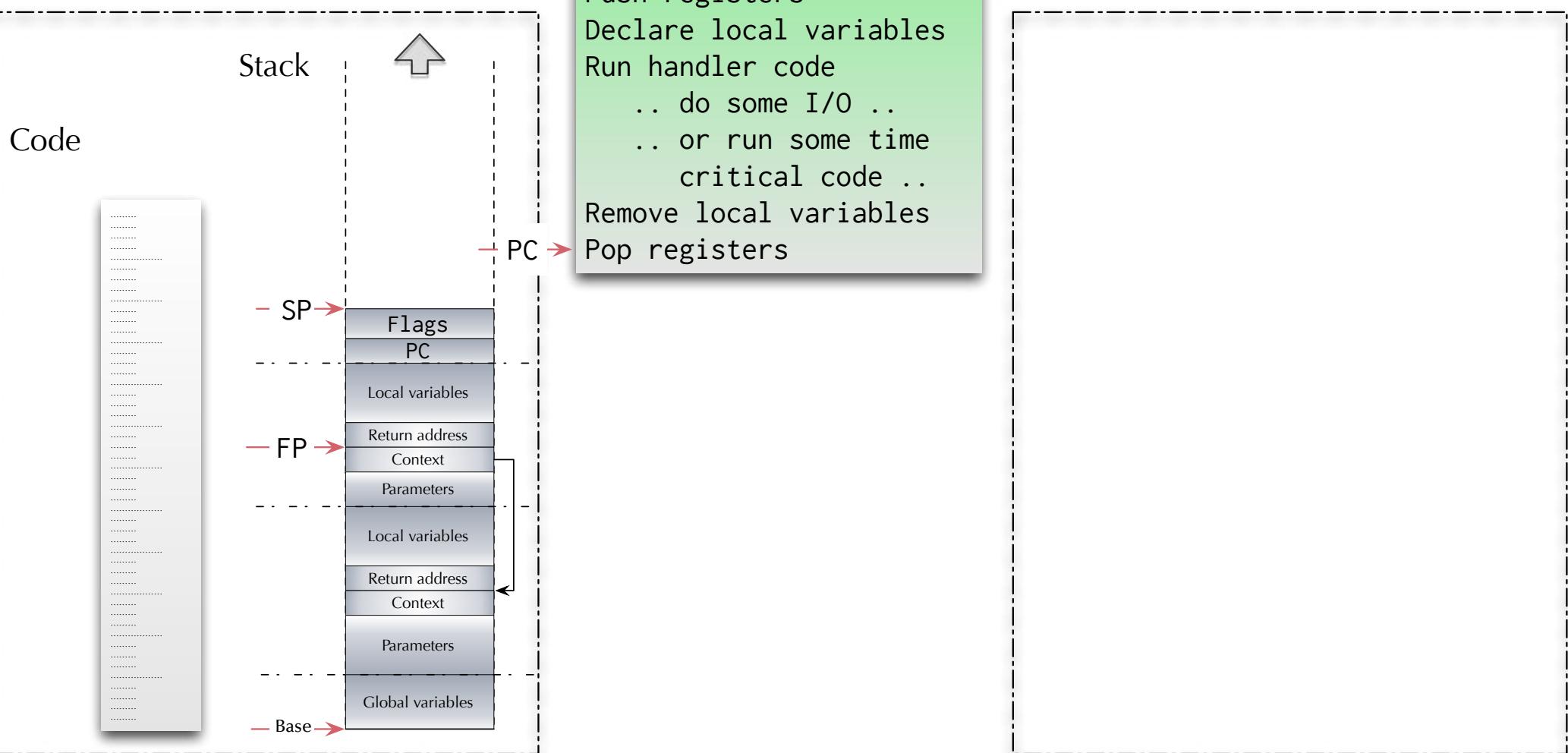


Asynchronism

Interrupt processing

Interrupt handler

Program



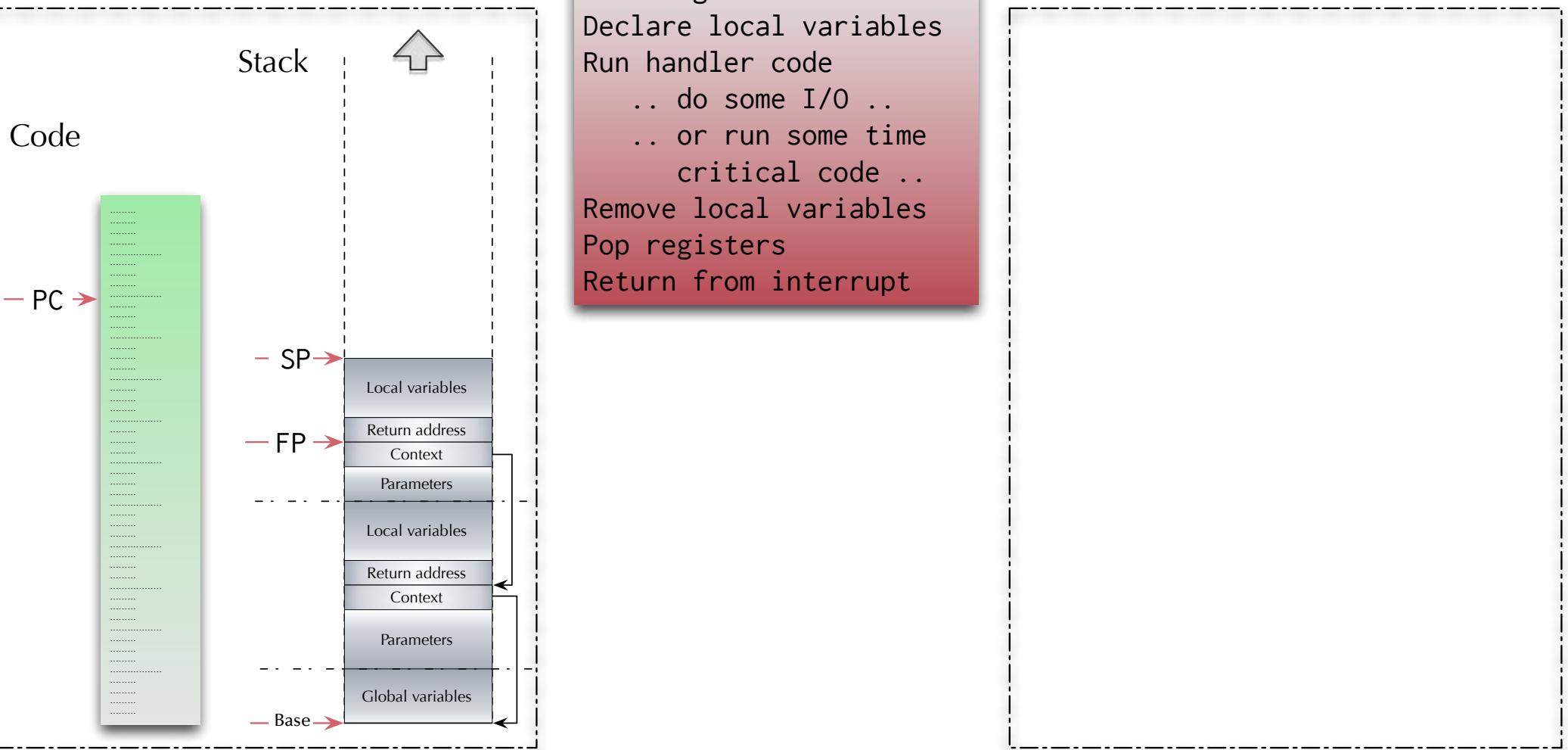


Asynchronism

Interrupt processing

Interrupt handler

Program



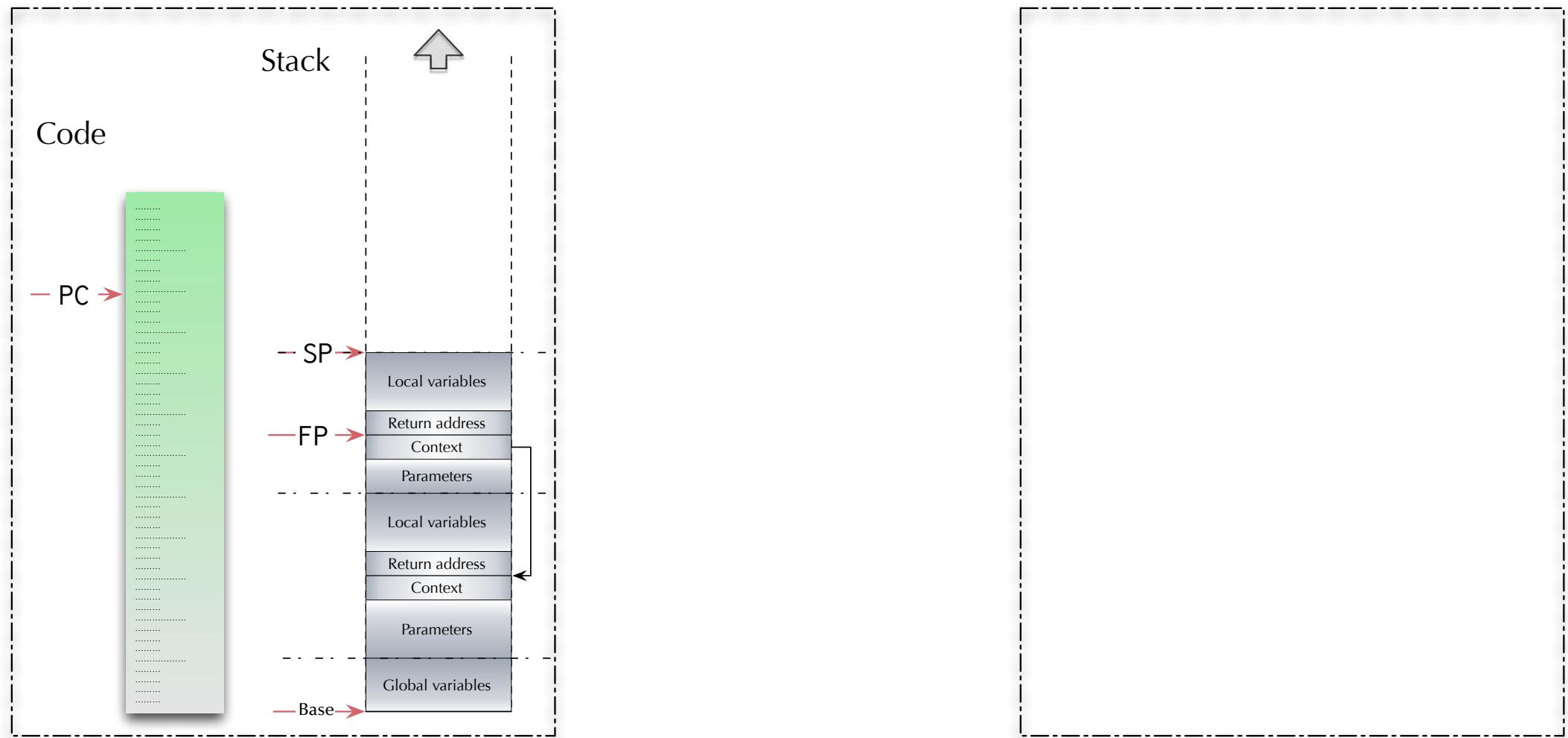


Asynchronism

Interrupt processing

Interrupt handler

Program





Asynchronism

Interrupt processing

Interrupt handler

Program

Stack

Code

- SP →
Scratch registers

Flags

PC

Local variables

- FP →
Return address

Context

Parameters

Local variables

- Base →
Return address

Context

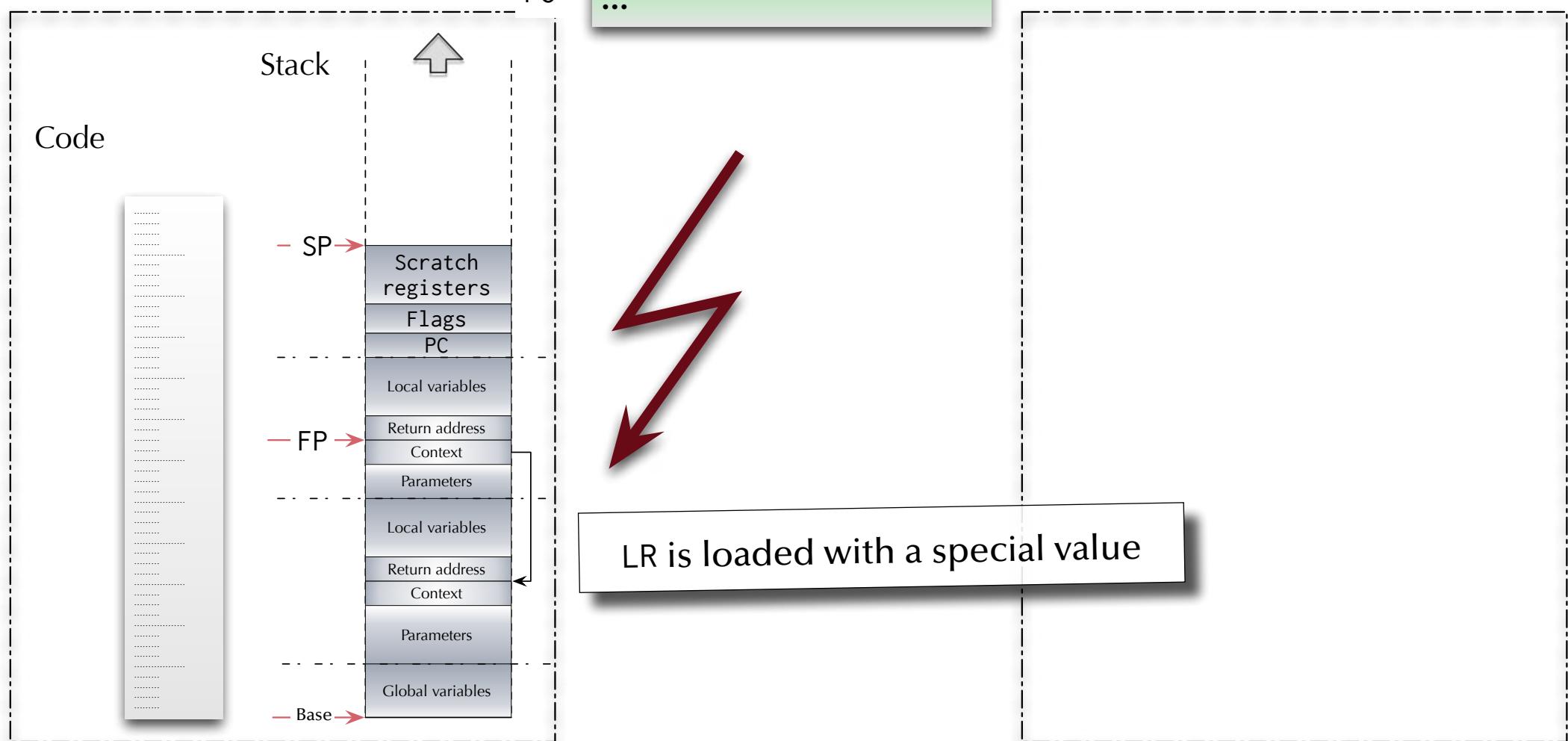
Parameters

Global variables

- PC →

...

LR is loaded with a special value





Asynchronism

Interrupt processing

Interrupt handler

Program

Stack



— PC →

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)

Code



— SP →

Scratch registers

Flags

PC

Local variables

— FP →

Return address

Context

Parameters

Local variables

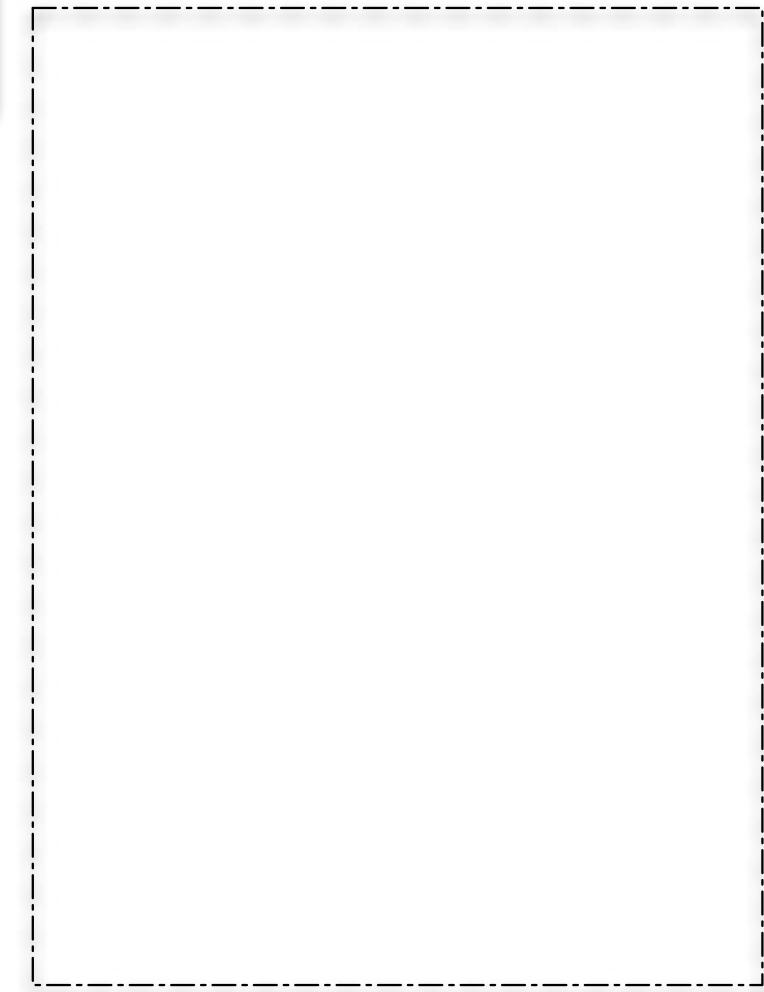
Return address

Context

Parameters

— Base →

Global variables



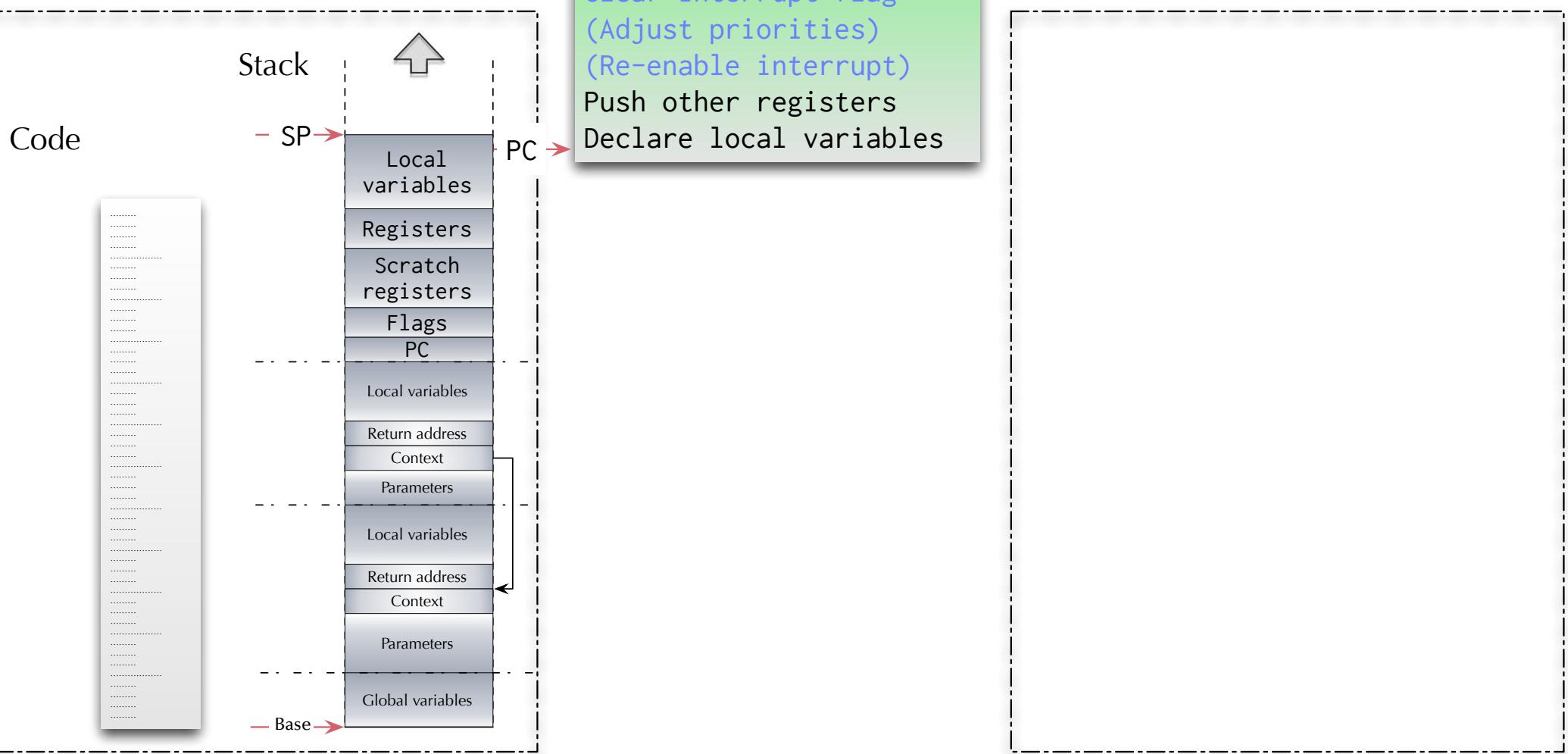


Asynchronism

Interrupt processing

Interrupt handler

Program



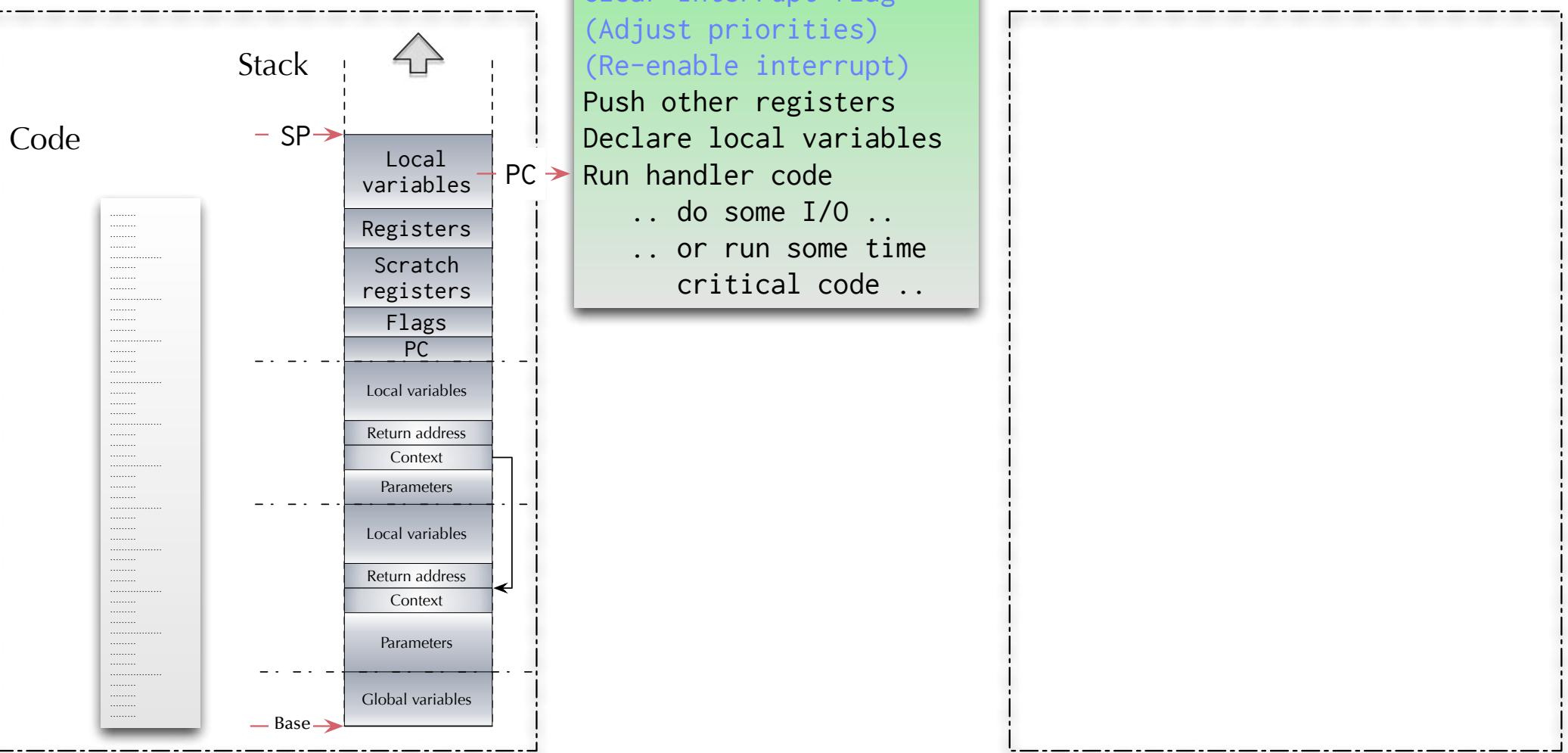


Asynchronism

Interrupt processing

Interrupt handler

Program



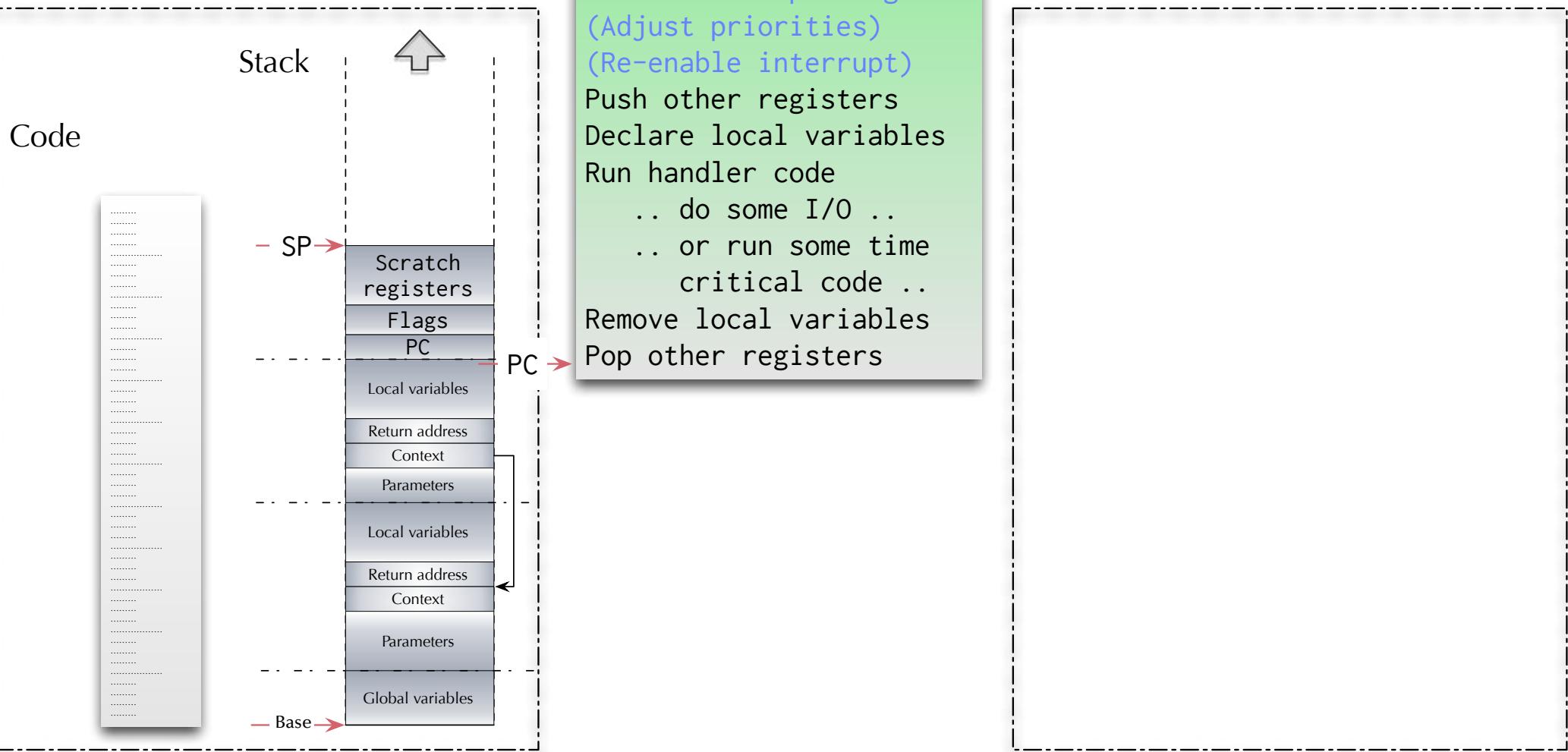


Asynchronism

Interrupt processing

Interrupt handler

Program



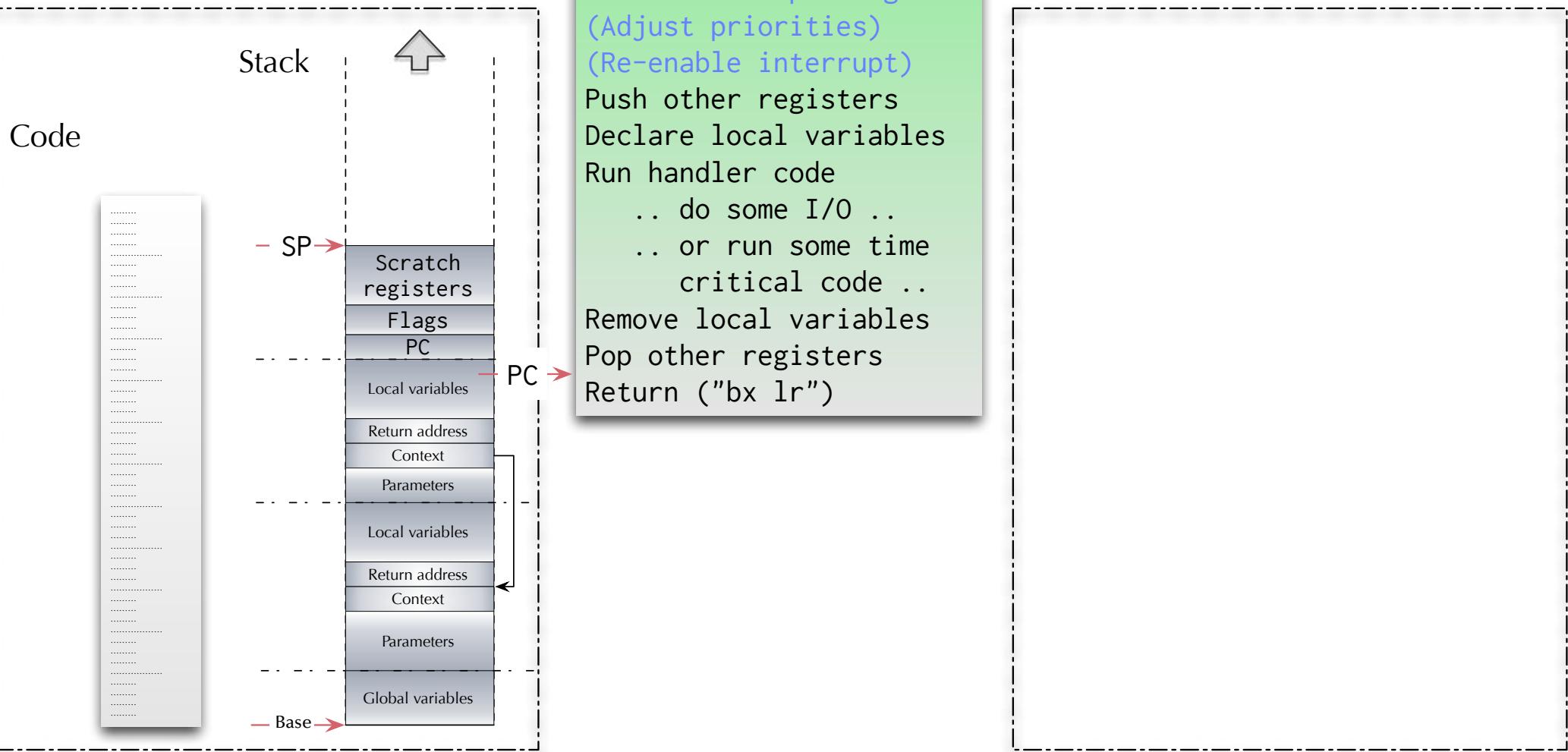


Asynchronism

Interrupt processing

Interrupt handler

Program



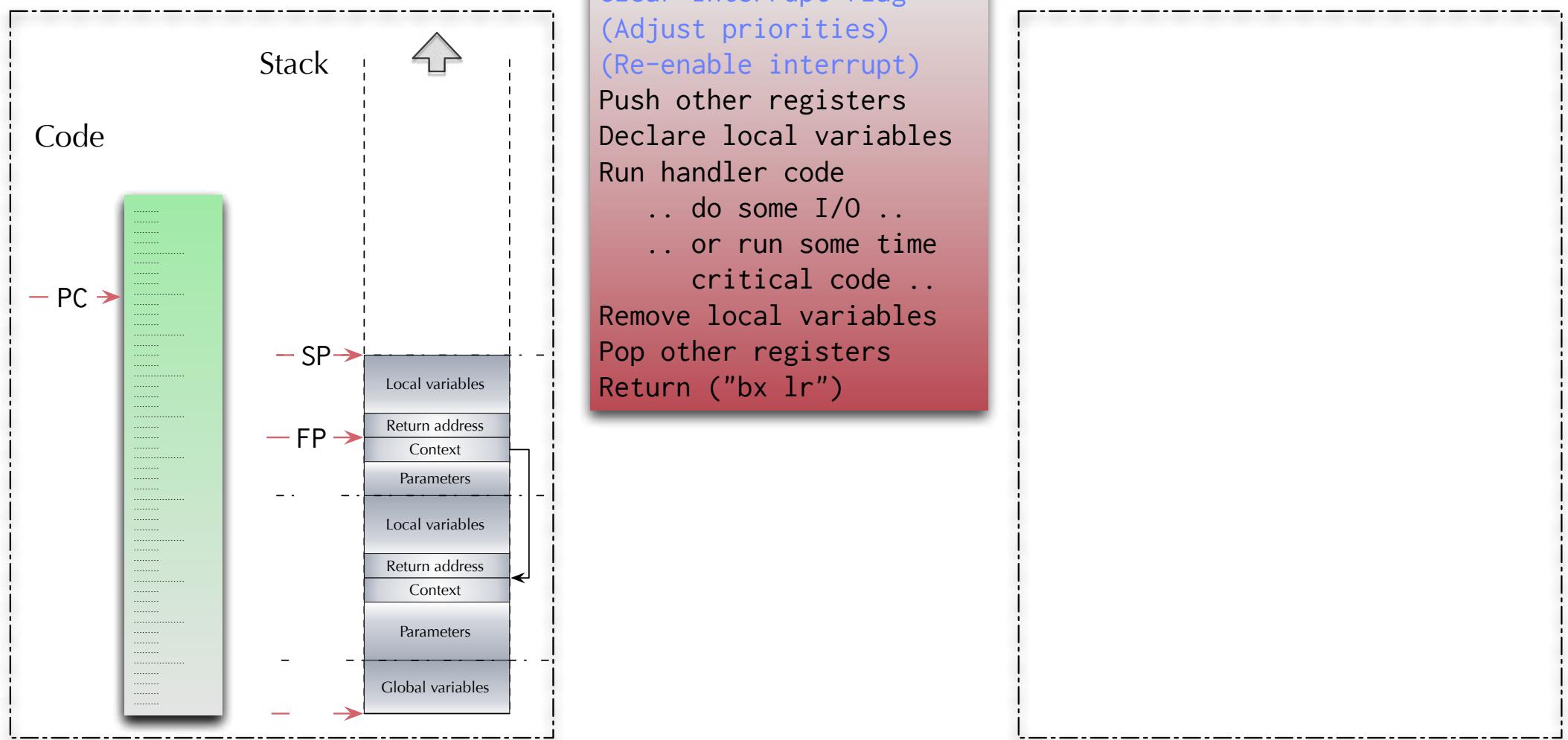


Asynchronism

Interrupt processing

Interrupt handler

Program





Asynchronism

Interrupt handler

Things to consider

- ☞ Interrupt handler code can be interrupted as well.
- ☞ Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?
- ☞ Can you overrun a stack with interrupt handlers?



Asynchronism

Interrupt handler

Things to consider

- ☞ Interrupt handler code can be interrupted as well.
- ☞ Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?
- ☞ Can you overrun a stack with interrupt handlers?
- ☞ Can we have one of those?

Busy!
Do Not Disturb!



Asynchronism

Multiple programs

If we can execute interrupt handler code
“concurrently” to our “main” program:

- ➡ Can we then also have multiple “main” programs?

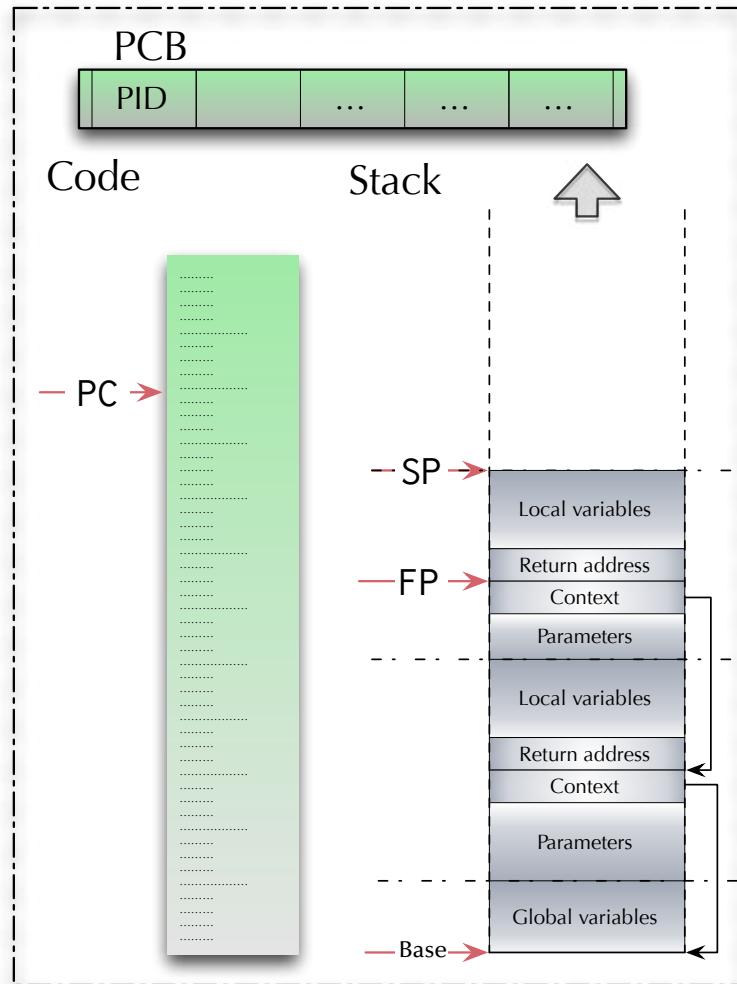


Asynchronism

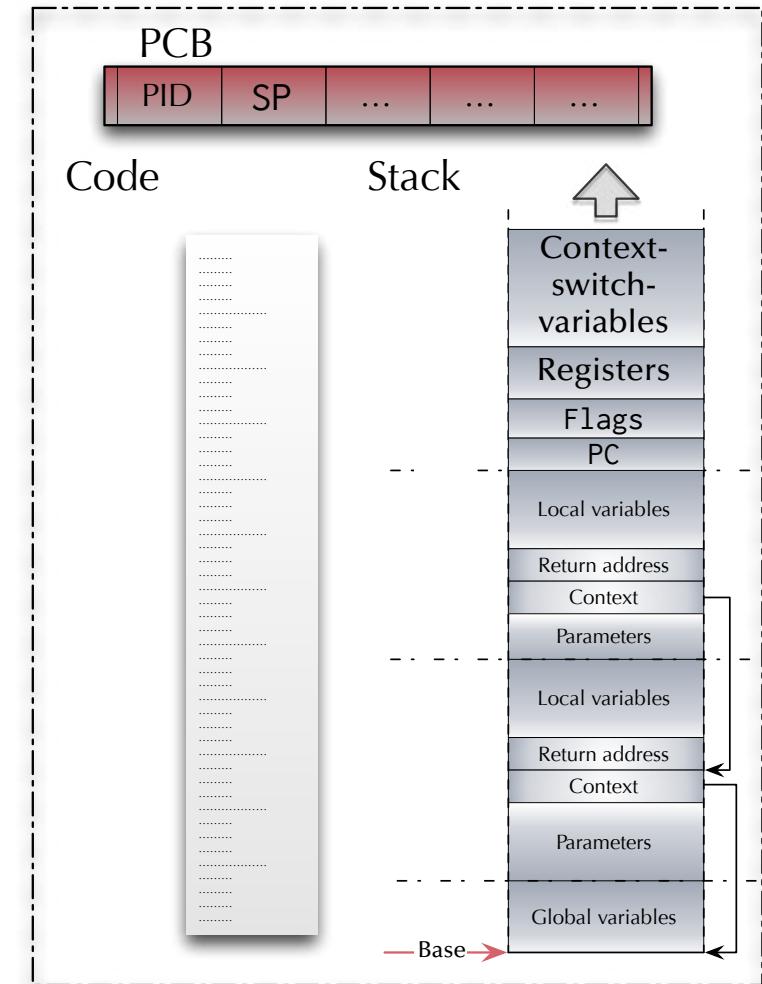
Context switch

Dispatcher

Process 1



Process 2



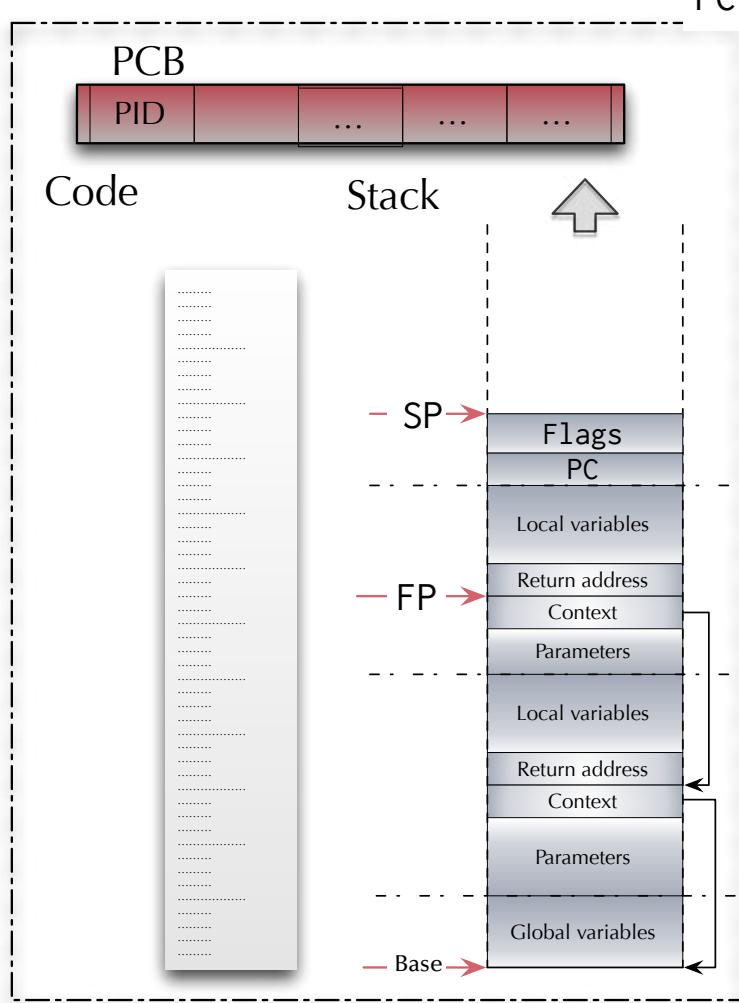


Asynchronism

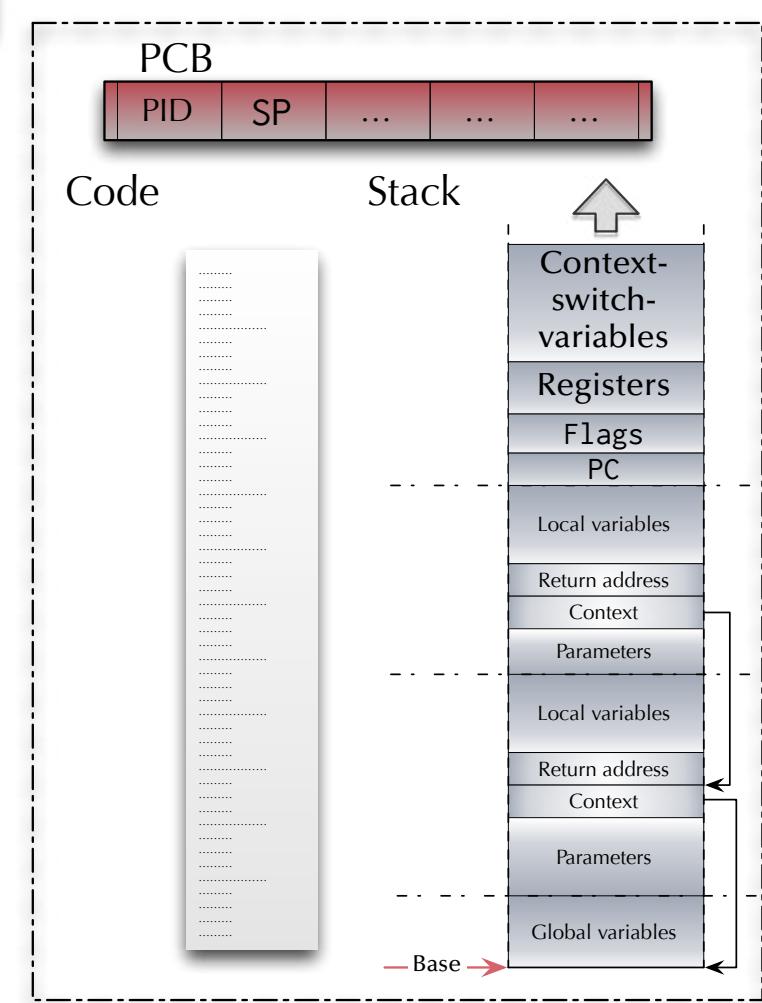
Context switch

Dispatcher

Process 1



Process 2



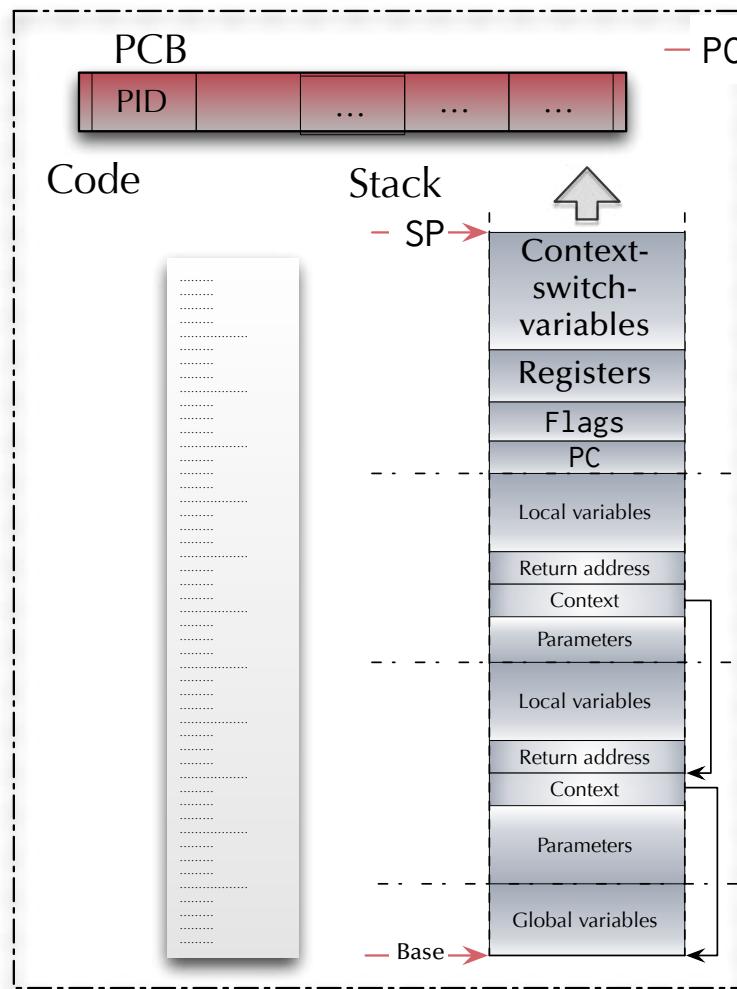


Asynchronism

Context switch

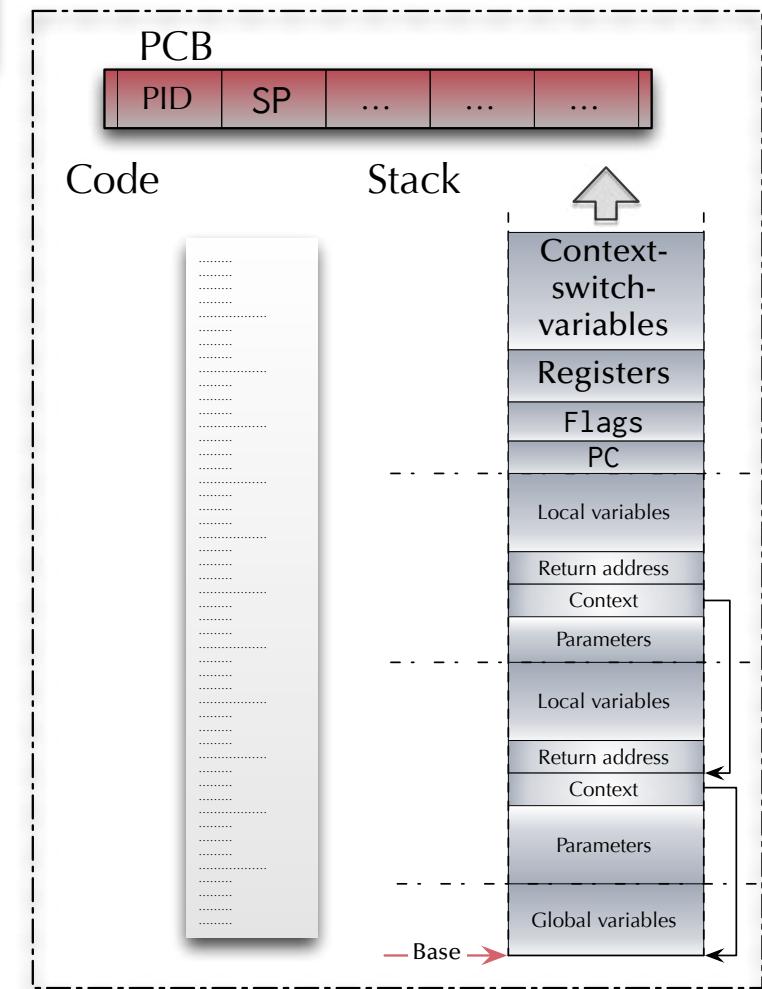
Dispatcher

Process 1



Push registers
Declare local variables

Process 2



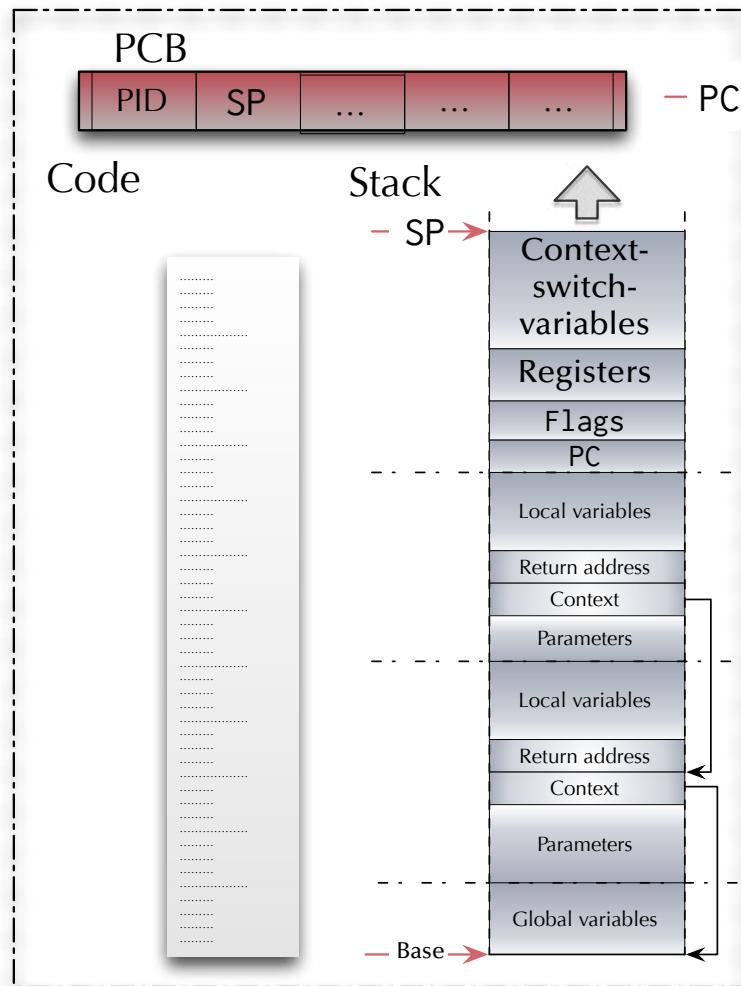


Asynchronism

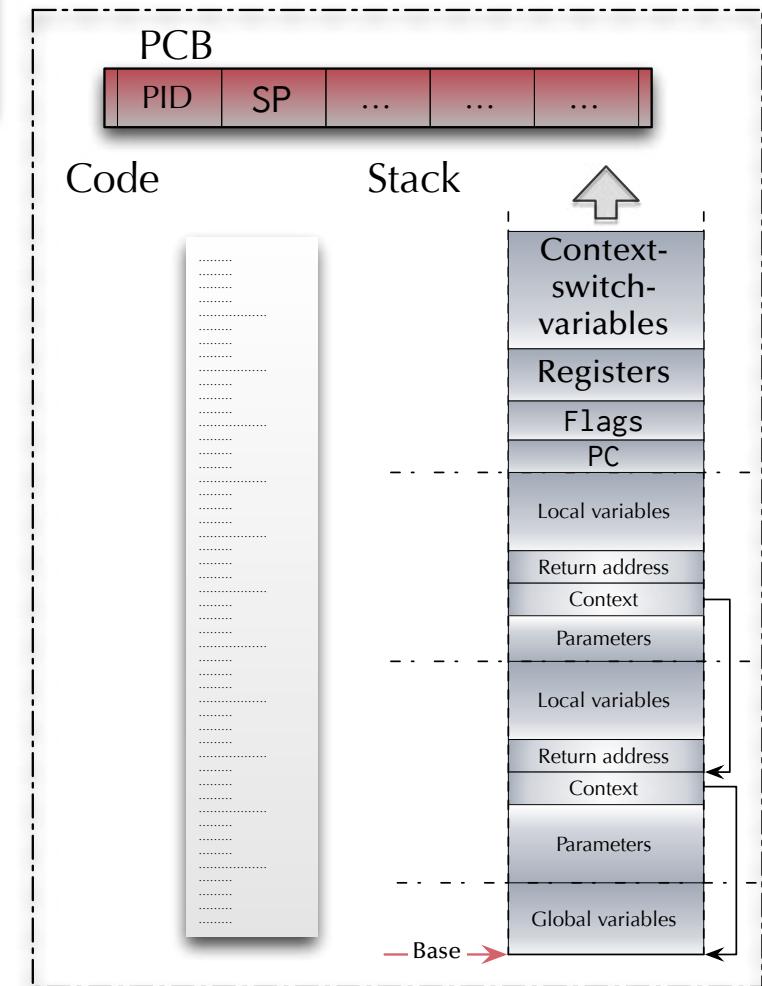
Context switch

Dispatcher

Process 1



Process 2



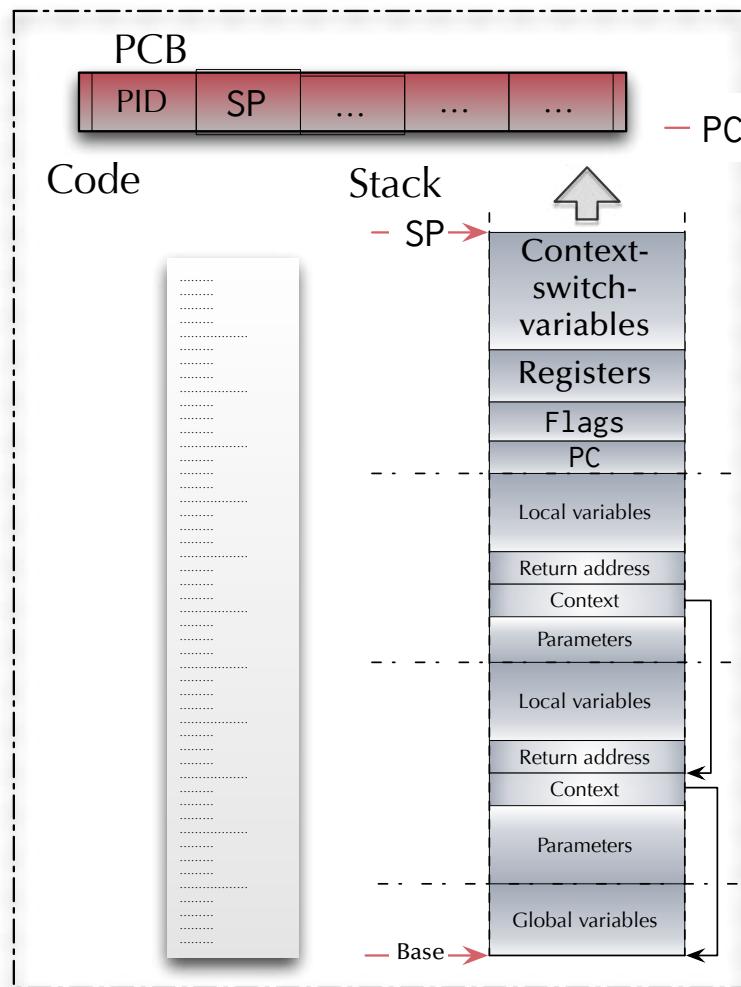


Asynchronism

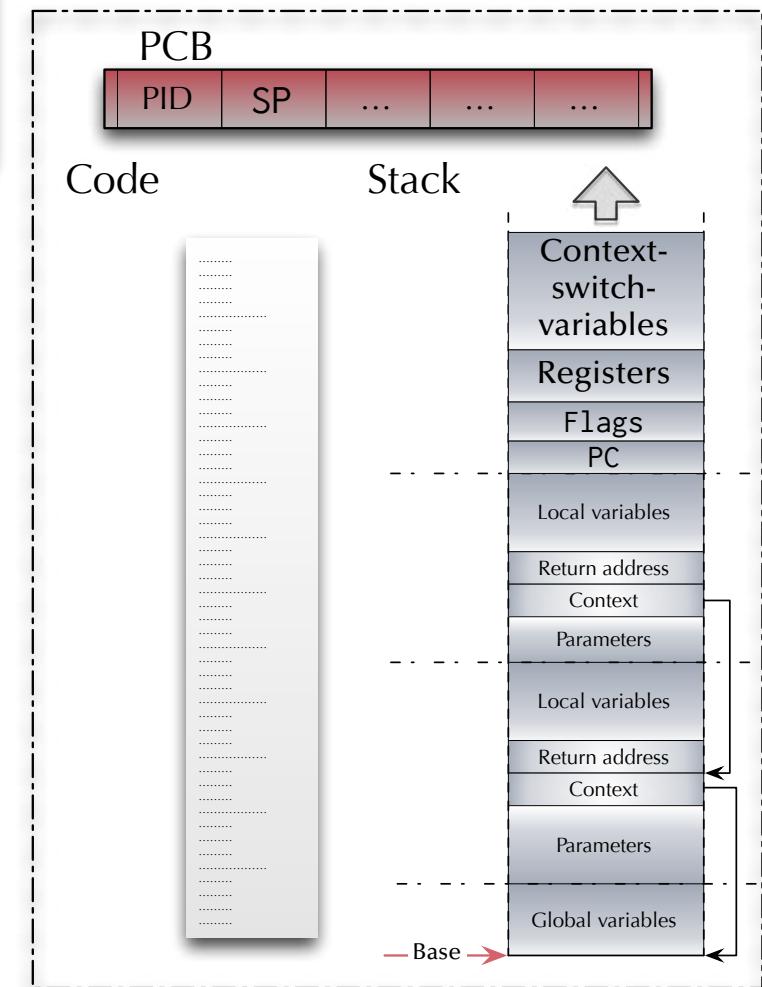
Context switch

Dispatcher

Process 1



Process 2



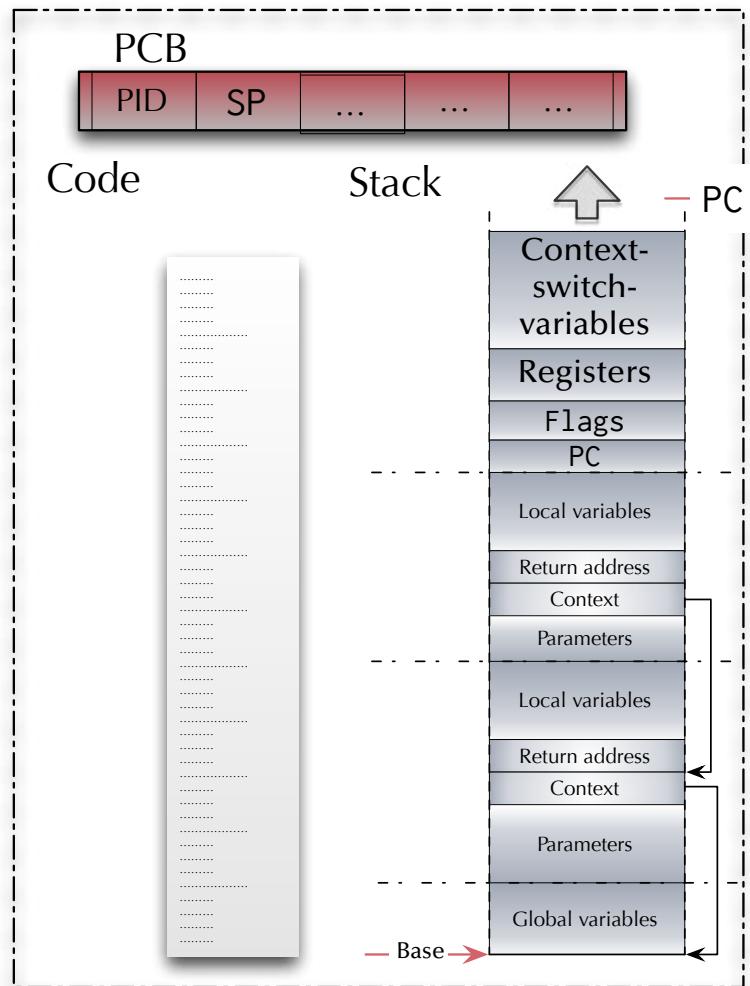


Asynchronism

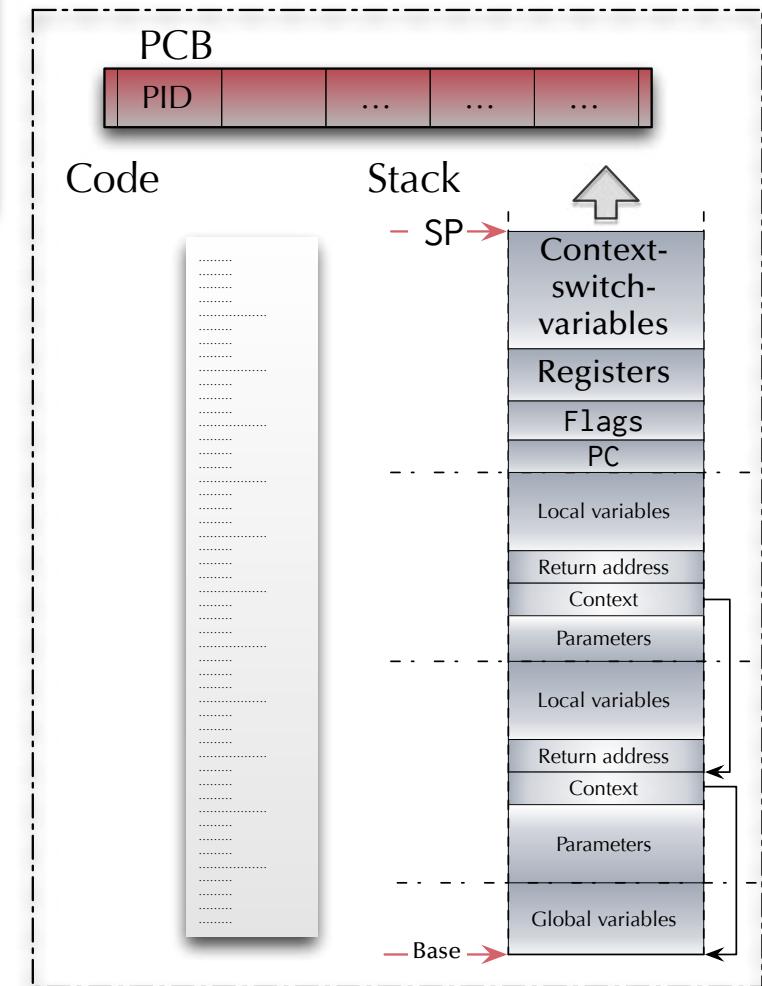
Context switch

Dispatcher

Process 1



Process 2



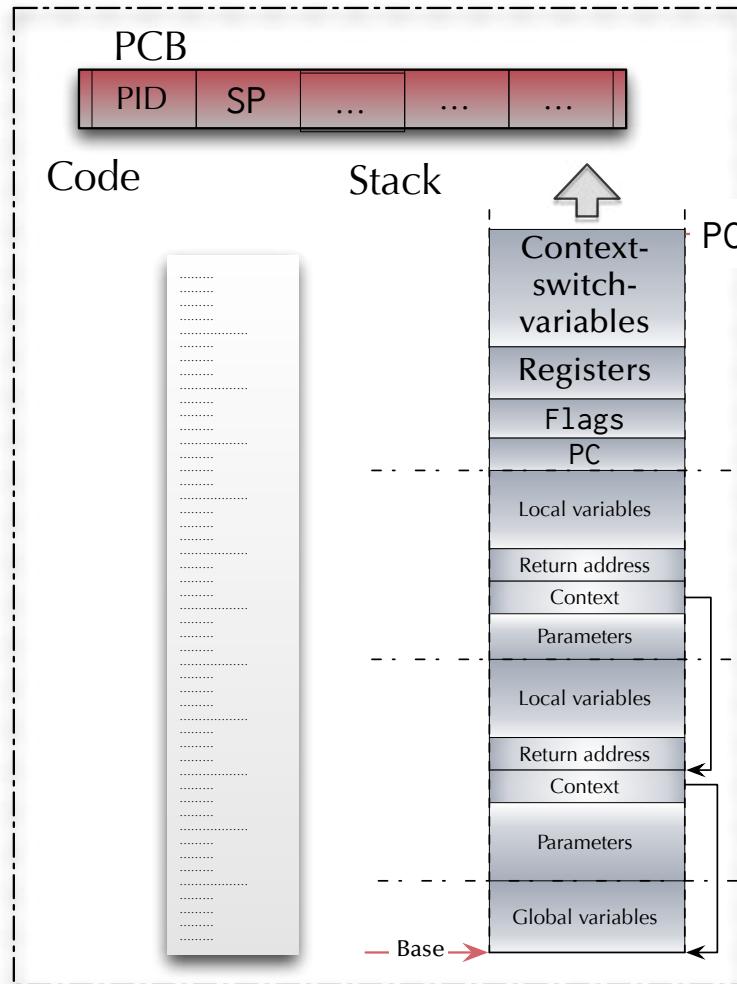


Asynchronism

Context switch

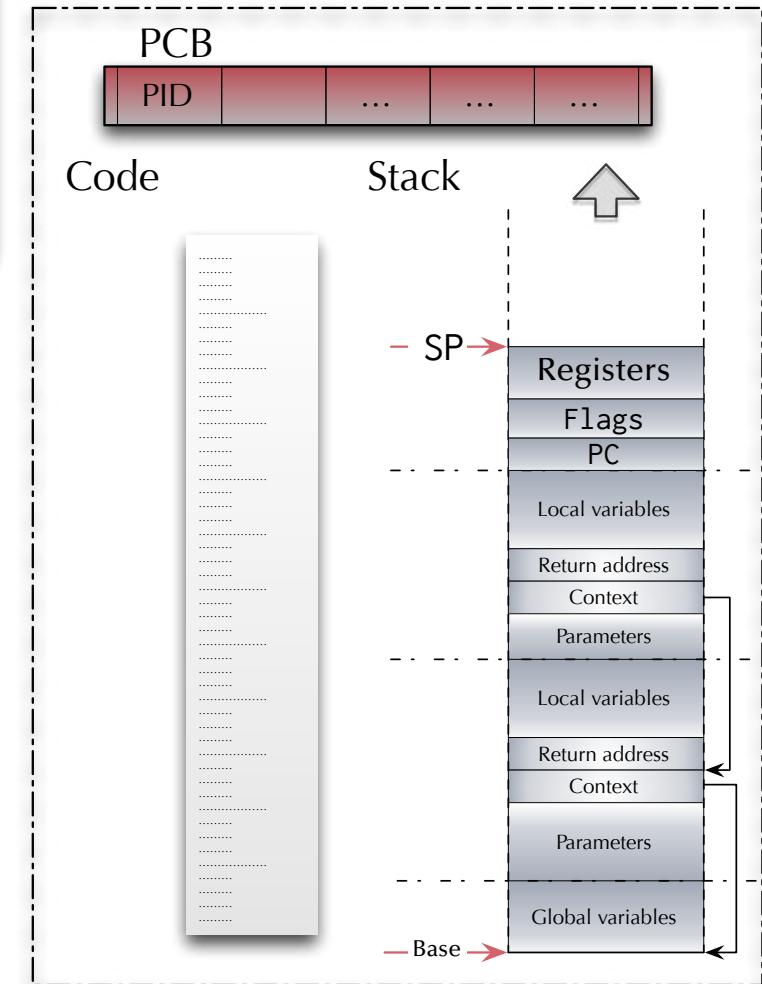
Dispatcher

Process 1



Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables

Process 2



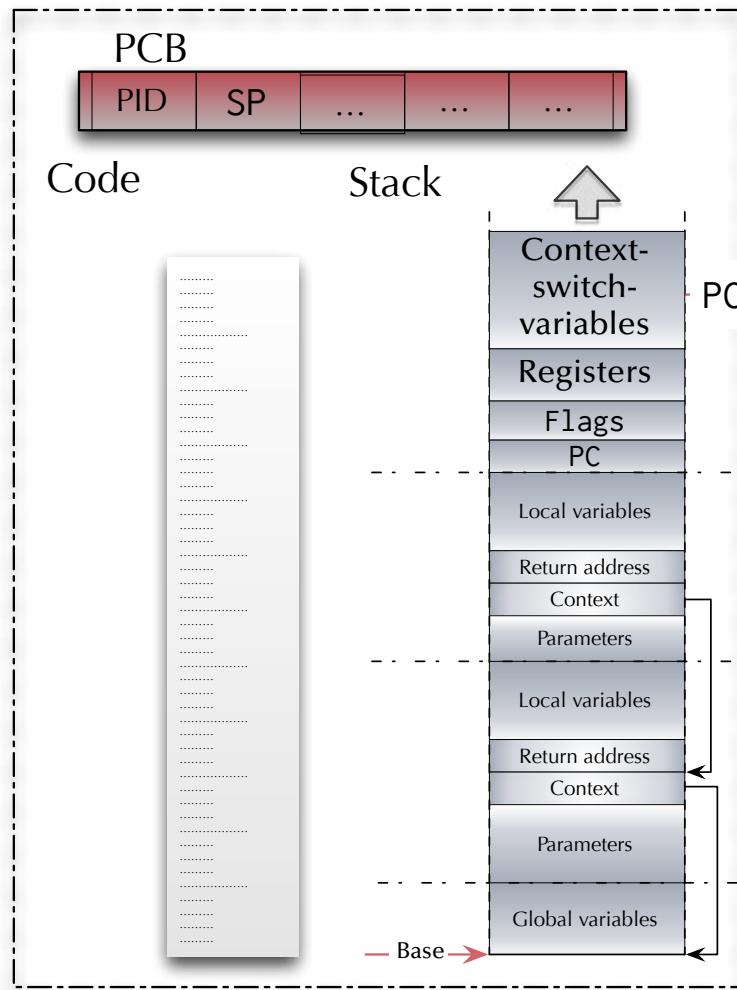


Asynchronism

Context switch

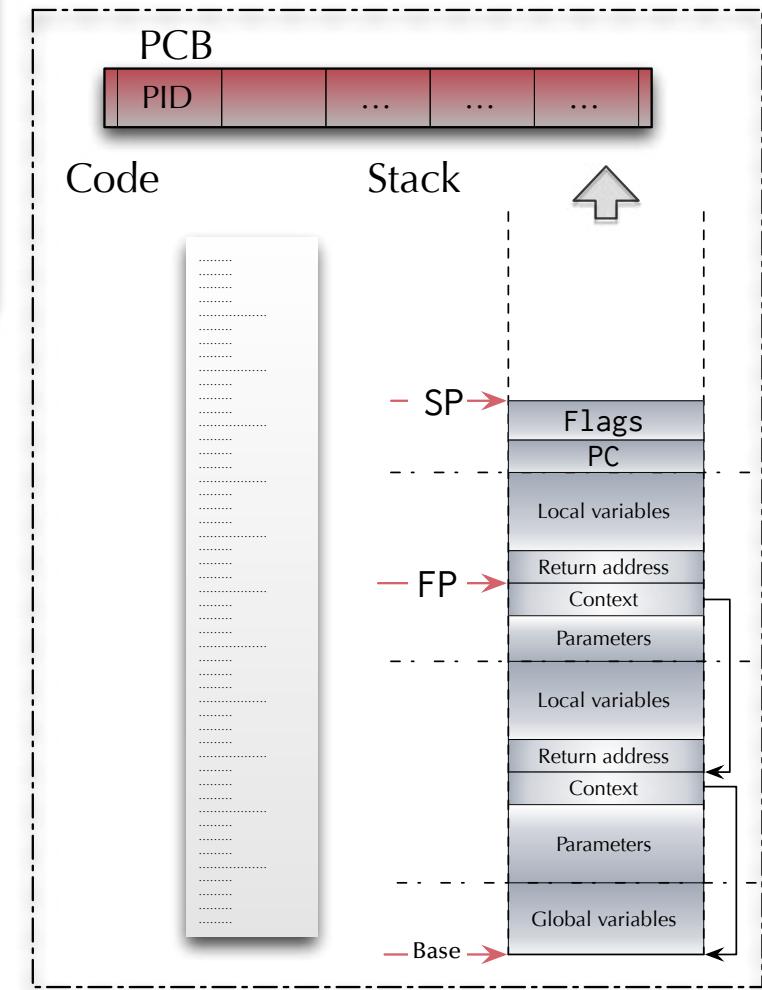
Dispatcher

Process 1



Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables
Pop registers

Process 2



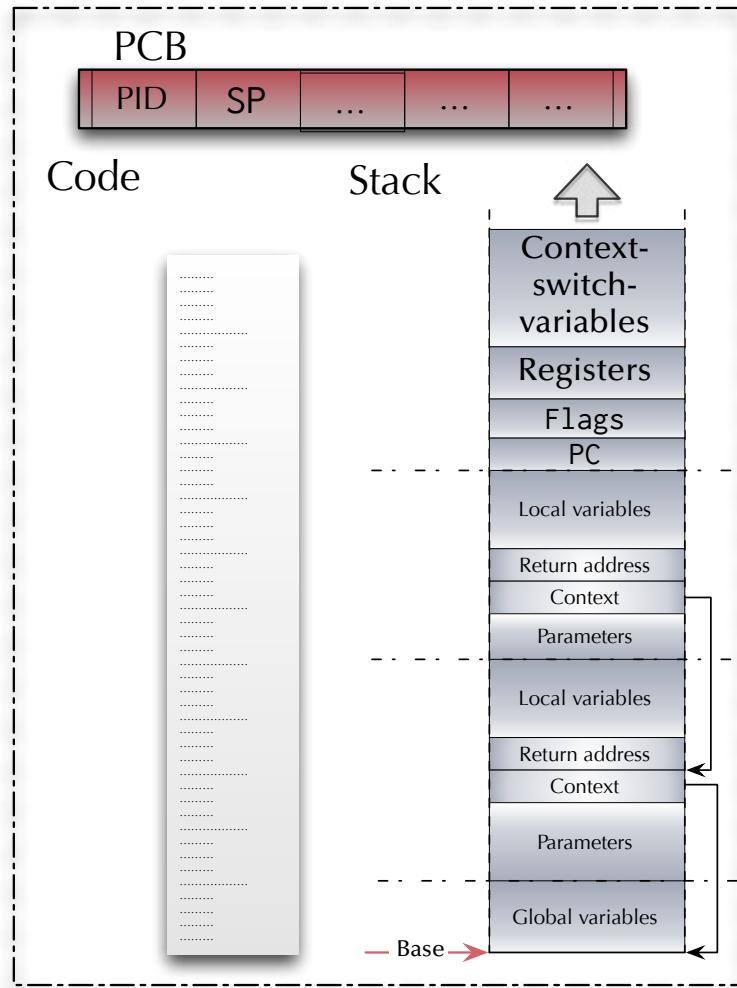


Asynchronism

Context switch

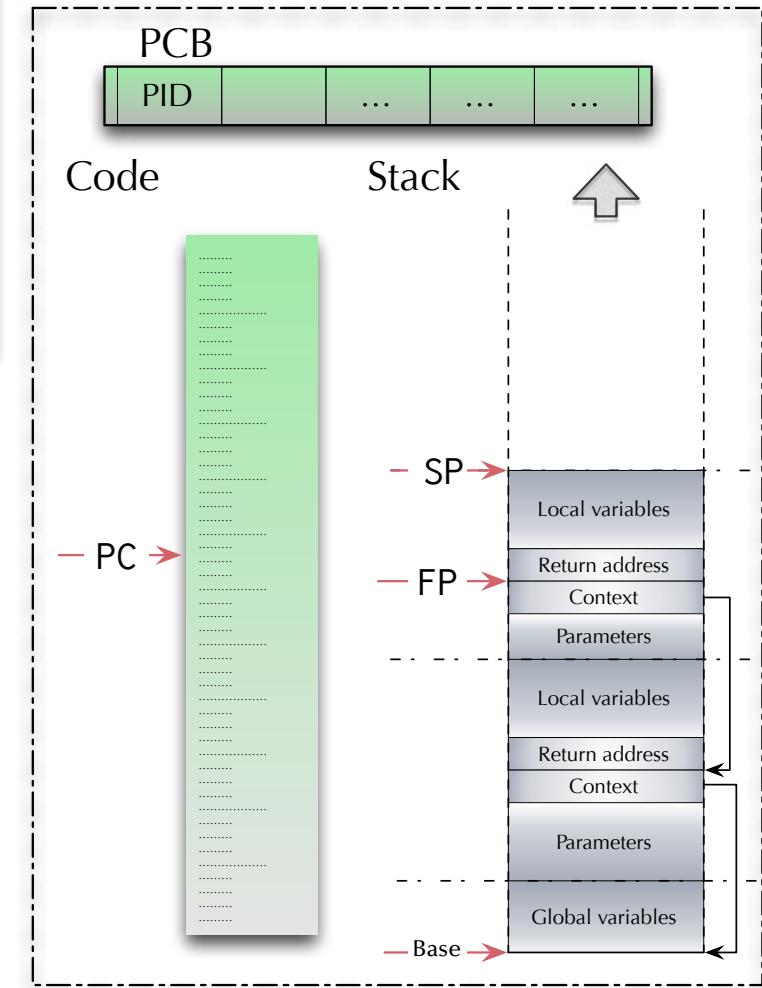
Dispatcher

Process 1



Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables
Pop registers
Return from interrupt

Process 2





Asynchronism

Multi-tasking and Contention

Anything else could go wrong?



Asynchronism

Multi-tasking and Contention

Anything else could go wrong?

- ☞ If there is neither communication nor contention between concurrent parts
... all is easy ... and boring.
- ☞ What happens if concurrent programs **share** data?



Asynchronism

Shared variables

Atomic load & store operations

- ☞ Assumption 1: every individual base memory cell (word) load and store access is *atomic*
- ☞ Assumption 2: there is *no* atomic combined load-store access

G : Natural := 0; -- assumed to be mapped on a 1-word cell in memory

```
task body P1 is
begin
  G := 1
  G := G + G;
end P1;
```

```
task body P2 is
begin
  G := 2
  G := G + G;
end P2;
```

```
task body P3 is
begin
  G := 3
  G := G + G;
end P3;
```

- ☞ What is the value of G?



Asynchronism

Shared variables

Atomic load & store operations

- ☞ Assumption 1: every individual base memory cell (word) load and store access is *atomic*
- ☞ Assumption 2: there is *no* atomic combined load-store access

G: .word 0x00000000

```
ldr r4, =G  
mov r1, #1  
str r1, [r4]  
ldr r2, [r4]  
ldr r3, [r4]  
add r1, r2, r3  
str r1, [r4]
```

```
ldr r4, =G  
mov r1, #2  
str r1, [r4]  
ldr r2, [r4]  
ldr r3, [r4]  
add r1, r2, r3  
str r1, [r4]
```

```
ldr r4, =G  
mov r1, #3  
str r1, [r4]  
ldr r2, [r4]  
ldr r3, [r4]  
add r1, r2, r3  
str r1, [r4]
```

- ☞ What is the value in memory cell G after all three programs complete?



Asynchronism

Shared variables

This is terrible!

Nobody in their right mind would analyse a program like that.

- ☞ ... are we missing something?
- ☞ ... is there an elegant way out?



Asynchronism

Mutual exclusion ... or the lack thereof

```
Count : Integer := 0;
```

```
task body Enter is
begin
  for i := 1 .. 100 loop
    Count := Count + 1;
  end loop;
end Enter;
```

```
task body Leave is
begin
  for i := 1 .. 100 loop
    Count := Count - 1;
  end loop;
end Leave;
```

☞ What is the value of Count after both programs complete?



Asynchronism

Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

<pre>ldr r4, =Count mov r1, #1 for_enter: cmp r1, #100 bgt end_for_enter ldr r2, [r4] add r2, #1 str r2, [r4] add r1, #1 b for_enter end_for_enter:</pre>	<pre>ldr r4, =Count mov r1, #1 for_leave: cmp r1, #100 bgt end_for_leave ldr r2, [r4] sub r2, #1 str r2, [r4] add r1, #1 b for_leave end_for_leave:</pre>
---	---

☞ What is the value at address Count after both programs complete?



Asynchronism

Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

```
ldr    r4, =Count  
mov    r1, #1  
for_enter:  
cmp    r1, #100  
bgt    end_for_enter
```

```
ldr    r2, [r4]  
add    r2, #1  
str    r2, [r4]
```

Critical section

```
ldr    r4, =Count  
mov    r1, #1  
for_leave:  
cmp    r1, #100  
bgt    end_for_leave
```

```
ldr    r2, [r4]  
sub    r2, #1  
str    r2, [r4]
```

Critical section

```
add    r1, #1  
b      for_enter  
end_for_enter:
```

```
add    r1, #1  
b      for_leave  
end_for_leave:
```

☞ What is the value at address Count after both programs complete?



Asynchronism

Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

```
ldr    r4, =Count
mov    r1, #1
for_enter:
    cmp   r1, #100
    bgt   end_for_enter
enter_critical_fail:
    ldrex r2, [r4] ; tag [r4] as exclusive
    add   r2, #1
    strex r0, r2, [r4] ; only if untouched
    cmp   r0, #0
    bne   enter_critical_fail
    add   r1, #1
    b     for_enter
end_for_enter:
```

```
ldr    r4, =Count
mov    r1, #1
for_leave:
    cmp   r1, #100
    bgt   end_for_leave
leave_critical_fail:
    ldrex r2, [r4] ; tag [r4] as exclusive
    sub   r2, #1
    strex r0, r2, [r4] ; only if untouched
    cmp   r0, #0
    bne   leave_critical_fail
    add   r1, #1
    b     for_leave
end_for_leave:
```

☞ What is the value at address Count after both programs complete?



Asynchronism

Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

```
ldr    r4, =Count
      mov    r1, #1
for_enter:
      cmp    r1, #100
      bgt    end_for_enter
enter_critical_fail:
      ldrex r2, [r4] ; tag [r4] as exclusive
      add    r2, #1
      strex r0, r2, [r4] ; only if untouched
      cmp    r0, #0
      bne    enter_critical_fail
      add    r1, #1
      b      for_enter
end_for_enter:
```

Any context switch
needs to clear
reservations

```
ldr    r4, =Count
      mov    r1, #1
for_leave:
      cmp    r1, #100
      bgt    end_for_leave
leave_critical_fail:
      ldrex r2, [r4] ; tag [r4] as exclusive
      sub    r2, #1
      strex r0, r2, [r4] ; only if untouched
      cmp    r0, #0
      bne    leave_critical_fail
      add    r1, #1
      b      for_leave
end_for_leave:
```

☞ What is the value at address Count after both programs complete?

```
Count: .word 0x00000000
```

```
ldr r4, =Count  
mov r1, #1
```

for_enter:

```
cmp r1, #100  
bgt end_for_enter
```

```
ldr r4, =Count  
mov r1, #1
```

for_leave:

```
cmp r1, #100  
bgt end_for_leave
```

Negotiate who goes first

```
ldr r2, [r4]  
add r2, #1  
str r2, [r4]
```

Critical section

```
ldr r2, [r4]  
sub r2, #1  
str r2, [r4]
```

Critical section

Indicate critical section completed

```
add r1, #1  
b for_enter
```

end_for_enter:

```
add r1, #1  
b for_leave
```

end_for_leave:

```
Count: .word 0x00000000
```

```
Lock: .word 0x00000000 ; #0 means unlocked
```

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_enter:

```
cmp r1, #100  
bgt end_for_enter
```

fail_lock_enter:

```
ldr r0, [r3]  
cmp r0, #0  
bne fail_lock_enter ; if locked
```

```
ldr r2, [r4]  
add r2, #1  
str r2, [r4]
```

Critical section

```
add r1, #1  
b for_enter
```

end_for_enter:

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_leave:

```
cmp r1, #100  
bgt end_for_leave
```

fail_lock_leave:

```
ldr r0, [r3]  
cmp r0, #0  
bne fail_lock_leave ; if locked
```

```
ldr r2, [r4]  
sub r2, #1  
str r2, [r4]
```

Critical section

```
add r1, #1  
b for_leave
```

end_for_leave:

```
Count: .word 0x00000000
```

```
Lock: .word 0x00000000 ; #0 means unlocked
```

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_enter:

```
cmp r1, #100  
bgt end_for_enter
```

fail_lock_enter:

```
ldr r0, [r3]  
cmp r0, #0  
bne fail_lock_enter ; if locked  
mov r0, #1           ; lock value  
str r0, [r3]         ; lock
```

```
ldr r2, [r4]  
add r2, #1  
str r2, [r4]
```

Critical section

```
add r1, #1  
b for_enter
```

end_for_enter:

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_leave:

```
cmp r1, #100  
bgt end_for_leave
```

fail_lock_leave:

```
ldr r0, [r3]  
cmp r0, #0  
bne fail_lock_leave ; if locked  
mov r0, #1           ; lock value  
str r0, [r3]         ; lock
```

```
ldr r2, [r4]  
sub r2, #1  
str r2, [r4]
```

Critical section

```
add r1, #1  
b for_leave
```

end_for_leave:

```
Count: .word 0x00000000
```

```
Lock: .word 0x00000000 ; #0 means unlocked
```

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_enter:

```
cmp r1, #100  
bgt end_for_enter
```

Any context switch
needs to clear
reservations

fail_lock_enter:

```
ldrex r0, [r3]  
cmp r0, #0  
bne fail_lock_enter ; if locked  
mov r0, #1 ; lock value  
strex r5, r0, [r3] ; try lock  
cmp r5, #0  
bne fail_lock_enter ; if touched  
dmb ; sync memory
```

```
ldr r2, [r4]  
add r2, #1  
str r2, [r4]
```

Critical section

```
add r1, #1  
b for_enter
```

end_for_enter:

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_leave:

```
cmp r1, #100  
bgt end_for_leave
```

fail_lock_leave:

```
ldrex r0, [r3]  
cmp r0, #0  
bne fail_lock_leave ; if locked  
mov r0, #1 ; lock value  
strex r5, r0, [r3] ; try lock  
cmp r5, #0  
bne fail_lock_leave ; if touched  
dmb ; sync memory
```

```
ldr r2, [r4]  
sub r2, #1  
str r2, [r4]
```

Critical section

```
add r1, #1  
b for_leave
```

end_for_leave:

```
Count: .word 0x00000000
```

```
Lock: .word 0x00000000 ; #0 means unlocked
```

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_enter:

```
cmp r1, #100  
bgt end_for_enter
```

Any context switch
needs to clear
reservations

fail_lock_enter:

```
ldrex r0, [r3]  
cmp r0, #0  
bne fail_lock_enter ; if locked  
mov r0, #1 ; lock value  
strex r5, r0, [r3] ; try lock  
cmp r5, #0  
bne fail_lock_enter ; if touched  
dmb ; sync memory
```

```
ldr r2, [r4]  
add r2, #1  
str r2, [r4]
```

Critical section

```
dmb ; sync memory  
mov r0, #0 ; unlock value  
str r0, [r3] ; unlock
```

```
add r1, #1  
b for_enter
```

end_for_enter:

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_leave:

```
cmp r1, #100  
bgt end_for_leave
```

fail_lock_leave:

```
ldrex r0, [r3]  
cmp r0, #0  
bne fail_lock_leave ; if locked  
mov r0, #1 ; lock value  
strex r5, r0, [r3] ; try lock  
cmp r5, #0  
bne fail_lock_leave ; if touched  
dmb ; sync memory
```

```
ldr r2, [r4]  
sub r2, #1  
str r2, [r4]
```

Critical section

```
dmb ; sync memory  
mov r0, #0 ; unlock value  
str r0, [r3] ; unlock
```

```
add r1, #1  
b for_leave
```

end_for_leave:



Asynchronism

Mutual exclusion: atomic test-and-set operation

```
type Flag is Natural range 0..1; C : Flag := 0;
```

```
task body Pi is
```

```
L : Flag;
```

```
begin
```

```
loop
```

```
loop
```

```
[L := C; C := 1];
```

```
exit when L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_i;
```

```
C := 0;
```

```
end loop;
```

```
end Pi;
```

```
task body Pj is
```

```
L : Flag;
```

```
begin
```

```
loop
```

```
loop
```

```
[L := C; C := 1];
```

```
exit when L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_j;
```

```
C := 0;
```

```
end loop;
```

```
end Pj;
```

☞ Does that work?



Asynchronism

Mutual exclusion: atomic test-and-set operation

```
type Flag is Natural range 0..1; C : Flag := 0;
```

```
task body Pi is
```

```
L : Flag;
```

```
begin
```

```
loop
```

```
loop
```

```
[L := C; C := 1];
```

```
exit when L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_i;
```

```
C := 0;
```

```
end loop;
```

```
end Pi;
```

```
task body Pj is
```

```
L : Flag;
```

```
begin
```

```
loop
```

```
loop
```

```
[L := C; C := 1];
```

```
exit when L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_j;
```

```
C := 0;
```

```
end loop;
```

```
end Pj;
```

☞ Mutual exclusion!, No deadlock!, No global live-lock!

☞ Works for any dynamic number of processes.

☞ Individual starvation possible! Busy waiting loops!



Asynchronism

Mutual exclusion: atomic exchange operation

```
type Flag is Natural range 0..1; C : Flag := 0;
```

```
task body Pi is
```

```
    L : Flag := 1;
```

```
begin
```

```
loop
```

```
loop
```

```
[Temp := L; L := C; C := Temp];
```

```
exit when L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_i;
```

```
L := 1; C := 0;
```

```
end loop;
```

```
end Pi;
```

```
task body Pj is
```

```
    L : Flag := 1;
```

```
begin
```

```
loop
```

```
loop
```

```
[Temp := L; L := C; C := Temp];
```

```
exit when L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_j;
```

```
L := 1; C := 0;
```

```
end loop;
```

```
end Pj;
```

☞ Does that work?



Asynchronism

Mutual exclusion: atomic exchange operation

```
type Flag is Natural range 0..1; C : Flag := 0;
```

```
task body Pi is
```

```
    L : Flag := 1;
```

```
begin
```

```
loop
```

```
loop
```

```
[Temp := L; L := C; C := Temp];
```

```
exit when L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_i;
```

```
L := 1; C := 0;
```

```
end loop;
```

```
end Pi;
```

```
task body Pj is
```

```
    L : Flag := 1;
```

```
begin
```

```
loop
```

```
loop
```

```
[Temp := L; L := C; C := Temp];
```

```
exit when L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_j;
```

```
L := 1; C := 0;
```

```
end loop;
```

```
end Pj;
```

☞ Mutual exclusion!, No deadlock!, No global live-lock!

☞ Works for any dynamic number of processes.

☞ Individual starvation possible! Busy waiting loops!



Asynchronism

Mutual exclusion: memory cell reservation

```
type Flag is Natural range 0..1; C : Flag := 0;
```

```
task body Pi is
```

```
L : Flag;
```

```
begin
```

```
loop
```

```
loop
```

```
L := C; C := 1;
```

```
exit when Untouched and L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_i;
```

```
C := 0;
```

```
end loop;
```

```
end Pi;
```

```
task body Pj is
```

```
L : Flag;
```

```
begin
```

```
loop
```

```
loop
```

```
L := C; C := 1;
```

```
exit when Untouched and L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_j;
```

```
C := 0;
```

```
end loop;
```

```
end Pj;
```

☞ Does that work?



Asynchronism

Mutual exclusion: memory cell reservation

```
type Flag is Natural range 0..1; C : Flag := 0;
```

```
task body Pi is
```

```
L : Flag;
```

```
begin
```

```
loop
```

```
loop
```

```
L := C; C := 1;
```

```
exit when Untouched and L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_i;
```

```
C := 0;
```

```
end loop;
```

```
end Pi;
```

Any context switch
needs to clear
reservations

```
task body Pj is
```

```
L : Flag;
```

```
begin
```

```
loop
```

```
L := C; C := 1;
```

```
exit when Untouched and L = 0;
```

```
----- change process
```

```
end loop;
```

```
----- critical_section_j;
```

```
C := 0;
```

```
end loop;
```

```
end Pj;
```

☞ Mutual exclusion!, No deadlock!, No global live-lock!

☞ Works for any dynamic number of processes.

☞ Individual starvation possible! Busy waiting loops!

```
Count: .word 0x00000000
```

```
Lock: .word 0x00000000 ; #0 means unlocked
```

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_enter:

```
cmp r1, #100  
bgt end_for_enter
```

Any context switch
needs to clear
reservations

fail_lock_enter:

```
ldrex r0, [r3]  
cmp r0, #0  
bne fail_lock_enter ; if locked  
mov r0, #1 ; lock value  
strex r5, r0, [r3] ; try lock  
cmp r5, #0  
bne fail_lock_enter ; if touched  
dmb ; sync memory
```

```
ldr r2, [r4]  
add r2, #1  
str r2, [r4]
```

Critical section

```
dmb ; sync memory  
mov r0, #0 ; unlock value  
str r0, [r3] ; unlock
```

```
add r1, #1  
b for_enter
```

end_for_enter:

```
ldr r3, =Lock  
ldr r4, =Count  
mov r1, #1
```

for_leave:

```
cmp r1, #100  
bgt end_for_leave
```

Asks for permission

fail_lock_leave:

```
ldrex r0, [r3]  
cmp r0, #0  
bne fail_lock_leave ; if locked  
mov r0, #1 ; lock value  
strex r5, r0, [r3] ; try lock  
cmp r5, #0  
bne fail_lock_leave ; if touched  
dmb ; sync memory
```

```
ldr r2, [r4]  
sub r2, #1  
str r2, [r4]
```

Critical section

```
dmb ; sync memory  
mov r0, #0 ; unlock value  
str r0, [r3] ; unlock
```

```
add r1, #1  
b for_leave
```

end_for_leave:



Asynchronism

Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

```
ldr    r4, =Count  
mov    r1, #1  
for_enter:  
    cmp   r1, #100  
    bgt   end_for_enter  
  
enter_critical_fail:  
    ldrex r2, [r4] ; tag [r4] as exclusive  
    add   r2, #1  
    strex r0, r2, [r4] ; only if untouched  
    cmp   r0, #0  
    bne   enter_critical_fail  
    add   r1, #1  
    b     for_enter  
  
end_for_enter:
```

Any context switch
needs to clear
reservations

```
ldr    r4, =Count  
mov    r1, #1  
for_leave:  
    cmp   r1, #100  
    bgt   end_for_leave  
  
leave_critical_fail:  
    ldrex r2, [r4] ; tag [r4] as exclusive  
    sub   r2, #1  
    strex r0, r2, [r4] ; only if untouched  
    cmp   r0, #0  
    bne   leave_critical_fail  
    add   r1, #1  
    b     for_leave  
  
end_for_leave:
```

Asks for forgiveness

☞ What is the value at address Count after both programs complete?



Asynchronism

Beyond atomic hardware operations

Semaphores

Basic definition (Dijkstra 1968)

Assuming the following three conditions on a shared memory cell between processes:

- a set of processes agree on a variable S operating as a flag to indicate synchronization conditions
- an atomic operation P on S — for ‘passeren’ (Dutch for ‘pass’):
 $P(S)$: [as soon as $S > 0$ then $S := S - 1$] ↗ this is a potentially delaying operation
- an atomic operation V on S — for ‘vrygeven’ (Dutch for ‘to release’):
 $V(S)$: [$S := S + 1$]

↗ then the variable S is called a **Semaphore**.



Asynchronism

Beyond atomic hardware operations

Semaphores

... as supplied by operating systems and runtime environments

- a set of processes $P_1 \dots P_N$ agree on a variable S operating as a flag to indicate synchronization conditions
- an atomic operation **Wait** on S : (aka ‘Suspend_Until_True’, ‘sem_wait’, ...)

Process P_i : **Wait** (S):

```
[if  $S > 0$  then  $S := S - 1$ 
   else suspend  $P_i$  on  $S$ ]
```

- an atomic operation **Signal** on S : (aka ‘Set_True’, ‘sem_post’, ...)

Process P_i : **Signal** (S):

```
[if  $\exists P_j$  suspended on  $S$  then release  $P_j$ 
   else  $S := S + 1$ ]
```

☞ then the variable S is called a **Semaphore** in a scheduling environment.



Asynchronism

Beyond atomic hardware operations

Semaphores

Types of semaphores:

- **Binary semaphores:** restricted to [0, 1] or [False, True] resp.
Multiple V (Signal) calls have the same effect than a single call.
 - Atomic hardware operations support binary semaphores.
 - Binary semaphores are sufficient to create all other semaphore forms.
 - **General semaphores** (counting semaphores): non-negative number; (range limited by the system) P and V increment and decrement the semaphore by one.
 - **Quantity semaphores:** The increment (and decrement) value for the semaphore is specified as a parameter with P and V.
- ☞ All types of semaphores must be initialized:
often the number of processes which are allowed inside a critical section, i.e. '1'.

Semaphore: .word 0x00000001

ldr r3, =Semaphore

...

wait (Semaphore)

...

Critical section

...

signal (Semaphore)

...

ldr r3, =Semaphore

...

wait (Semaphore)

...

Critical section

...

signal (Semaphore)

Semaphore: .word 0x00000001

ldr r3, =Semaphore

...

wait_1:

```
ldr r0, [r3]
cmp r0, #0
beq wait_1      ; if Semaphore = 0
sub r0, #1      ; dec Semaphore
str r0, [r3]     ; update
```

...

Critical section

signal (Semaphore)

...

ldr r3, =Semaphore

...

wait_2:

```
ldr r0, [r3]
cmp r0, #0
beq wait_2      ; if Semaphore = 0
sub r0, #1      ; dec Semaphore
str r0, [r3]     ; update
```

...

Critical section

signal (Semaphore)

...

Semaphore: .word 0x00000001

ldr r3, =Semaphore

Any context switch
needs to clear
reservations

wait_1:

```
ldrex r0, [r3]
cmp r0, #0
beq wait_1      ; if Semaphore = 0
sub r0, #1      ; dec Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0
bne wait_1      ; if touched
dmb             ; sync memory
```

...

Critical section

signal (Semaphore)

...

ldr r3, =Semaphore

wait_2:

```
ldrex r0, [r3]
cmp r0, #0
beq wait_2      ; if Semaphore = 0
sub r0, #1      ; dec Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0
bne wait_2      ; if touched
dmb             ; sync memory
```

...

Critical section

signal (Semaphore)

...

Semaphore: .word 0x00000001

ldr r3, =Semaphore

...

wait_1:

```
ldrex r0, [r3]
cmp r0, #0
beq wait_1      ; if Semaphore = 0
sub r0, #1      ; dec Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0
bne wait_1      ; if touched
dmb             ; sync memory
```

...

...

...

Critical section

```
ldr r0, [r3]
add r0, #1      ; inc Semaphore
str r0, [r3]      ; update
```

...

ldr r3, =Semaphore

...

wait_2:

```
ldrex r0, [r3]
cmp r0, #0
beq wait_2      ; if Semaphore = 0
sub r0, #1      ; dec Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0
bne wait_2      ; if touched
dmb             ; sync memory
```

...

...

...

Critical section

```
ldr r0, [r3]
add r0, #1      ; inc Semaphore
str r0, [r3]      ; update
```

...

Semaphore: .word 0x00000001

ldr r3, =Semaphore

...

Any context switch
needs to clear
reservations

wait_1:

```
ldrex r0, [r3]
cmp r0, #0
beq wait_1      ; if Semaphore = 0
sub r0, #1      ; dec Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0
bne wait_1      ; if touched
dmb             ; sync memory
```

...

...

...

Critical section

signal_1:

```
ldrex r0, [r3]
add r0, #1      ; inc Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0
bne signal_1    ; if touched
dmb             ; sync memory
```

...

ldr r3, =Semaphore

...

wait_2:

```
ldrex r0, [r3]
cmp r0, #0
beq wait_2      ; if Semaphore = 0
sub r0, #1      ; dec Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0
bne wait_2      ; if touched
dmb             ; sync memory
```

...

...

...

Critical section

signal_2:

```
ldrex r0, [r3]
add r0, #1      ; inc Semaphore
strex r1, r0, [r3] ; try update
cmp r1, #0
bne signal_2    ; if touched
dmb             ; sync memory
```

...



Asynchronism

Semaphores

```
S : Semaphore := 1;
```

```
task body Pi is
begin
loop
    ----- non_critical_section_i;
    wait (S);
    ----- critical_section_i;
    signal (S);
end loop;
end Pi;
```

```
task body Pj is
begin
loop
    ----- non_critical_section_j;
    wait (S);
    ----- critical_section_j;
    signal (S);
end loop;
end Pj;
```

☞ Works?



Asynchronism

Semaphores

```
S : Semaphore := 1;
```

```
task body Pi is
begin
loop
    ----- non_critical_section_i;
    wait (S);
    ----- critical_section_i;
    signal (S);
end loop;
end Pi;
```

```
task body Pj is
begin
loop
    ----- non_critical_section_j;
    wait (S);
    ----- critical_section_j;
    signal (S);
end loop;
end Pj;
```

- ☞ Mutual exclusion!, No deadlock!, No global live-lock!
- ☞ Works for any dynamic number of processes
- ☞ Individual starvation possible!



Asynchronism

Semaphores

```
S1, S2 : Semaphore := 1;
```

```
task body Pi is
begin
loop
    ----- non_critical_section_i;
    wait (S1);
    wait (S2);
    ----- critical_section_i;
    signal (S2);
    signal (S1);
end loop;
end Pi;
```

```
task body Pj is
begin
loop
    ----- non_critical_section_j;
    wait (S2);
    wait (S1);
    ----- critical_section_j;
    signal (S1);
    signal (S2);
end loop;
end Pj;
```

☞ Works too?



Asynchronism

Semaphores

```
S1, S2 : Semaphore := 1;
```

```
task body Pi is
begin
loop
    ----- non_critical_section_i;
    wait (S1);
    wait (S2);
    ----- critical_section_i;
    signal (S2);
    signal (S1);
end loop;
end Pi;
```

```
task body Pj is
begin
loop
    ----- non_critical_section_j;
    wait (S2);
    wait (S1);
    ----- critical_section_j;
    signal (S1);
    signal (S2);
end loop;
end Pj;
```

- ☞ Mutual exclusion!, No global live-lock!
- ☞ Works for any dynamic number of processes.
- ☞ Individual starvation possible!
- ☞ Deadlock possible!



Asynchronism

Semaphores

```
S1, S2 : Semaphore := 1;
```

```
task body Pi is
begin
loop
    ----- non_critical_section_i;
    wait (S1);
    wait (S2);
    ----- critical_section_i;
    signal (S2);
    signal (S1);
end loop;
end Pi;
```

- ☞ Mutual exclusion!, No global live-lock!
- ☞ Works for any dynamic number of processes.
- ☞ Individual starvation possible!
- ☞ Deadlock possible!

```
task body Pj is
begin
loop
    ----- non_critical_section_j;
    wait (S2);
    wait (S1);
    ----- critical_section_j;
    signal (S1);
    signal (S2);
end loop;
end Pj;
```

Concurrent programming languages offer **higher abstraction** and **safer synchronization mechanisms**.



Asynchronism

Summary

Aynchronism

- **Interrupts & Exceptions**

- Concept
- Hardware/Software interaction
- Recursive interrupts

- **Concurrency & Synchronization**

- Race conditions
- Synchronization
- Passing data

Computer Organisation & Program Execution 2021



6

Control Structures

Uwe R. Zimmer - The Australian National University



Control Structures

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

Chapter 2 “Instructions: Language of the Computer”

ARM edition, Morgan Kaufmann 2017



Control Structures

Imperative Programming

Essential control structures for all imperative programming languages are:

- Conditionals: **if**, **case**, **switch**, ...
- Open Loops: **while**, **repeat**, ...
- Bound Loops: **for**, **foreach**, **forall**, ...
- Procedures and Functions (already covered)

☞ How do we create those basic control structures in Assembly?

Functional programming languages are based on functions, but also on conditional expressions.

☞ How do those control structures in programming languages translate into Assembly?



Control Structures

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then  
    Register_3 := 1;  
else  
    Register_3 := 0;  
end if;
```

... same structure? ...
many syntax versions?

```
if (register1 == register2) {  
    register3 = 1;  
} else {  
    register3 = 0;  
}
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of  
    True  -> 1  
    False -> 0
```

```
if register1 == register2:  
    register3 = 1  
else:  
    register3 = 0
```



Control Structures

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then  
    Register_3 := 1;  
else  
    Register_3 := 0;  
end if;
```

```
if (register1 == register2) {  
    register3 = 1;  
} else {  
    register3 = 0;  
}
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of  
    True  -> 1  
    False -> 0
```

```
if register1 == register2:  
    register3 = 1  
else:  
    register3 = 0
```

1. an expression (if)
2. a boolean condition (if)
3. code for True (then)
4. code for False (else)

How do either of those look in assembly?



Control Structures

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
cmp    r1, r2      ; 1. Instructions to generate status flags
beq    then        ; 2. Branch depending on the status flags
mov    r3, #0      ; 4. Instructions for the else branch
b     end_if
```

then:

```
    mov    r3, #1    ; 3. Instructions for the then branch
```

end_if:

It seems there are *three distinguishable code sections* and *one status flag condition*.

☞ Can we form a general pattern for this?



Control Structures

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
.macro if condition_code condition then_code else_code
\condition_code
b\condition then
\else_code
b      end_if

then:
\then_code

end_if:
.endm
```



Control Structures

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
.macro if condition_code condition then_code else_code  
\\condition_code  
b\\condition then  
\\else_code  
b      end_if  
  
then:  
\\then_code  
  
end_if:  
.endm
```

We might need a lot of those,
hence the labels need to be
unique to each if-else block.



Control Structures

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
.macro if condition_code condition then_code else_code
\condition_code
b\condition then@
\else_code
b      end_if@

then@:
\then_code

end_if@:
.endm
```



Control Structures

Conditionals – IF-ELSE

Assuming the values have already been transferred from memory into registers:

```
.macro if condition_code condition then_code else_code
\condition_code
b\condition then\@
\else_code
b    end_if\@

then\@:
\then_code

end_if\@:
.endm
```

We can now write:

```
if “cmp r1, r2”, eq, “mov r3, #1”, “mov r3, #0”
```

... in the general case (with lots of code in each part)

we could create macros for the individual sections as well, so we can e.g. write:

```
if compare_r1_r2, eq, load_1_to_r3, load_0_to_r3
```



Control Structures

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then
    Register_3 := 1;
else
    Register_3 := 0;
end if;
```

```
if (register1 == register2) {
    register3 = 1;
} else {
    register3 = 0;
}
```

```
.macro if condition_code condition then else
    \condition_code
    b\condition then
    \else
    b    end_if
then:
    \then
end_if:
    .endm
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of
    True  -> 1
    False -> 0
```

```
if register1 == register2:
    register3 = 1
else:
    register3 = 0
```



Control Structures

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then  
    Register_3 := 1;  
else  
    Register_3 := 0;  
end if;
```

```
if (register1 == register2) {  
    register3 = 1;  
} else {  
    register3 = 0;  
}
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of  
    True  -> 1  
    False -> 0
```

```
if register1 == register2:  
    register3 = 1  
else:  
    register3 = 0
```

if “`cmp r1, r2`”, eq, “`mov r3, #1`”, “`mov r3, #0`”



Control Structures

Conditionals – IF-ELSE

```
if Register_1 = Register_2 then
    Register_3 := 1;
else
    Register_3 := 0;
end if;
```

```
if (register1 == register2) {
    register3 = 1;
} else {
    register3 = 0;
}
```

```
Register_3 := (if Register_1 = Register_2 then 1 else 0);
```

```
register_3 register_1 register_2 = case register_1 == register_2 of
    True  -> 1
    False -> 0
```

```
if register1 == register2:
    register3 = 1
else:
    register3 = 0
```

```
cmp   r1, r2
beq  then
mov  r3, #0
b    end_if
```

then:
 mov r3, #1
end_if:

Computational complexity: $\Theta(1)$



Control Structures

Loops – FOR

```
for Register_1 in 1..100 loop  
    Register_3 := Register_3 + Register_1;  
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {  
    register3 += register1;  
}
```

```
for register1 in range (1, 101):  
    register3 += register1
```

```
for Register_1 := 1 to 100 do  
    Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100  
    Register_3 = Register_3 + Register_1  
end do
```

```
for Register_1 in 1..100 do  
    Register_3 += Register_1;
```

What are the components?



Control Structures

Loops – FOR

```
for Register_1 in 1..100 loop
```

```
    Register_3 := Register_3 + Register_1;
```

```
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
```

```
    register3 += register1;
```

```
}
```

```
for register1 in range (1, 101):
```

```
    register3 += register1
```

```
for Register_1 := 1 to 100 do
```

```
    Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100
```

```
    Register_3 = Register_3 + Register_1
```

```
end do
```

```
for Register_1 in 1..100 do
```

```
    Register_3 += Register_1;
```

1. an **index**
2. a **start value**
3. an **end value**
4. **code inside loop**

How do either of those look in assembly?



Control Structures

Loops – FOR

Assuming the values have already been transferred from memory into registers:

```
mov r1, #1 ; set index to start value
```

for:

```
cmp r1, #100 ; check whether it went beyond its end value
bgt end_for ; if so, stop the loop
add r3, r1 ; do the work
add r1, #1 ; increment the index
b for
```

end_for:

We can find the index, the start and end values and the body code.

👉 Can we form a general pattern for this?



Control Structures

Loops – FOR

Assuming the values have already been transferred from memory into registers:

```
.macro for register, from, to, body
    mov \register, #\from
    for@:
        cmp \register, #\to
        bgt end_for@
        \body
        add \register, #1
        b for@
end_for@:
.endm
```



Control Structures

Loops – FOR

Assuming the values have already been transferred from memory into registers:

```
.macro for register, from, to, body
    mov \register, #\from

    for@\@:
        cmp \register, #\to
        bgt end_for@\@

        \body

        add \register, #1
        b for@\@

    end_for@\@:
.endm
```

We can now write:

```
for r1, 1, 100 “add r3, r1”
```

... in the general case (with lots of code inside the loop or multiple loops):

```
for r1, 1, 100, loop_body
for r1, 1, 100, “for r2, 1, 100, loop_body”
```



Control Structures

Loops – FOR

```
for Register_1 in 1..100 loop
    Register_3 := Register_3 + Register_1;
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
    register3 += register1;
}
```

```
for register1 in range (1, 101):
    register3 += register1
```

```
for Register_1 := 1 to 100 do
    Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100
    Register_3 = Register_3 + Register_1
end do
```

```
for Register_1 in 1..100 do
    Register_3 += Register_1;
```

```
.macro for register, from, to, body
    mov \register, #\from
    for@:
        cmp \register, #\to
        bgt end_for@
        \body
        add \register, #1
        b for@
.end_for@:
.endm
```



Control Structures

Loops – FOR

```
for Register_1 in 1..100 loop
    Register_3 := Register_3 + Register_1;
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
    register3 += register1;
}
```

```
for register1 in range (1, 101):
    register3 += register1
```

```
for Register_1 := 1 to 100 do
    Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100
    Register_3 = Register_3 + Register_1
end do
```

```
for Register_1 in 1..100 do
    Register_3 += Register_1;
```

for r1, 1, 100 “add r3, r1”



Control Structures

Loops – FOR

```
for Register_1 in 1..100 loop
```

```
    Register_3 := Register_3 + Register_1;
```

```
end loop;
```

```
for (register1 = 1; register1 <= 100; register1++) {
```

```
    register3 += register1;
```

```
}
```

```
for register1 in range (1, 101):
```

```
    register3 += register1
```

```
for Register_1 := 1 to 100 do
```

```
    Register_3 := Register_3 + Register_1;
```

```
do Register_1 = 1, 100
```

```
    Register_3 = Register_3 + Register_1
```

```
end do
```

```
for Register_1 in 1..100 do
```

```
    Register_3 += Register_1;
```

```
mov r1, #1  
for:  
    cmp r1, #100  
    bgt end_for  
    add r3, r1  
    add r1, #1  
    b for  
end_for:
```

Computational complexity: $\Theta(n)$



Control Structures

Loops – WHILE

```
while Register_1 < 100 loop  
    Register_1 := Register_1 ** 2;  
end loop;
```

```
while (register1 < 100) {  
    register1 = register1 * register1;  
}
```

```
while register1 < 100:  
    register1 = register1 ** 2
```

```
while Register_1 < 100 do  
    Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do  
    Register_1 = Register_1 ** 2  
enddo
```

```
while (Register_1 < 100) {  
    Register_1 = Register_1 ** 2;  
}
```

What are the components?



Control Structures

Loops – WHILE

```
while Register_1 < 100 loop  
    Register_1 := Register_1 ** 2;  
end loop;
```

```
while (register1 < 100) {  
    register1 = register1 * register1;  
}
```

```
while register1 < 100:  
    register1 = register1 ** 2
```

```
while Register_1 < 100 do  
    Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do  
    Register_1 = Register_1 ** 2  
enddo
```

```
while (Register_1 < 100) {  
    Register_1 = Register_1 ** 2;  
}
```

1. an expression (if)
2. a boolean condition (if)
3. code inside the loop



Control Structures

Loops – WHILE

```
b      while_condition  
while:  
  mul   r1, r1           ; 3. Loop body  
while_condition:  
  cmp   r1, #100         ; 1. Instructions to generate status flags  
  blt   while            ; 2. Branch depending on the status flags
```

👉 Can we form a general pattern for this?



Control Structures

Loops – WHILE

```
.macro while while_expression, while_condition, body
b    while_condition@

while@:
\body

while_condition@:
\while_expression
b\while_condition while@
.endm
```

We can now write:

```
while “cmp r1, #100”, lt, “mul r1, r1”
```

... try to re-write our power functions from the previous chapter with the macros you have now.



Control Structures

Loops – WHILE

```
while Register_1 < 100 loop
    Register_1 := Register_1 ** 2;
end loop;
```

```
while (register1 < 100) {
    register1 = register1 * register1;
}
```

```
while register1 < 100:
    register1 = register1 ** 2
```

```
while Register_1 < 100 do
    Register_1 := Register_1 **
```

```
while Register_1 < 100 do
    Register_1 = Register_1 **
enddo
```

```
while (Register_1 < 100) {
    Register_1 = Register_1 ** 2;
}
```

```
.macro while while_expression, while_condition, body
    b      while_condition@
        while@:
            2; \body
        while_condition@:
            \while_expression
            b\while_condition while@
        .endm
```



Control Structures

Loops – WHILE

```
while Register_1 < 100 loop
    Register_1 := Register_1 ** 2;
end loop;
```

```
while (register1 < 100) {
    register1 = register1 * register1;
}
```

```
while register1 < 100:
    register1 = register1 ** 2
```

```
while Register_1 < 100 do
    Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do
    Register_1 = Register_1 ** 2
enddo
```

```
while (Register_1 < 100) {
    Register_1 = Register_1 ** 2;
}
```

```
while “cmp r1, #100”, lt, “mul r1, r1”
```



Control Structures

Loops – WHILE

```
while Register_1 < 100 loop
    Register_1 := Register_1 ** 2;
end loop;
```

```
while (register1 < 100) {
    register1 = register1 * register1;
}
```

```
while register1 < 100:
    register1 = register1 ** 2
```

```
while Register_1 < 100 do
    Register_1 := Register_1 ** 2;
```

```
while Register_1 < 100 do
    Register_1 = Register_1 ** 2
enddo
```

```
while (Register_1 < 100) {
    Register_1 = Register_1 ** 2;
}
```

```
b      while_condition
while:
    mul   r1, r1
while_condition:
    cmp   r1, #100
    blt   while
```

Computational complexity: *Undefined*



Control Structures

Conditionals – CASE (indexed)

```
type Colour is (Red, Green, Blue);
```

These values can be represented by (which is also the default in most systems)

```
for Colour use (
    Red    => 0,
    Green  => 1,
    Blue   => 2);
```

Assuming that Register_1 is associated with this type, we can then expect a highly efficient implementation of a case construct such as:

```
case Register_1 is
    when Red    => Register_2 := Register_3;
    when Green  => Register_2 := Register_4;
    when Blue   => Register_2 := Register_5;
end case;
```



Control Structures

Conditionals – CASE (indexed)

A table based branching implementation of:

```
case Register_1 is
    when Red    => Register_3 := Register_2;
    when Green  => Register_4 := Register_2;
    when Blue   => Register_5 := Register_2;
end case;
```

would look like:



Control Structures

Conditionals – CASE (indexed)

tbh [PC, r1, lsl #1] ; PC used as base of branch table, r1 is index

branch_table:

.hword (case_red - branch_table)/2 ; case_red 16 bit offset

.hword (case_green - branch_table)/2 ; case_green 16 bit offset

.hword (case_blue - branch_table)/2 ; case_blue 16 bit offset

case_red:

mov r3, r2 ; Code for case Red

b end_case

case_green:

mov r4, r2 ; Code for case Green

b end_case

case_blue:

mov r5, r2 ; Code for case Blue

end_case:

The complexity of this operation is $\Theta(1)$, e.g. it is independent of the number of cases!

Can we generate this via a macro automatically in one line?, for instance as:

indexed_case r1, “**mov** r3, r2”, “**mov** r4, r2”, “**mov** r5, r2”



Control Structures

Conditionals – CASE (indexed)

.. yes, but as the number of cases is variable, we need to write this recursively:

```
.macro indexed_case index case_body other_cases:vararg
indexed_case_id \@, \index, “\case_body”, \other_cases ; add a unique id
.endm
```

```
.macro indexed_case_id id index case_body other_cases:vararg
tbh [pc, \index, lsl #1]
```

branch_table_\id:

```
table_entry \id, i, “\case_body”, \other_cases ; build up the table entries
case_entry \id, i, “\case_body”, \other_cases ; add the codes with a label each
```

indexed_case_end_\id:

```
.endm
```

...

The parts which are actually producing code are **highlighted**.

... recursive parts are following on the next page ... hold on to something!



Control Structures

Conditionals – CASE (indexed)

...

```
.macro table_entry id case_nr case_body other_cases:vararg
.hword  (case_\id\()_\\case_nr - branch_table_\id)/2
.ifnb \other_cases
table_entry \id, \case_nr\()i, \other_cases      ; still more entries to add
.endif
.endm

.macro case_entry id case_nr case_body other_cases:vararg
case_\id\()_\\case_nr:
\case_body
b    indexed_case_end_\id
.ifnb \other_cases
case_entry \id, \case_nr\()i, \other_cases      ; still more entries to add
.endif
.endm
```

... yes, this is a bit more involved than the previous macros, yet it is here to demonstrate that more complex and dynamic structures can also be macro generated.



Control Structures

Conditionals – CASE (indexed)

```
case Register_1 is
    when Red    => Register_3 := Register_2;
    when Green  => Register_4 := Register_2;
    when Blue   => Register_5 := Register_2;
end case;
```

indexed_case r1, “**mov r3, r2**”, “**mov r4, r2**”, “**mov r5, r2**”



Control Structures

Conditionals – CASE (indexed)

```
case Register_1 is
    when Red    => Register_3 := Register_2;
    when Green  => Register_4 := Register_2;
    when Blue   => Register_5 := Register_2;
end case;
```

Computational complexity: $\Theta(1)$

Side remark: if you disassemble such a structure, it will look different.
☞ How and why?

```
tbh      [PC, r1, lsl #1]
branch_table:
    .hword (case_red - branch_table)/2
    .hword (case_green - branch_table)/2
    .hword (case_blue - branch_table)/2
case_red:
    mov    r3, r2
    b     end_case
case_green:
    mov    r4, r2
    b     end_case
case_Blue:
    mov    r5, r2
end_case:
```



Control Structures

Conditionals – CASE (guarded expressions, list of conditions)

```
r0 ::= Int -> Int -> Int -> Int  
r0 r1 r2 r3  
| r1 < r2 = r1  
| r1 > r2 = r2  
| r1 == r2 = 0  
| otherwise = error "How did I get here?"
```

```
switch (r1) {  
    case 4 : r0 = r1;  
        break;  
    case 5 : r0 = r2;  
        break;  
    case 6 : r0 = 0;  
}
```

```
r0 := (if r1 < r2 then r1  
       elsif r1 > r2 then r2  
       elsif r1 = r2 then 0  
       else Integer'Invalid);
```

... similar structure? ...
many syntax versions?



Control Structures

Conditionals – CASE (guarded expressions, list of conditions)

```
r0 :: Int -> Int -> Int -> Int
```

```
r0 r1 r2 r3
```

```
| r1 < r2 = r1  
| r1 > r2 = r2  
| r1 == r2 = 0  
| otherwise = error "How did I get here?"
```

```
switch (r1) {  
    case 4 : r0 = r1;  
    break;  
    case 5 : r0 = r2;  
    break;  
    case 6 : r0 = 0;  
}
```

```
r0 := (if r1 < r2 then r1  
        elsif r1 > r2 then r2  
        elsif r1 = r2 then 0  
        else Integer'Invalid);
```

1. guards
2. guard conditions
3. guard expressions / statements



Control Structures

Conditionals – CASE (guarded expressions, list of conditions)

```
cmp    r1, r2
blt    case_a
cmp    r1, r2
bgt    case_b
cmp    r1, r2
beq    case_c
b      end_case

case_a:
mov    r0, r1
b      end_case

case_b:
mov    r0, r2
b      end_case

case_c:
mov    r0, #0
b      end_case

end_case:
```

1. guards
2. guard conditions
3. guard expressions / statements

Generated by:

```
case “cmp r1, r2”, lt, “mov r0, r1”,
“cmp r1, r2”, gt, “mov r0, r2”,
“cmp r1, r2”, gt, “mov r0, #0”
```



Control Structures

Conditionals – CASE (indexed)

This is again recursive to handle the variable number of cases:

```
.macro case expression condition case_body other_cases:vararg
case_id \@, “\expression”, \condition, “\case_body”, \other_cases
.endm

.macro case_id id expression condition case_body other_cases:vararg
guards_rec \id, i, “\expression”, \condition, “\case_body”, \other_cases
cases_rec \id, i, “\expression”, \condition, “\case_body”, \other_cases
end_case_\id:
.endm

...
```

The parts which are actually producing code are **highlighted**.

... and we still need to generate the list of guards, followed by the list of code sections.



Control Structures

Conditionals – CASE (indexed)

...

```
.macro guards_rec id case_nr expression condition case_body other_cases:vararg
\expression
b\condition case_\id\()_case_nr
.ifnb \other_cases
guards_rec \id, \case_nr\()i, \other_cases
.else
b    end_case_\id
.endif
.endm

.macro cases_rec id case_nr expression condition case_body other_cases:vararg
case_\id\()_case_nr:
\case_body
b    end_case_\id
.ifnb \other_cases
cases_rec \id, \case_nr\()i, \other_cases
.endif
.endm
```

Keep in mind:
Macro programming is pure
textual replacement.
The result is a text which is then translat-
ed by the assembler into machine code.



Control Structures

Conditionals – CASE (guarded expressions, list of conditions)

```
r0 :: Int -> Int -> Int -> Int
```

```
r0 r1 r2 r3
```

```
| r1 < r2 = r1  
| r1 > r2 = r2  
| r1 == r2 = 0  
| otherwise = error "How did I get here?"
```

```
switch (r1) {  
    case 4 : r0 = r1;  
    break;  
    case 5 : r0 = r2;  
    break;  
    case 6 : r0 = 0;  
}
```

```
r0 := (if r1 < r2 then r1  
        elseif r1 > r2 then r2  
        elseif r1 = r2 then 0  
        else Integer'Invalid);
```

```
case "cmp r1, r2", lt, "mov r0, r1",  
      "cmp r1, r2", gt, "mov r0, r2",  
      "cmp r1, r2", gt, "mov r0, #0"
```



Control Structures

Conditionals – CASE (guarded expressions, list of conditions)

```
r0 ::= Int -> Int -> Int -> Int
```

```
r0 r1 r2 r3
```

```
| r1 < r2 = r1  
| r1 > r2 = r2  
| r1 == r2 = 0  
| otherwise = error "How did I get here?"
```

```
switch (r1) {  
    case 4 : r0 = r1;  
    break;  
    case 5 : r0 = r2;  
    break;  
    case 6 : r0 = 0;  
}
```

```
r0 := (if r1 < r2 then r1  
        elseif r1 > r2 then r2  
        elseif r1 = r2 then 0  
        else Integer'Invalid);
```

```
cmp   r1, r2  
blt  case_a  
cmp   r1, r2  
bgt  case_b  
cmp   r1, r2  
beq  case_c  
b    end_case
```

case_a:

```
mov  r0, r1  
b    end_case
```

case_b:

```
mov  r0, r2  
b    end_case
```

case_c:

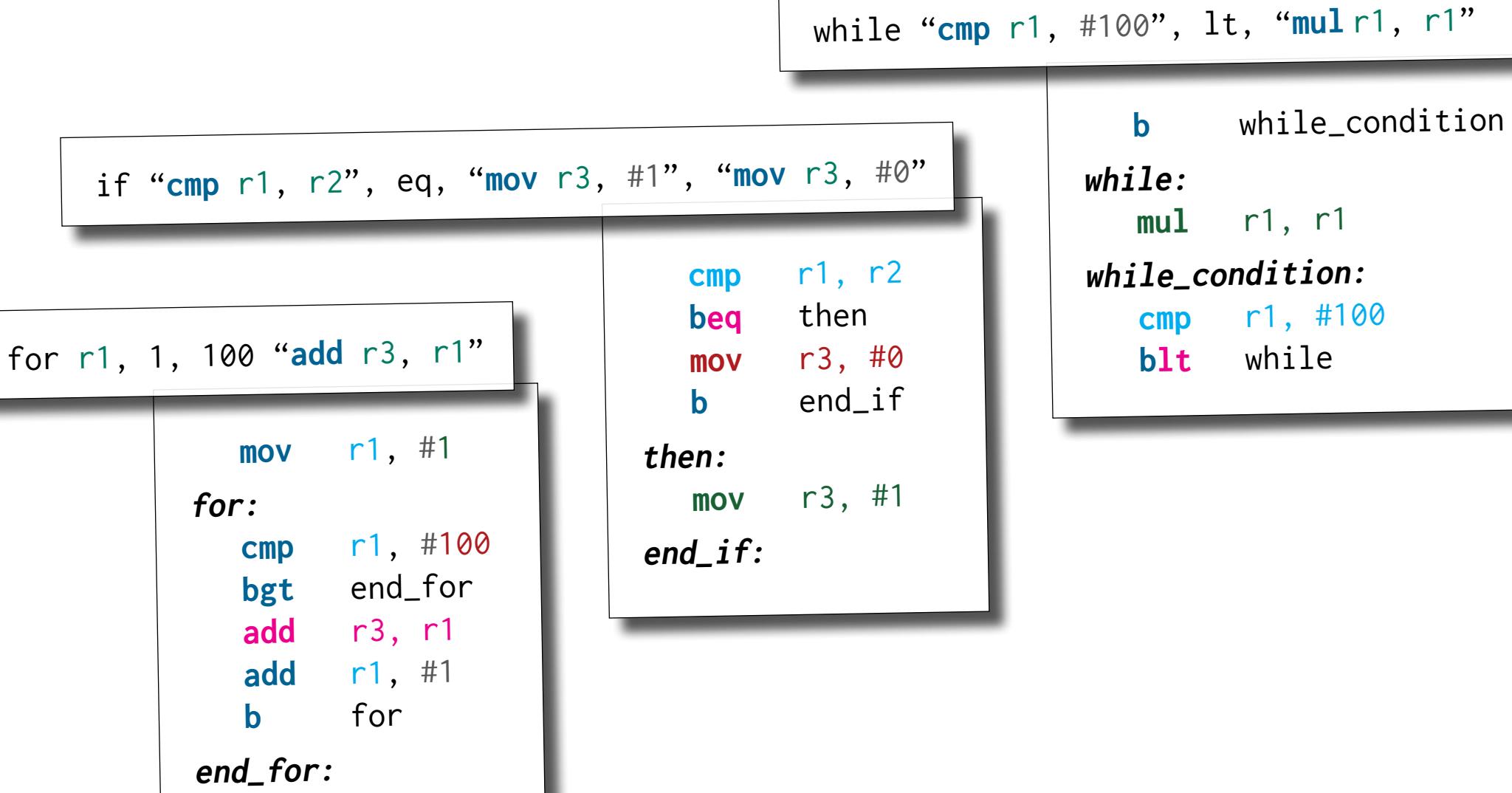
```
mov  r0, #0  
b    end_case
```

end_case:

Computational complexity: $O(n)$



Control Structures





Control Structures

```
case "cmp r1, r2", lt, "mov r0, r1",
      "cmp r1, r2", gt, "mov r0, r2",
      "cmp r1, r2", gt, "mov r0, #0"
```

```
indexed_case r1, "mov r3, r2", "mov r4, r2", "mov r5, r2"
```

```
tbh [PC, r1, lsl #1]
```

branch_table:

```
.hword (case_red - branch_table)/2
.hword (case_green - branch_table)/2
.hword (case_blue - branch_table)/2
```

case_red:

```
    mov r3, r2
    b end_case
```

case_green:

```
    mov r4, r2
    b end_case
```

case_Blue:

```
    mov r5, r2
```

end_case:

cmp	r1, r2
blt	case_a
cmp	r1, r2
bgt	case_b
cmp	r1, r2
beq	case_c
b	end_case

case_a:

mov	r0, r1
b	end_case

case_b:

mov	r0, r2
b	end_case

case_c:

mov	r0, #0
b	end_case

end_case:



Control Structures

```
indexed_case r1, "mov r3, r2", "mov r4, r2", "mov r5, r2"
```

```
tbh [PC, r1, lsl #1]
```

branch_table:

```
.hword (case_red - branch_table)/2  
.hword (case_green - branch_table)/2  
.hword (case_blue - branch_table)/2
```

case_red:

```
    mov r3, r2  
    b end_case
```

case_green:

```
    mov r4, r2  
    b end_case
```

case_Blue:

```
    mov r5, r2
```

end_case:

```
switch r1,  
4, "mov r0, r1",  
5, "mov r0, r2",  
6, "mov r0, #0"
```

```
    cmp r1, #4  
    beq case_a  
    cmp r1, #5  
    beq case_b  
    cmp r1, #6  
    beq case_c  
    b end_case
```

case_a:

```
    mov r0, r1  
    b end_case
```

case_b:

```
    mov r0, r2  
    b end_case
```

case_c:

```
    mov r0, #0  
    b end_case
```

end_case:



Control Structures

☞ You can form all common sequential control structures
(or generate them via macros if you wish)

(including function calls)



Control Structures

Summary

Control Structures

- **Assembler Macros**
 - Local labels
 - Recursive macros
- **Control Structures in machine code**
 - IF
 - WHILE
 - FOR
 - CASEs

Computer Organisation & Program Execution 2021



7

Operating Systems

Uwe R. Zimmer - The Australian National University



Operating Systems

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

Chapter 4 “The Processor”,

Chapter 6 “Parallel Processors from Client to Cloud”

ARM edition, Morgan Kaufmann 2017



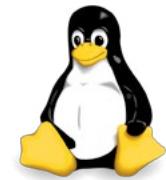
Operating Systems



iOS



FreeBSD®



VxWorks

AdaCore
The GNAT Pro Company



Windows 10





Operating Systems

What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment

(e.g. memory management and protection, hardware abstraction,
process management, inter-process communication, ...)



Operating Systems

What is an operating system?

1. A virtual machine!

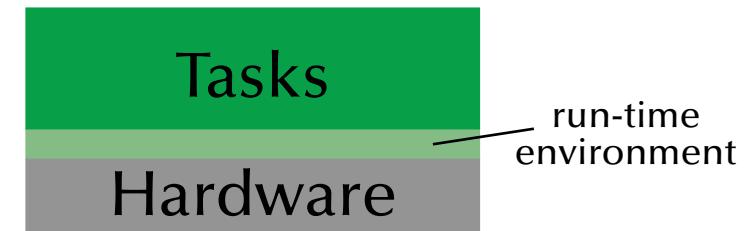
... offering a more comfortable and safer environment



Typ. general OS



Typ. real-time system



Typ. embedded system



Operating Systems

What is an operating system?

2. A resource manager!

... coordinating access to hardware resources



Operating Systems

What is an operating system?

2. A resource manager!

... coordinating access to hardware resources

Operating systems deal with

- processors
- memory
- mass storage
- communication channels
- devices (timers, special purpose processors, peripheral hardware, ...)

 and tasks/processes/programs which are applying for access to these resources!



Operating Systems

The evolution of operating systems

- in the beginning: single user, single program, single task, serial processing - no OS
- 50s: System monitors / batch processing
 - ☞ the monitor ordered the sequence of jobs and triggered their sequential execution
- 50s-60s: Advanced system monitors / batch processing:
 - ☞ the monitor is handling interrupts and timers
 - ☞ first support for memory protection
 - ☞ first implementations of privileged instructions (accessible by the monitor only).
- early 60s: Multiprogramming systems:
 - ☞ employ the long device I/O delays for switches to other, runnable programs
- early 60s: Multiprogramming, time-sharing systems:
 - ☞ assign time-slices to each program and switch regularly
- early 70s: Multitasking systems – multiple developments resulting in UNIX (besides others)
- early 80s: single user, single tasking systems, with emphasis on user interface or APIs. MS-DOS, CP/M, MacOS and others first employed ‘small scale’ CPUs (personal computers).
- mid-80s: Distributed/multiprocessor operating systems - modern UNIX systems (SYSV, BSD)



Operating Systems

Types of current operating systems

Personal computing systems, workstations, and workgroup servers:

- late 70s: Workstations starting by porting UNIX or VMS to 'smaller' computers.
- 80s: PCs starting with almost none of the classical OS-features and services, but with a user-interface (MacOS) and simple device drivers (MS-DOS)
- ☞ last 20 years: evolving and expanding into current general purpose OSs, like for instance:
 - Solaris (based on SVR4, BSD, and SunOS)
 - LINUX (open source UNIX re-implementation for x86 processors and others)
 - current Windows (used to be partly based on Windows NT, which is 'related' to VMS)
 - MacOS (Mach kernel with BSD Unix and a proprietary user-interface)
- Multiprocessing is supported by all these OSs to some extent.
- None of these OSs are suitable for embedded systems, although trials have been performed.
- None of these OSs are suitable for distributed or real-time systems.



Operating Systems

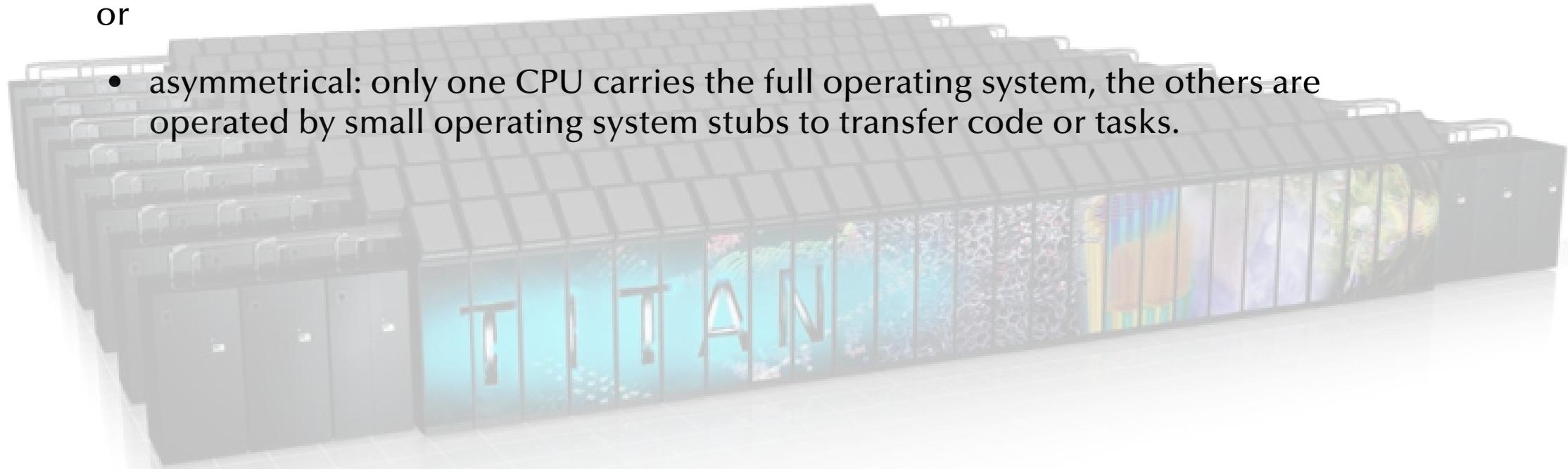
Types of current operating systems

Parallel operating systems

- support for a large number of processors, either:
 - symmetrical: each CPU has a full copy of the operating system

or

- asymmetrical: only one CPU carries the full operating system, the others are operated by small operating system stubs to transfer code or tasks.



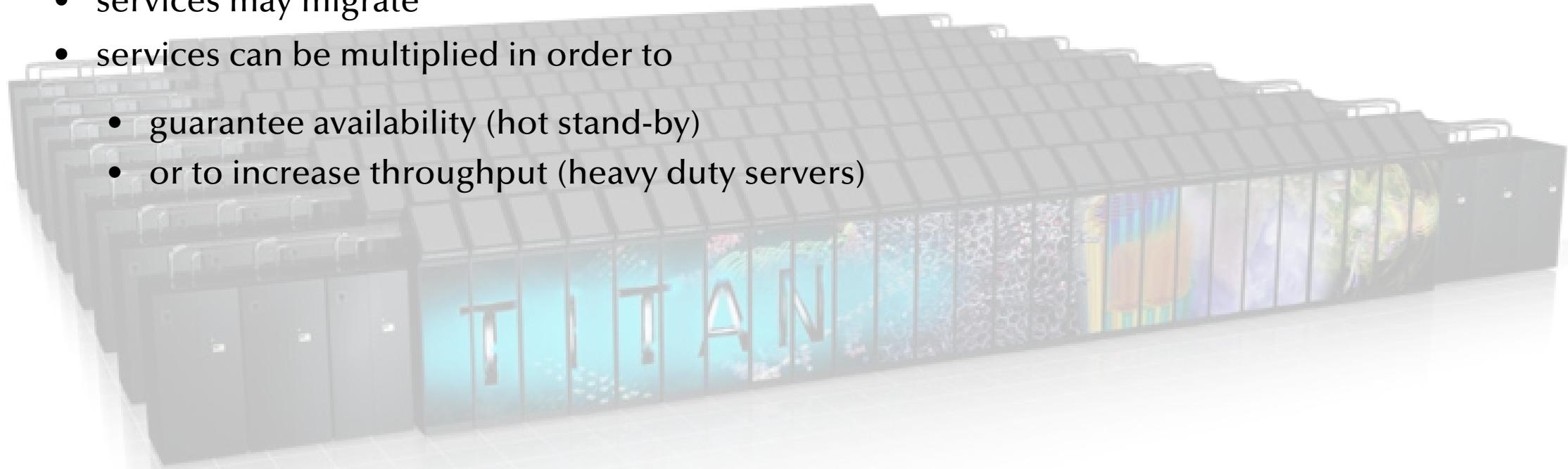


Operating Systems

Types of current operating systems

Distributed operating systems

- all CPUs carry a small kernel operating system for communication services.
- all other OS-services are distributed over available CPUs
- services may migrate
- services can be multiplied in order to
 - guarantee availability (hot stand-by)
 - or to increase throughput (heavy duty servers)





Operating Systems

Types of current operating systems

Real-time operating systems

- Fast context switches?
- Small size?
- Quick response to external interrupts?
- Multitasking?
- ‘low level’ programming interfaces?
- Interprocess communication tools?
- High processor utilization?



Operating Systems

Types of current operating systems

Real-time operating systems

- ~~Fast context switches?~~ should be fast anyway
- ~~Small size?~~ should be small anyway
- ~~Quick response to external interrupts?~~ not 'quick', but predictable
- ~~Multitasking?~~ often, not always
- ~~'low level' programming interfaces?~~ needed in many operating systems
- ~~Interprocess communication tools?~~ needed in almost all operating systems
- ~~High processor utilization?~~ fault tolerance builds on redundancy!



Operating Systems

Types of current operating systems

Real-time operating systems need to provide...

- ☞ the logical correctness of the results as well as
- ☞ the correctness of the time, when the results are delivered



Predictability!
(not performance!)

Photo: NASA

- ☞ All results are to be delivered just-in-time – not too early, not too late.

Timing constraints are specified in many different ways ...

... often as a response to 'external' events

- ☞ reactive systems



Operating Systems

Types of current operating systems

Embedded operating systems

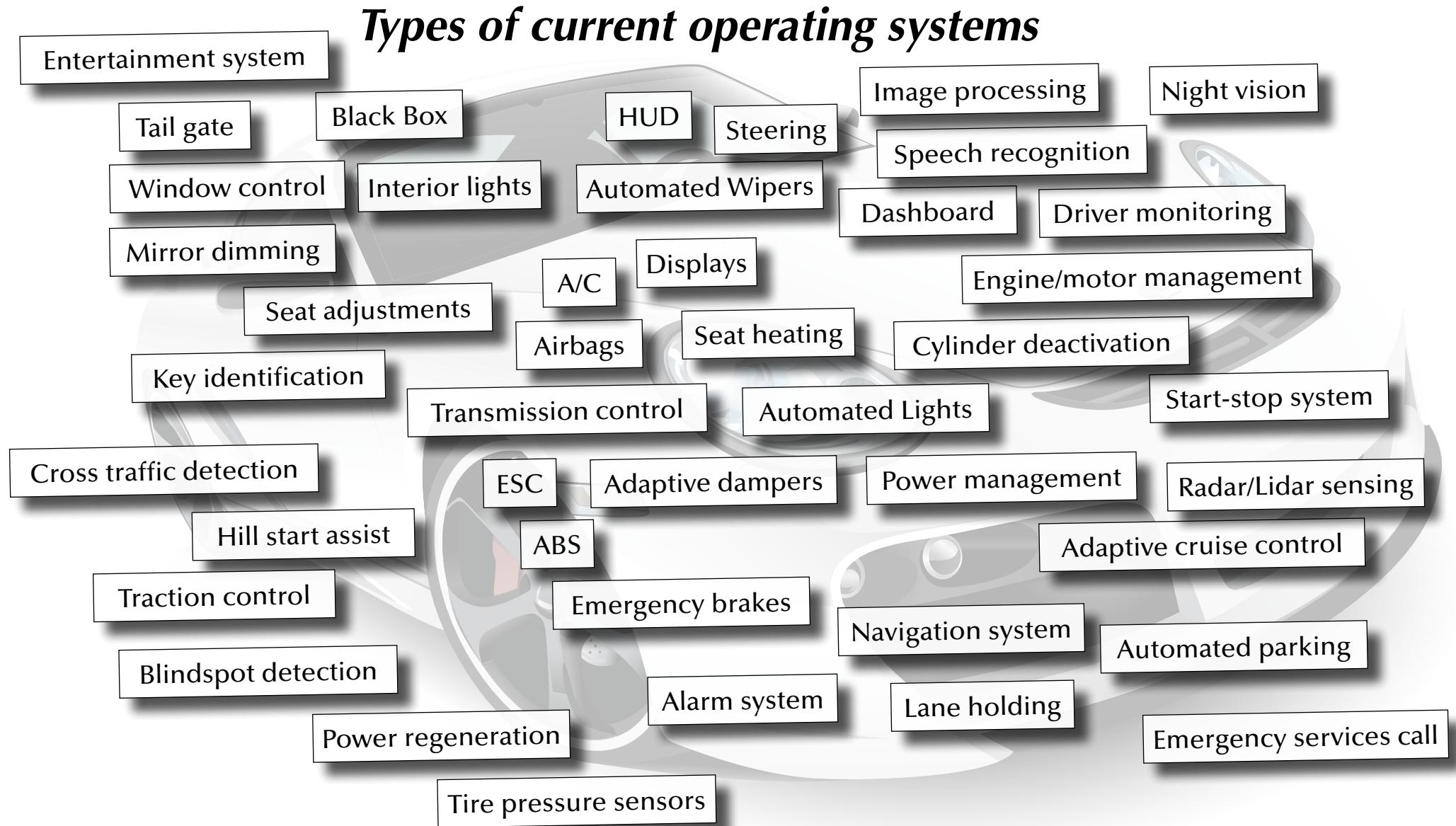
- usually real-time systems, often hard real-time systems
- very small footprint (often a few kBytes)
- none or limited user-interaction
- ☞ 90-95% of all processors are working here!



Artwork by Q. Mehdi (cc attribution license)



Operating Systems



Artwork by Q. Mehdi (cc attribution license)



Operating Systems

Types of current operating systems

Embedded operating systems

- usually real-time systems, often hard real-time systems
 - very small footprint (often a few kBytes)
 - none or limited user-interaction
- ☞ 90-95% of all processors are working here!

Often over 100 MPUs per car
(and some of them quite high performant)



Operating Systems

What is an operating system?

Is there a standard set of features for operating systems?



Operating Systems

What is an operating system?

Is there a standard set of features for operating systems?

👉 **no:**

the term ‘operating system’ covers 4 kB microkernels,
as well as > 1 GB installations of desktop general purpose operating systems.



Operating Systems

What is an operating system?

Is there a standard set of features for operating systems?

👉 **no:**

the term ‘operating system’ covers 4 kB microkernels,
as well as > 1 GB installations of desktop general purpose operating systems.

Is there a minimal set of features?



Operating Systems

What is an operating system?

Is there a standard set of features for operating systems?

👉 no:

the term ‘operating system’ covers 4 kB microkernels,
as well as > 1 GB installations of desktop general purpose operating systems.

Is there a minimal set of features?

👉 almost:

memory management, process management and inter-process communication/synchronisation
will be considered essential in most systems



Operating Systems

What is an operating system?

Is there a standard set of features for operating systems?

👉 no:

the term ‘operating system’ covers 4 kB microkernels,
as well as > 1 GB installations of desktop general purpose operating systems.

Is there a minimal set of features?

👉 almost:

memory management, process management and inter-process communication/synchronisation
will be considered essential in most systems

Is there always an explicit operating system?



Operating Systems

What is an operating system?

Is there a standard set of features for operating systems?

☞ **no:**

the term ‘operating system’ covers 4 kB microkernels,
as well as > 1 GB installations of desktop general purpose operating systems.

Is there a minimal set of features?

☞ **almost:**

memory management, process management and inter-process communication/synchronisation
will be considered essential in most systems

Is there always an explicit operating system?

☞ **no:**

some languages and development systems operate with standalone runtime environments



Operating Systems

Typical features of operating systems

Process management:

- Context switch
- Scheduling
- Book keeping (creation, states, cleanup)

☞ context switch:

☞ needs to...

- ‘remove’ one process from the CPU while preserving its state
- choose another process (scheduling)
- ‘insert’ the new process into the CPU, restoring the CPU state

Some CPUs have hardware support for context switching, otherwise:

☞ use interrupt mechanism



Operating Systems

Typical features of operating systems

Memory management:

- Allocation / Deallocation
- Virtual memory: logical vs. physical addresses, segments, paging, swapping, etc.
- Memory protection (privilege levels, separate virtual memory segments, ...)
- Shared memory

Synchronisation / Inter-process communication

- semaphores, mutexes, cond. variables, channels, mailboxes, MPI, etc. (chapter 4)
- ☞ tightly coupled to scheduling / task switching!

Hardware abstraction

- Device drivers
- API
- Protocols, file systems, networking, everything else...



Operating Systems

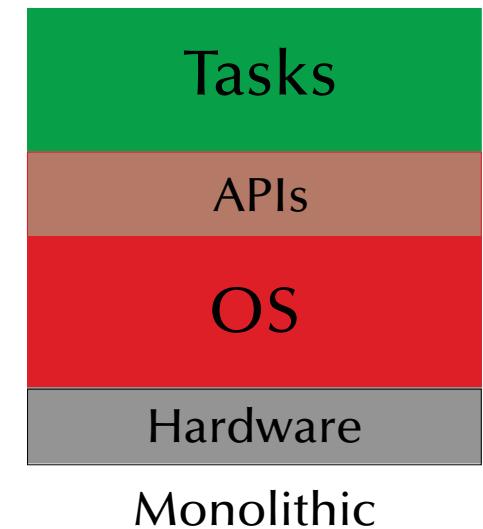
Typical structures of operating systems

Monolithic

(or ‘the big mess...’)

- non-portable
- hard to maintain
- lacks reliability
- all services are in the kernel (on the same privilege level)

☞ but: may reach high efficiency



e.g. most early UNIX systems,
MS-DOS (80s), Windows (all non-NT based versions)
MacOS (until version 9), and many others...



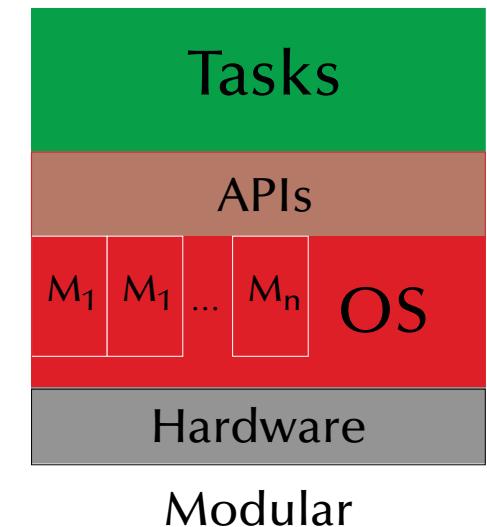
Operating Systems

Typical structures of operating systems

Monolithic & Modular

- Modules can be platform independent
- Easier to maintain and to develop
- Reliability is increased
- all services are still in the kernel (on the same privilege level)

☞ may reach high efficiency



e.g. current Linux versions

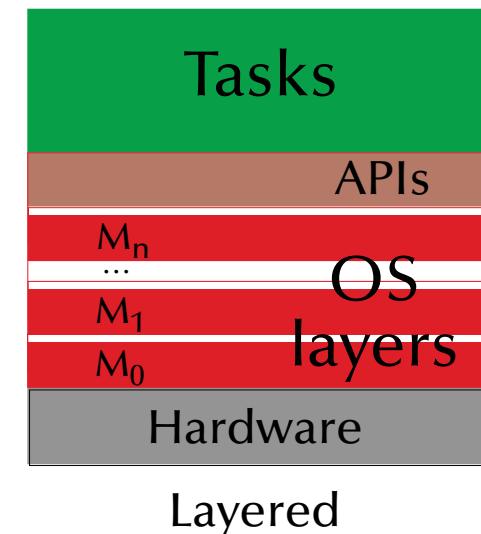


Operating Systems

Typical structures of operating systems

Monolithic & layered

- easily portable
- significantly easier to maintain
- crashing layers do not necessarily stop the whole OS
- possibly reduced efficiency through many interfaces
- rigorous implementation of the stacked virtual machine perspective on OSs



e.g. some current UNIX implementations (e.g. Solaris) to a certain degree, many research OSs (e.g. 'THE system', Dijkstra '68)



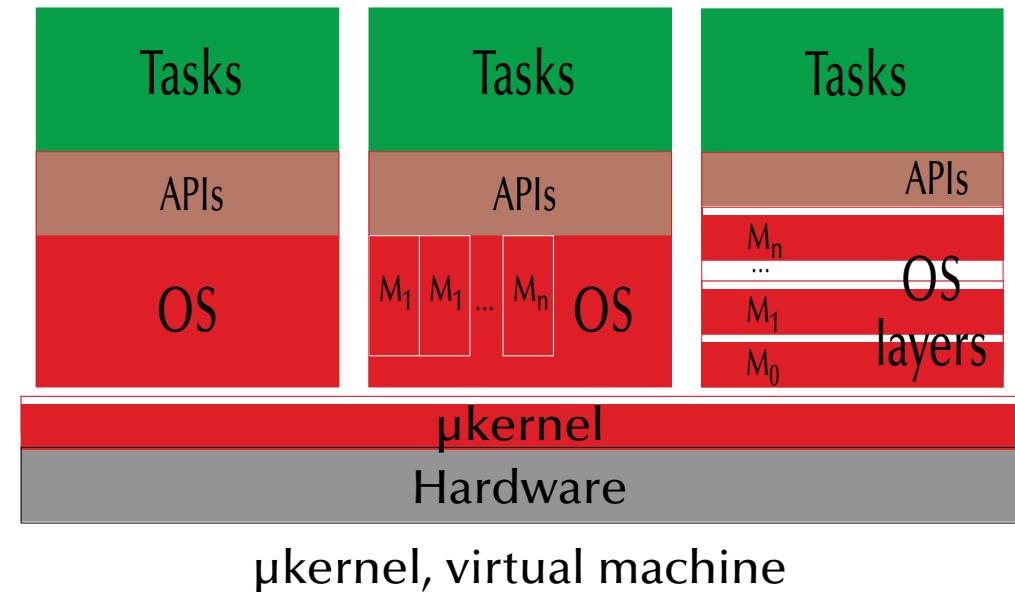
Operating Systems

Typical structures of operating systems

μ Kernels & virtual machines

- μ kernel implements essential process, memory, and message handling
- all 'higher' services are dealt with outside the kernel ↪ no threat for the kernel stability
- significantly easier to maintain
- multiple OSs can be executed at the same time
- μ kernel is highly hardware dependent ↪ only the μ kernel needs to be ported.
- possibly reduced efficiency through increased communications

e.g. wide spread concept: as early as the CP/M, VM/370 ('79) or as recent as MacOS X (mach kernel + BSD unix), ...



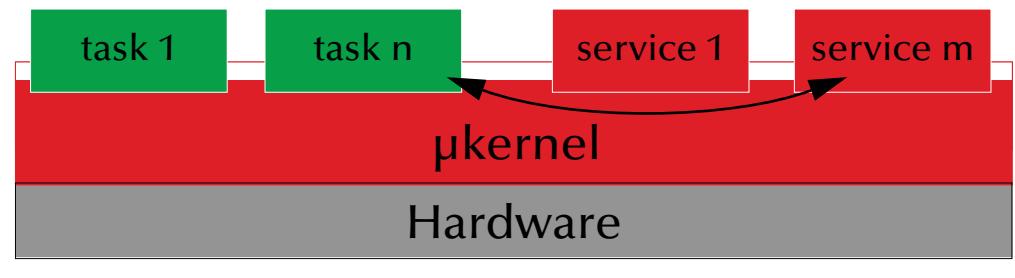


Operating Systems

Typical structures of operating systems

μ Kernels & client-server models

- μ kernel implements essential process, memory, and message handling
- all 'higher' services are user level servers
- significantly easier to maintain
- kernel ensures reliable message passing between clients and servers
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications



μ kernel, client server structure

e.g. current research projects, L4, etc.

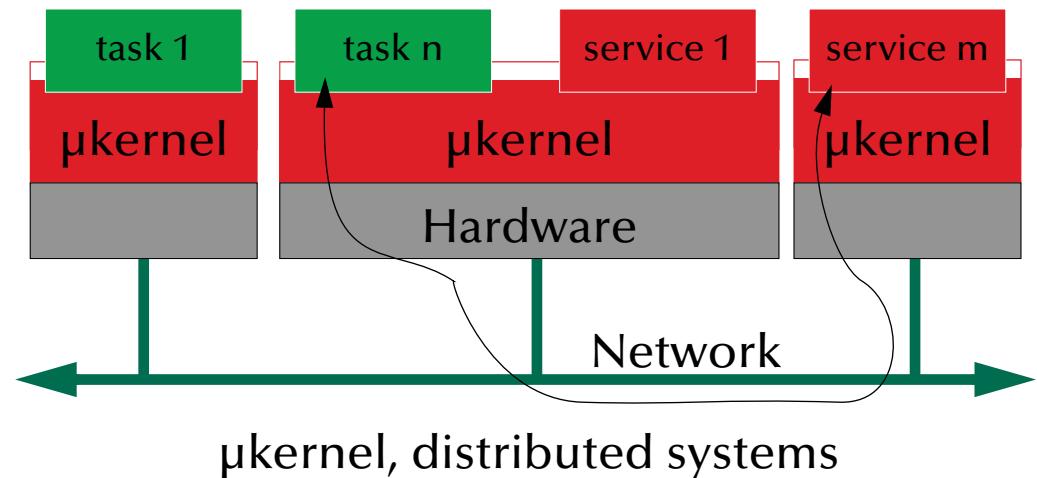


Operating Systems

Typical structures of operating systems

μ Kernels & client-server models

- μ kernel implements essential process, memory, and message handling
- all 'higher' services are user level servers
- significantly easier to maintain
- kernel ensures reliable message passing between clients and servers: locally and through a network
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications



e.g. Java engines,
distributed real-time operating systems, current distributed OSs research projects



Operating Systems

UNIX

UNIX features

- Hierarchical file-system (maintained via ‘mount’ and ‘umount’)
 - Universal file-interface applied to files, devices (I/O), as well as IPC
 - Dynamic process creation via duplication
 - Choice of shells
 - Internal structure as well as all APIs are based on ‘C’
 - Relatively high degree of portability
- ☞ UNICS, UNIX, **BSD**, XENIX, **System V**, **QNX**, IRIX, SunOS, Ultrix, Sinix, **Mach**, Plan 9, NeXTSTEP, **AIX**, HP-UX, **Solaris**, **NetBSD**, **FreeBSD**, **Linux**, OPEN-STEP, **OpenBSD**, **Darwin**, **QNX/Neutrino**, **OS X**, **QNX RTOS**,



Operating Systems

Introduction to processes and threads

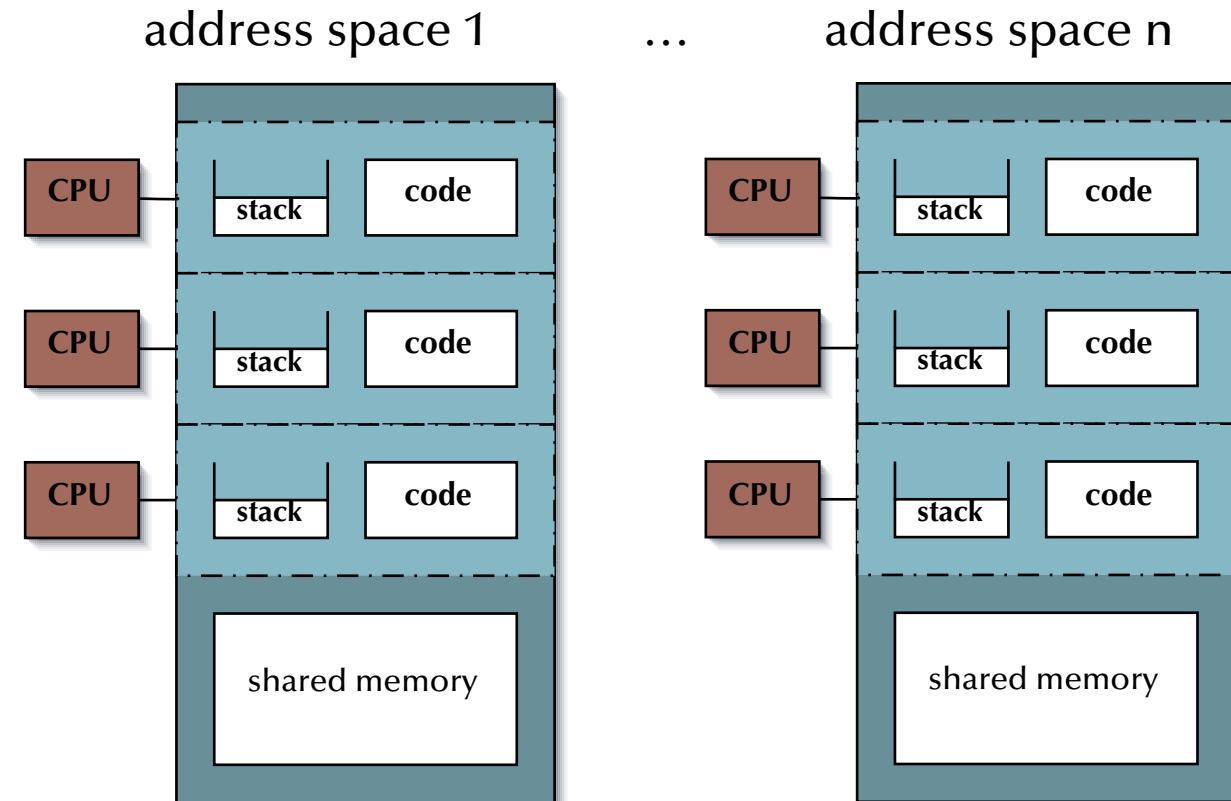
1 CPU per control-flow

Specific configurations only, e.g.:

- Distributed µcontrollers.

- Physical process control systems:

1 cpu per task,
connected via a
bus-system.



☞ **Process management**
(scheduling) not required.

☞ **Shared memory access**
need to be coordinated.

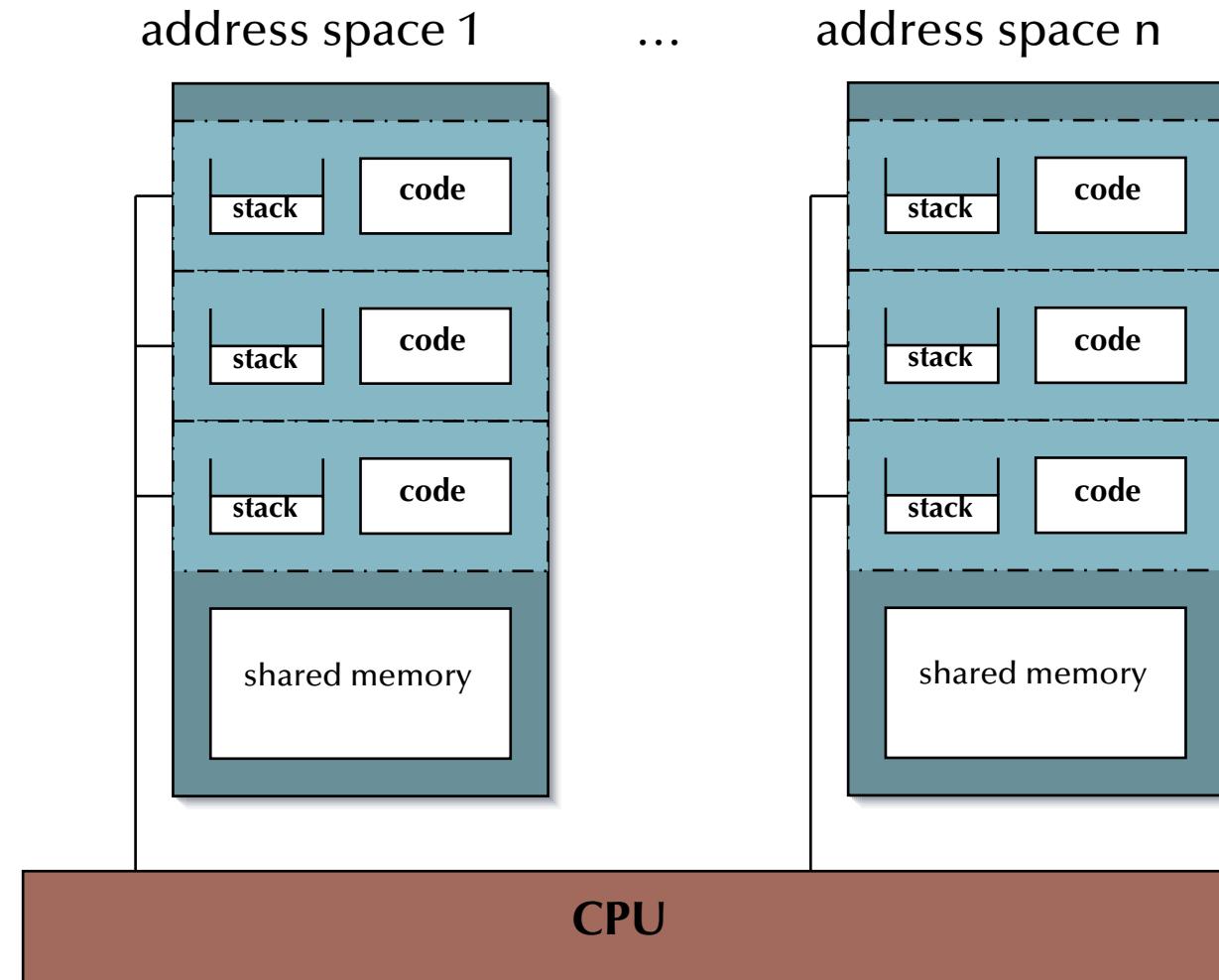


Operating Systems

Introduction to processes and threads

1 CPU for all control-flows

- OS: emulate one CPU for every control-flow:
Multi-tasking operating system
 - ☞ Support for **memory protection** essential.
 - ☞ **Process management** (scheduling) required.
 - ☞ **Shared memory access** need to be coordinated.





Operating Systems

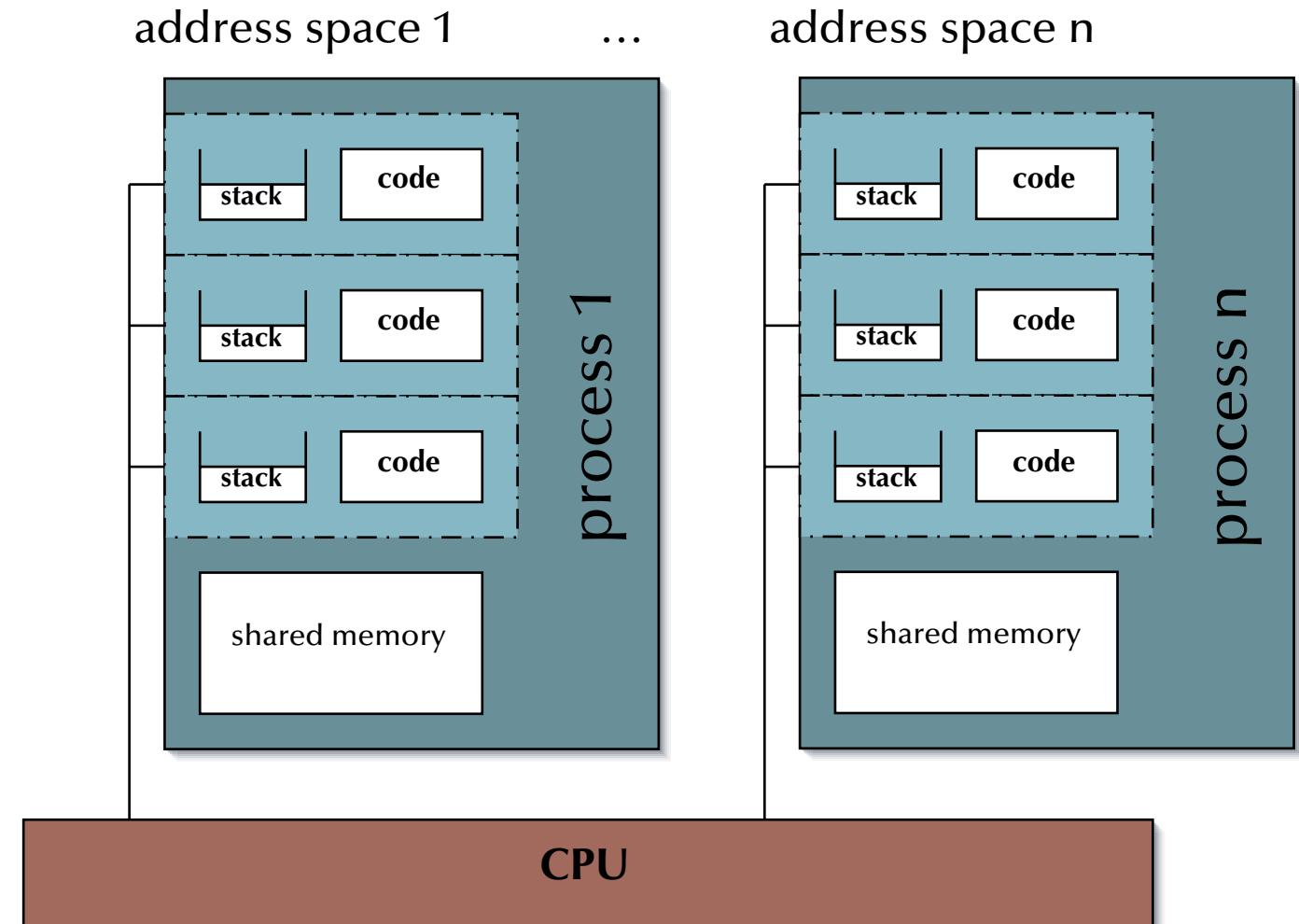
Introduction to processes and threads

Processes

Process ::=

Address space
+ Control flow(s)

- Kernel has full knowledge about all processes as well as their **states, requirements** and currently held **resources**.





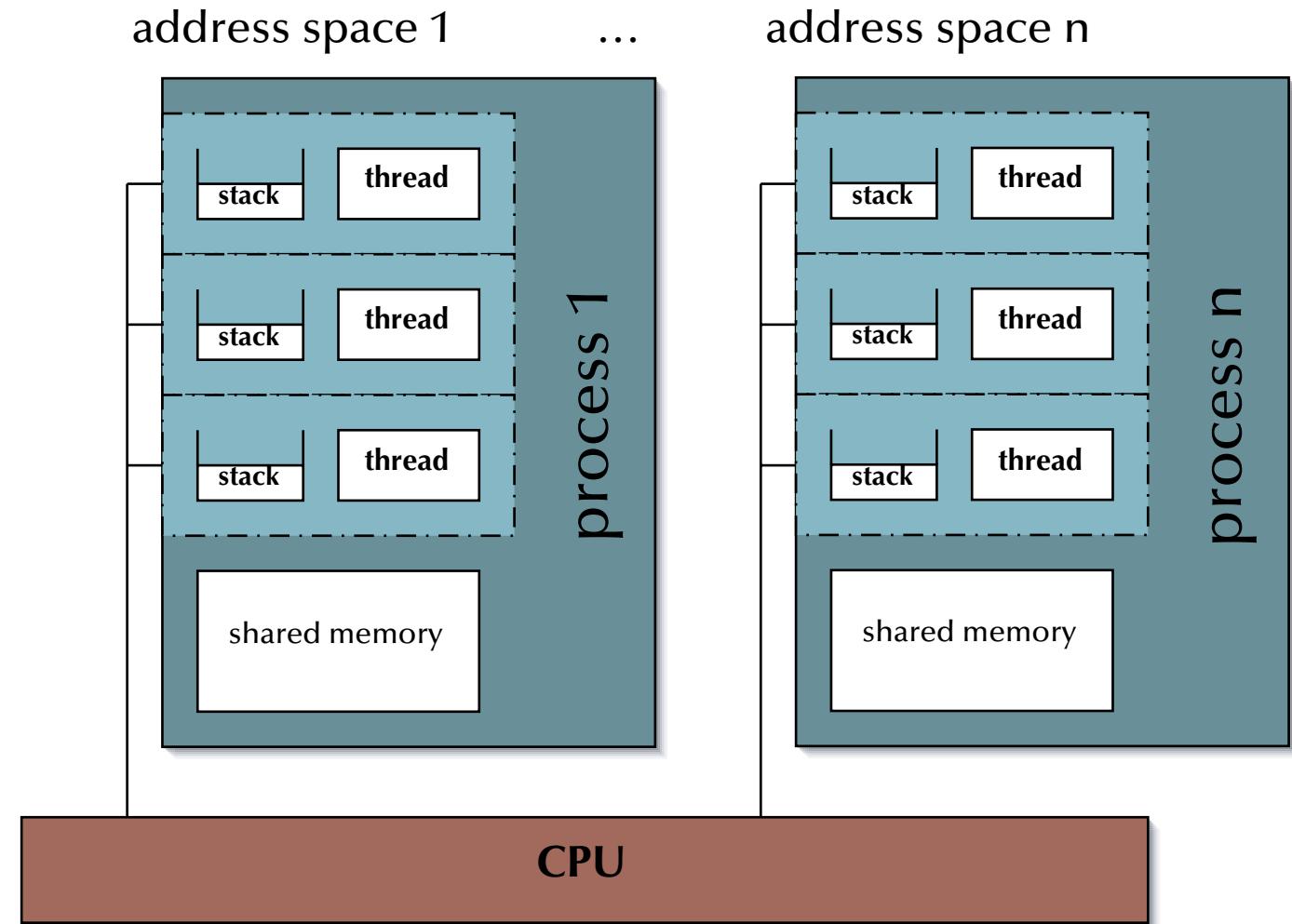
Operating Systems

Introduction to processes and threads

Threads

Threads (individual control-flows) can be handled:

- *Inside the OS:*
 - ☞ Kernel scheduling.
 - Thread can easily be connected to external events (I/O).
- *Outside the OS:*
 - ☞ User-level scheduling.
 - Threads may need to go through their parent process to access I/O.





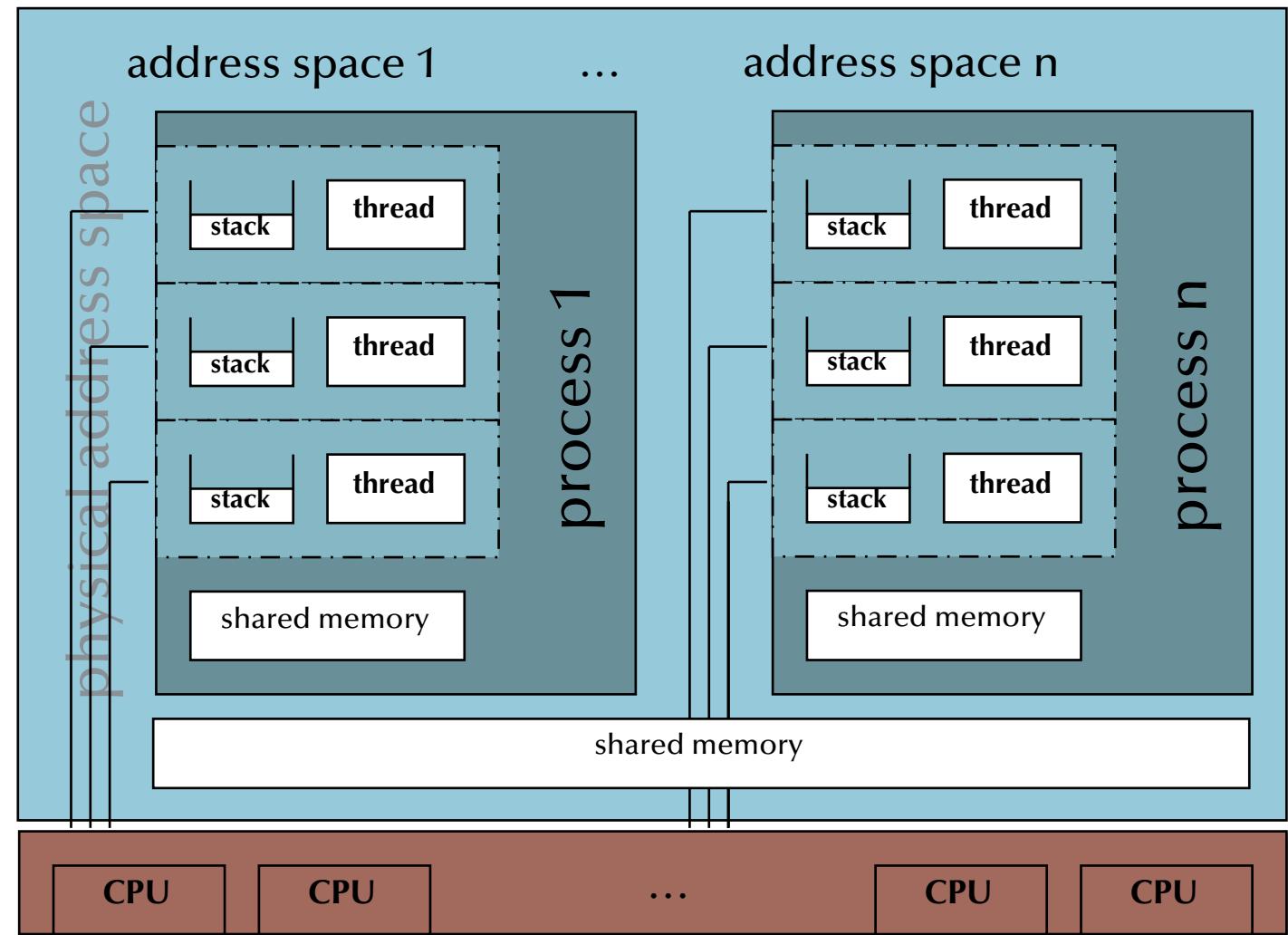
Operating Systems

Introduction to processes and threads

Symmetric Multiprocessing (SMP)

All CPUs share the same physical address space (and access to resources).

- ☞ Any process / thread can be executed on any available CPU.





Operating Systems

Introduction to processes and threads

Processes ↔ Threads

Also processes can share memory and the specific definition of threads is different in different operating systems and contexts:

- ☞ Threads can be regarded as a group of processes, which share some resources (☞ process-hierarchy).
- ☞ Due to the overlap in resources, the attributes attached to threads are less than for ‘first-class-citizen-processes’.
- ☞ Thread switching and inter-thread communication can be more efficient than switching on process level.
- ☞ Scheduling of threads depends on the actual thread implementations:
 - e.g. *user-level control-flows*, which the kernel has no knowledge about at all.
 - e.g. *kernel-level control-flows*, which are handled as processes with some restrictions.



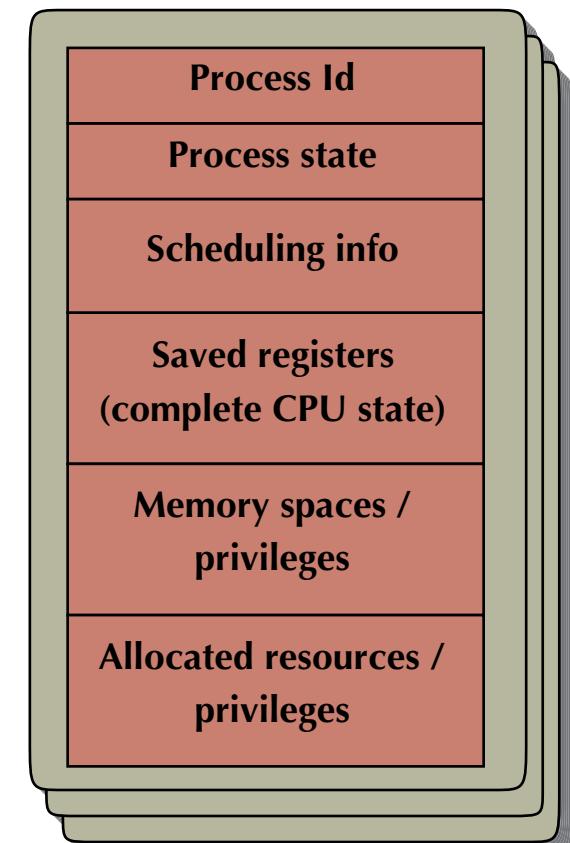
Operating Systems

Introduction to processes and threads

Process Control Blocks

- **Process Id**
- **Process state:**
{created, ready, executing, blocked, suspended, bored ...}
- **Scheduling attributes:**
Priorities, deadlines, consumed CPU-time, ...
- **CPU state:** Saved/restored information while context switches (incl. the program counter, stack pointer, ...)
- **Memory attributes / privileges:**
Memory base, limits, shared areas, ...
- **Allocated resources / privileges:**
Open and requested devices and files, ...

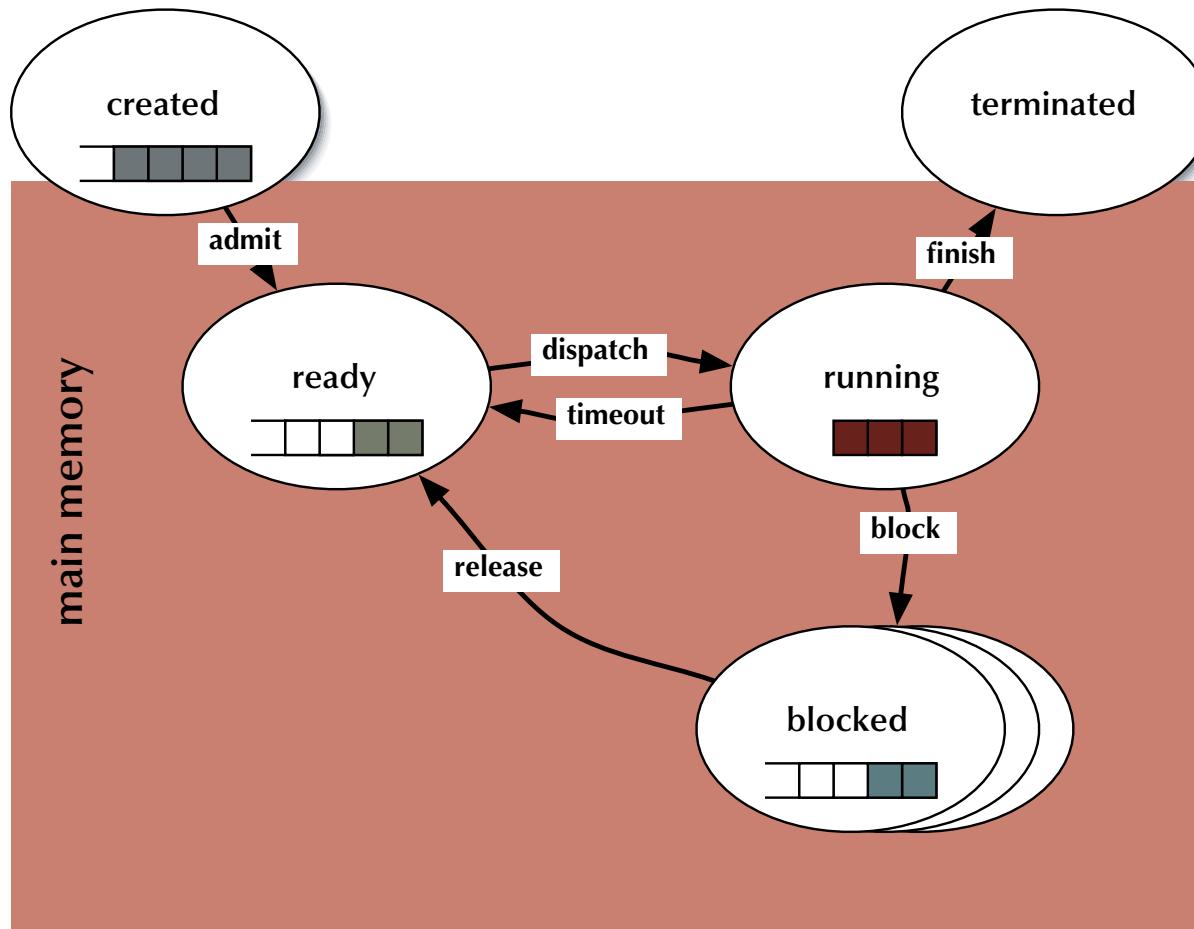
Process Control Blocks (PCBs)



... PCBs (links thereof) are commonly enqueued at a certain state or condition (awaiting access or change in state)



Operating Systems

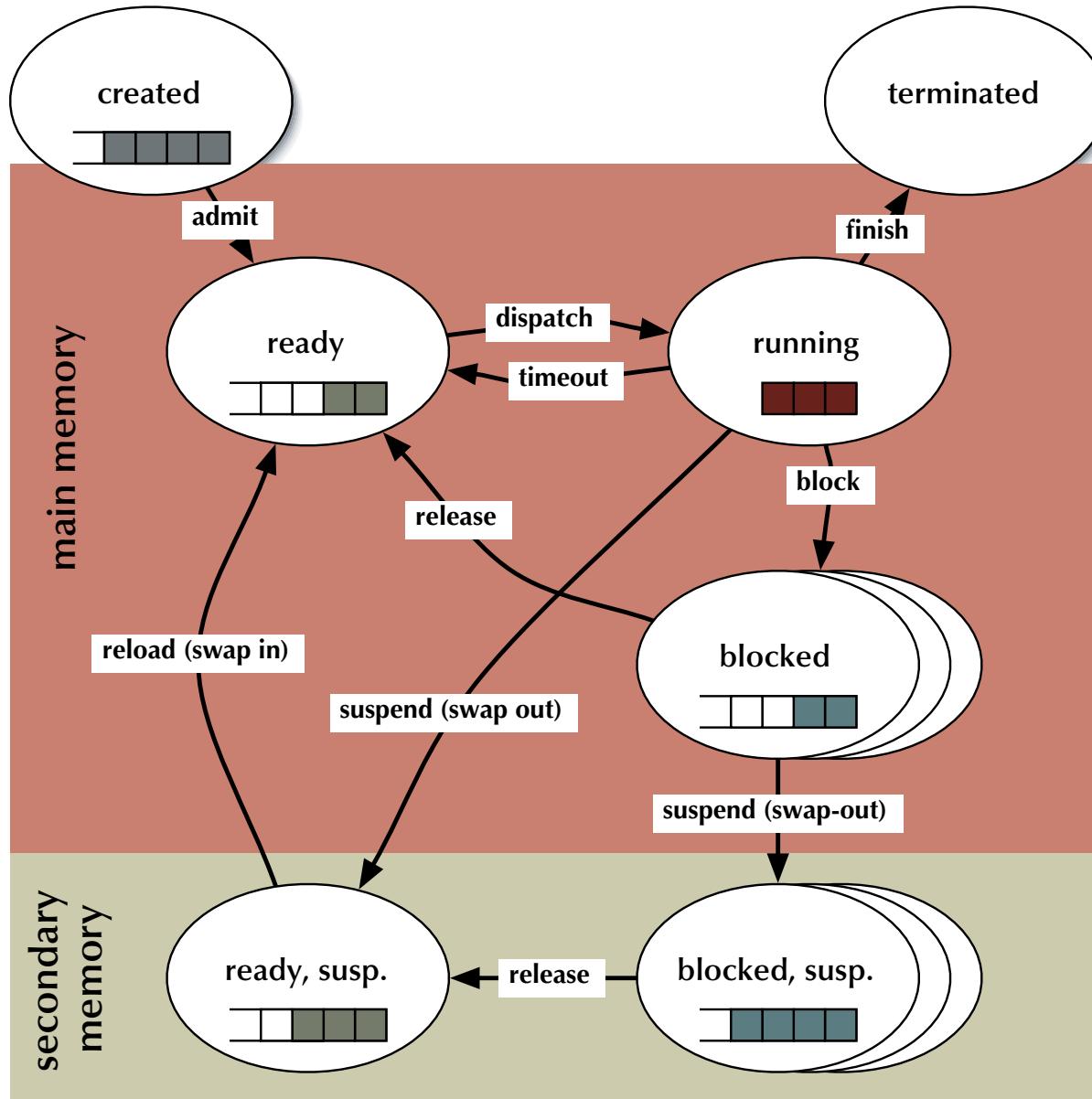


Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher
☞ waiting for admission
- **ready**: ready to run
☞ waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
☞ waiting for a resource



Operating Systems

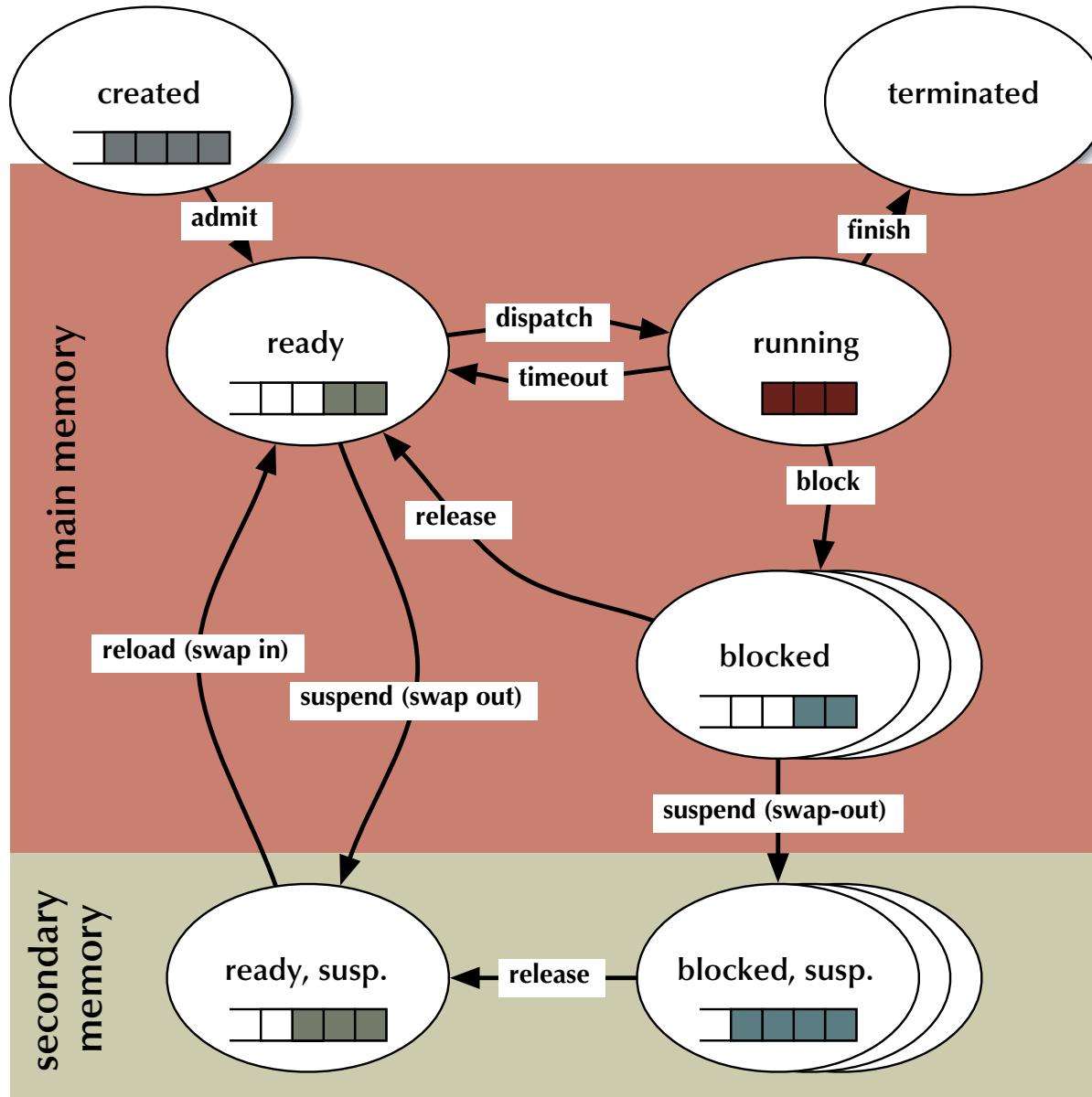


Process states

- **created:** the task is ready to run, but not yet considered by any dispatcher
☞ waiting for admission
- **ready:** ready to run
☞ waiting for a free CPU
- **running:** holds a CPU and executes
- **blocked:** not ready to run
☞ waiting for a resource
- **suspended states:** swapped out of main memory
(none time critical processes)
☞ waiting for main memory space (and other resources)



Operating Systems



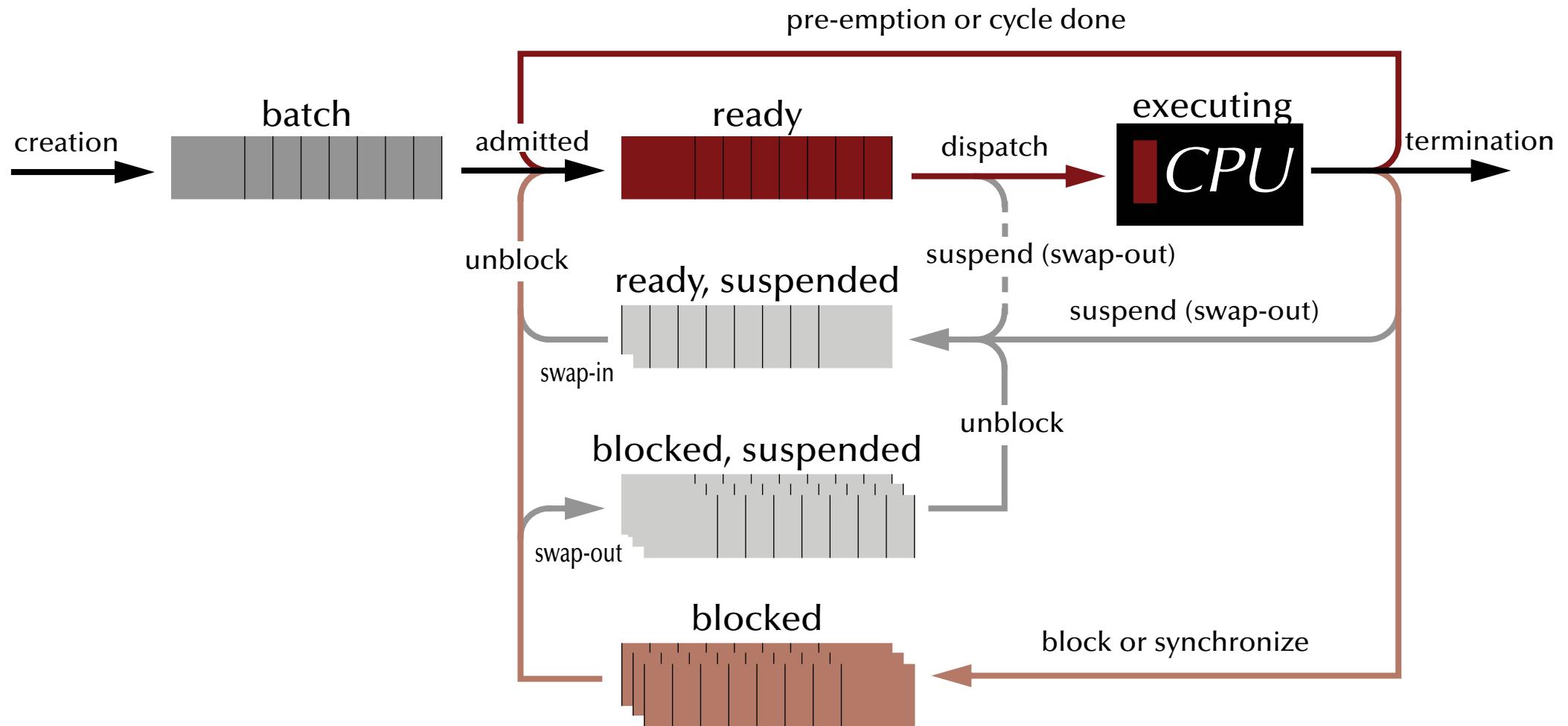
Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher
☞ waiting for admission
 - **ready**: ready to run
☞ waiting for a free CPU
 - **running**: holds a CPU and executes
 - **blocked**: not ready to run
☞ waiting for a resource
 - **suspended states**: swapped out of main memory
(none time critical processes)
☞ waiting for main memory space (and other resources)
- ☞ dispatching and suspending can now be independent modules



Operating Systems

Process states

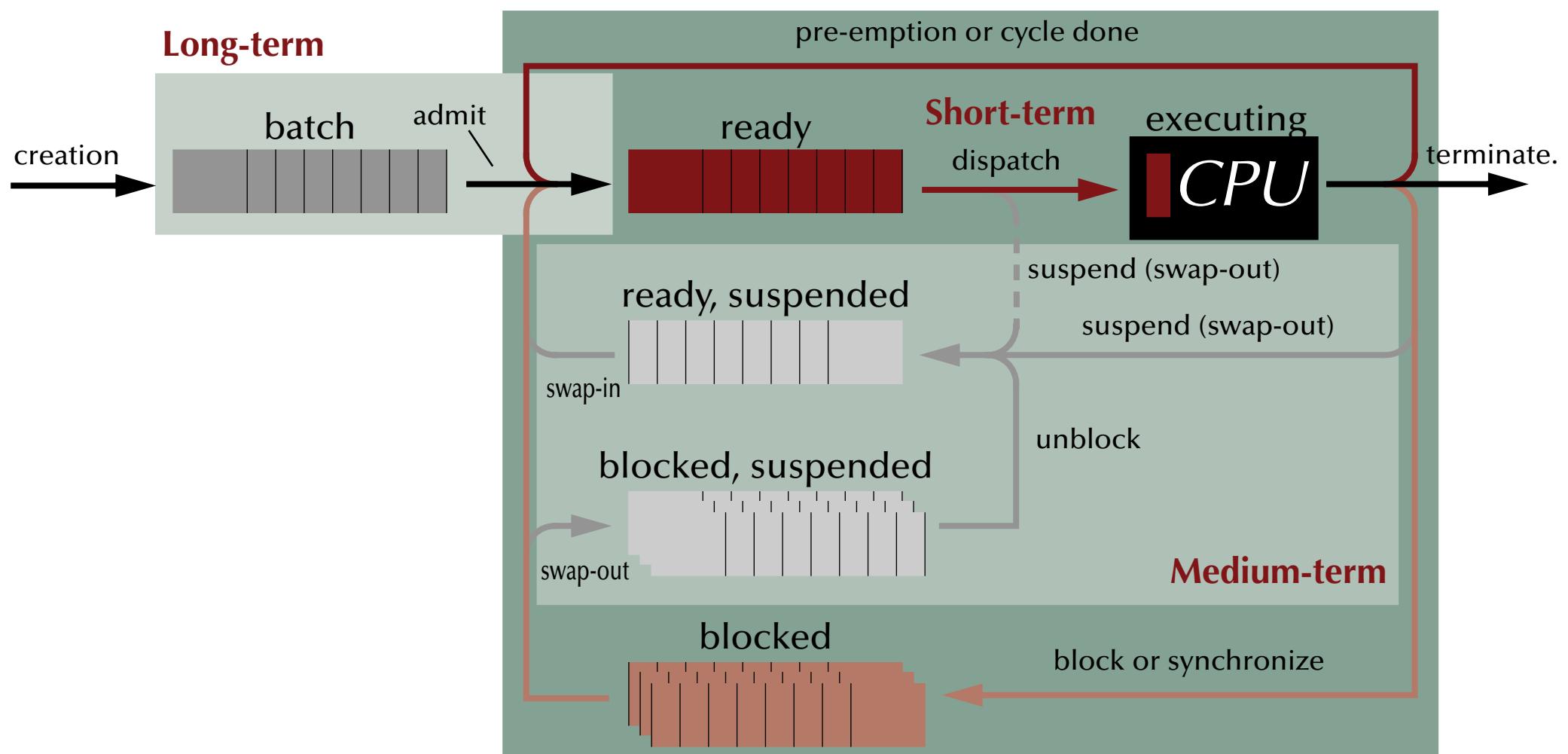




Operating Systems

Definition of terms

Time scales of scheduling

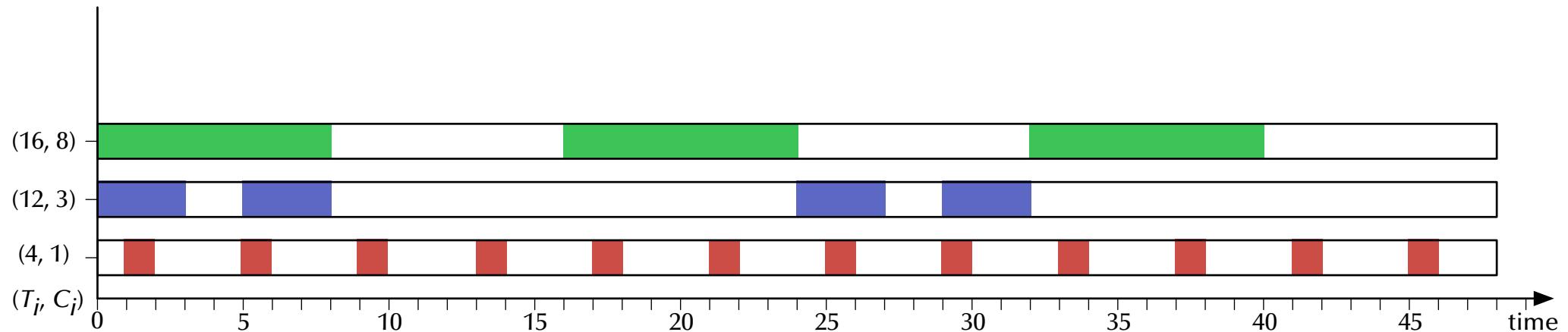




Operating Systems

Performance scheduling

Requested resource times



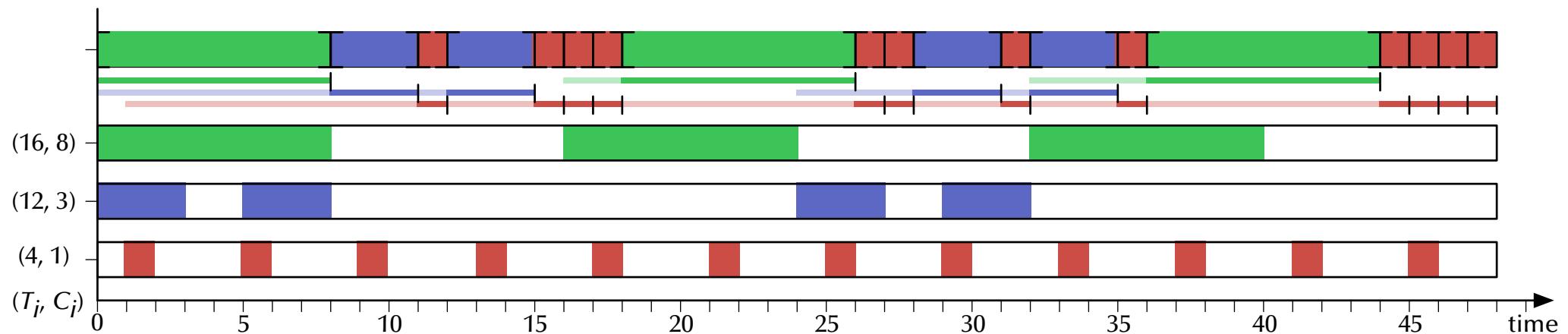
Tasks have an **average time between instantiations** of T_i
and a constant **computation time** of C_i



Operating Systems

Performance scheduling

First come, first served (FCFS)



Waiting time: 0..11, average: 5.9 – **Turnaround time:** 3..12, average: 8.4

As tasks apply *concurrently* for resources, the actual sequence of arrival is non-deterministic.

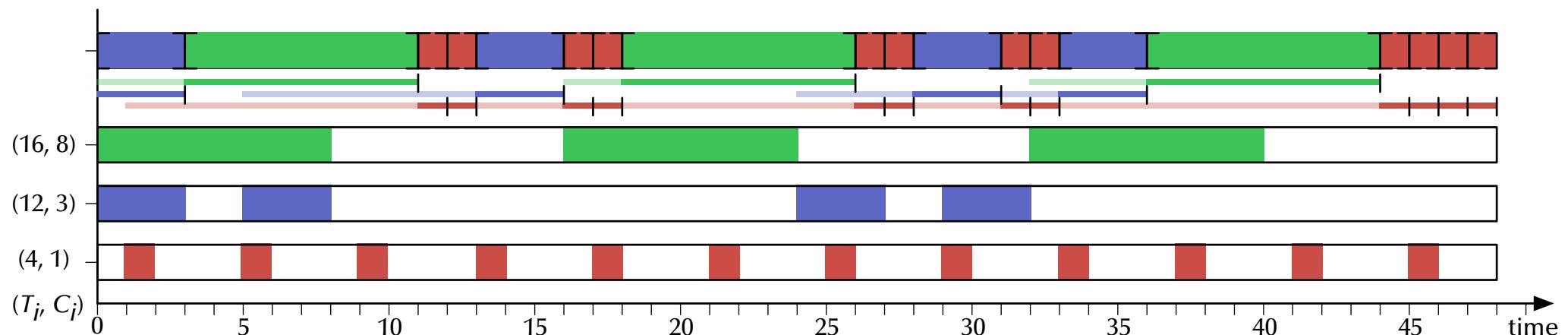
☞ hence even a deterministic scheduling schema like FCFS can lead to different outcomes.



Operating Systems

Performance scheduling

First come, first served (FCFS)



Waiting time: 0..11, average: 5.4 – **Turnaround time:** 3..12, average: 8.0

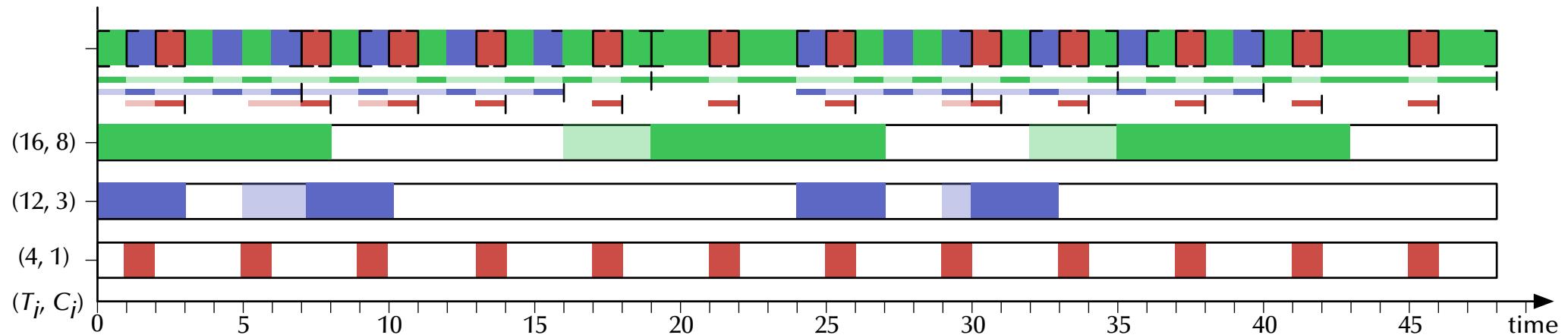
- ☞ In this example:
 - the average waiting times vary between 5.4 and 5.9
 - the average turnaround times vary between 8.0 and 8.4
- ☞ **Shortest possible maximal turnaround time!**



Operating Systems

Performance scheduling

Round Robin (RR)



Waiting time: 0..5, average: 1.2 – Turnaround time: 1..20, average: 5.8

- ☞ Optimized for swift initial responses.
- ☞ “Stretches out” long tasks.
- ☞ **Bound maximal waiting time!** (depended only on the number of tasks)



Operating Systems

Summary

Operating Systems

- **Operating Systems**

- Concept
- Categories
- Architectures

- **Processes**

- Definition
- Relation to architectures
- Scheduling

Computer Organisation & Program Execution 2021



8

Networks

Uwe R. Zimmer - The Australian National University



Networks

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

Chapter 4 “The Processor”,

Chapter 6 “Parallel Processors from Client to Cloud”

ARM edition, Morgan Kaufmann 2017



Networks

Network protocols & standards

OSI network reference model

Standardized as the

Open Systems Interconnection (OSI) reference model by the International Standardization Organization (ISO) in 1977

- 7 layer architecture
- Connection oriented

Hardy implemented anywhere in full ...

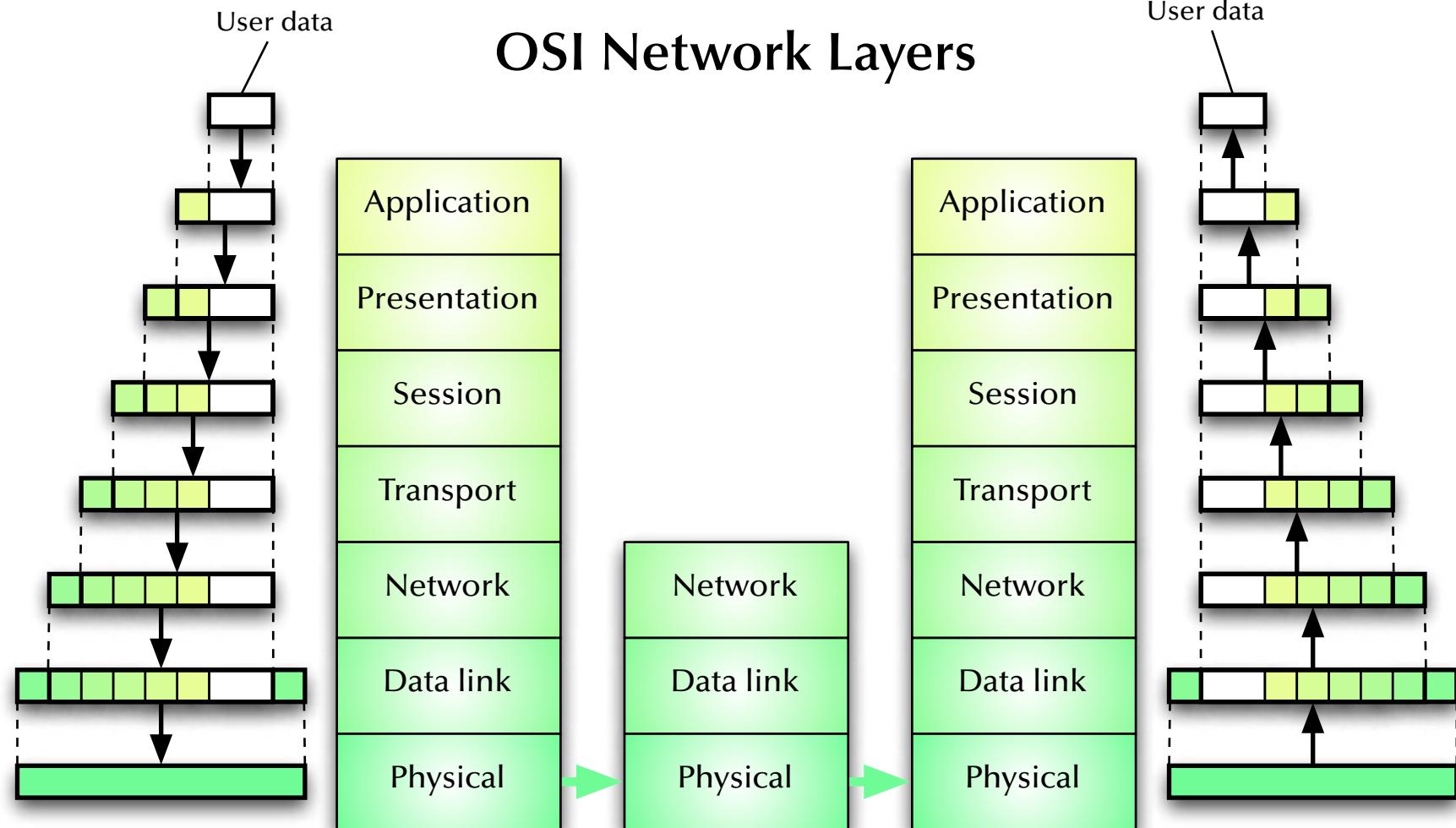
...but its **concepts and terminology** are *widely used*,
when describing existing and designing new protocols ...



Networks

Network protocols & standards

OSI Network Layers

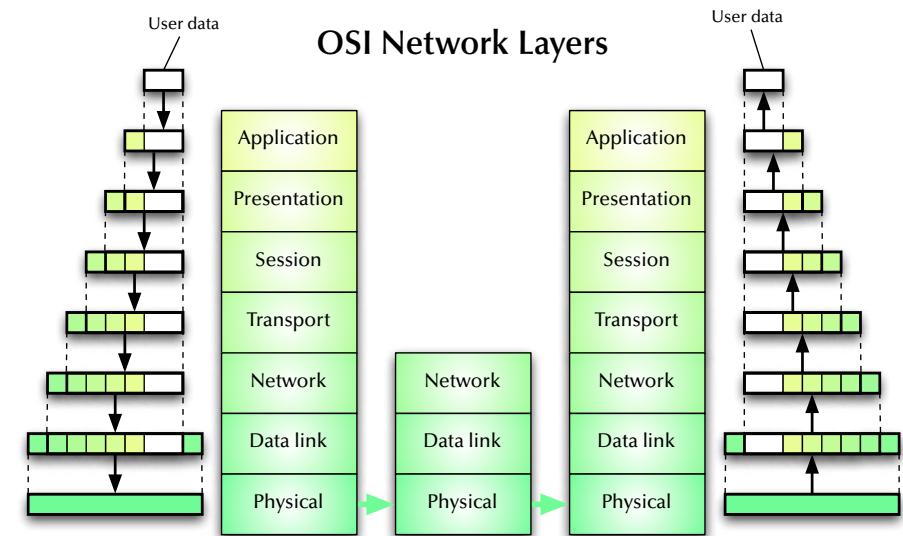




Networks

Network protocols & standards

1: Physical Layer



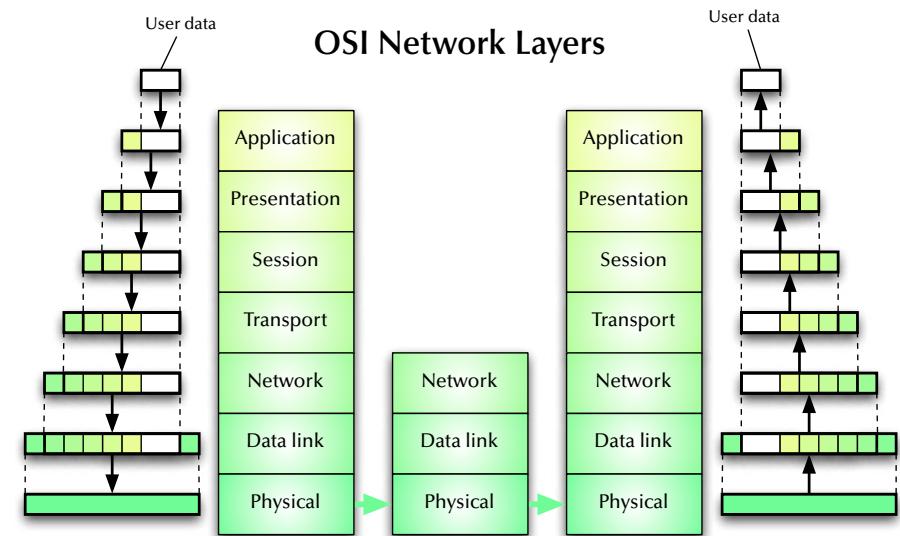
- *Service:* Transmission of a raw bit stream over a communication channel
- *Functions:* Conversion of bits into electrical or optical signals
- *Examples:* X.21, Ethernet (cable, detectors & amplifiers)



Networks

Network protocols & standards

2: Data Link Layer



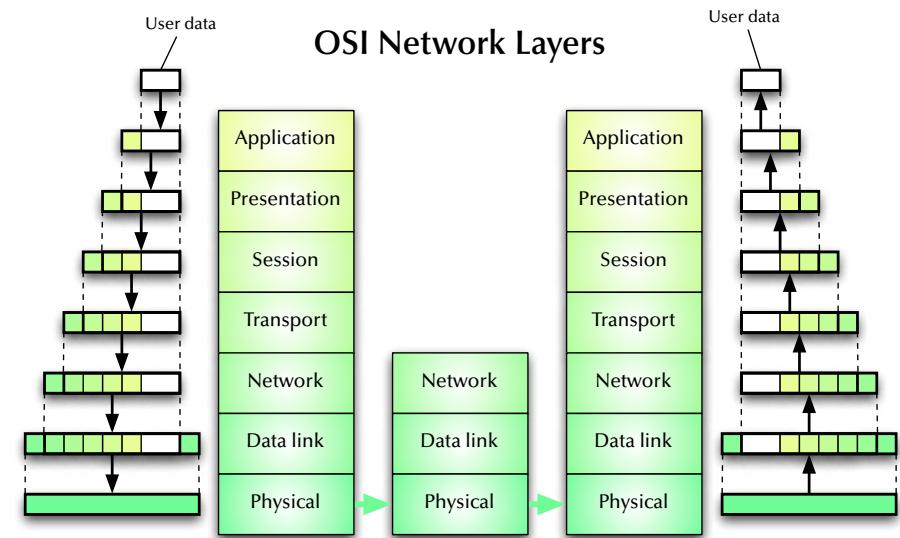
- *Service:* Reliable transfer of frames over a link
- *Functions:* Synchronization, error correction, flow control
- *Examples:* HDLC (high level data link control protocol), LAP-B (link access procedure, balanced), LAP-D (link access procedure, D-channel), LLC (link level control), ...



Networks

Network protocols & standards

3: Network Layer



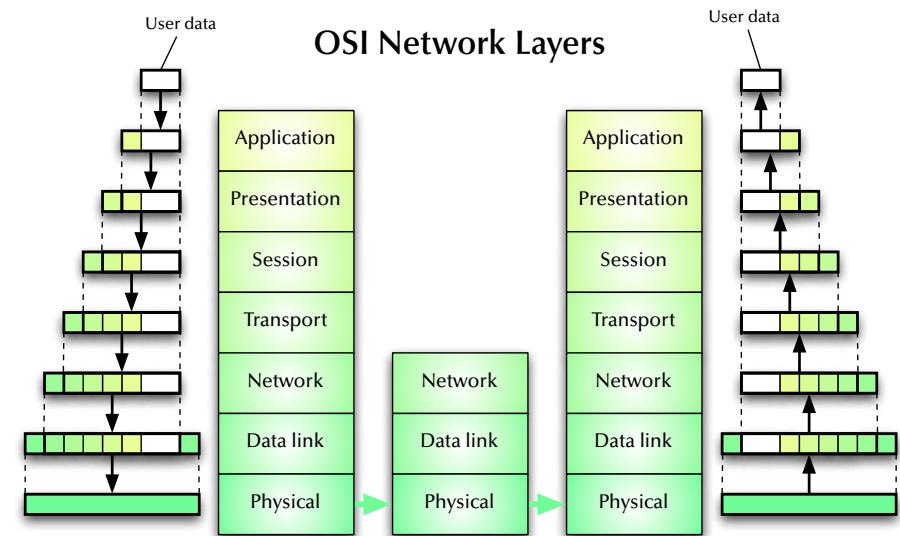
- *Service:* Transfer of packets inside the network
- *Functions:* Routing, addressing, switching, congestion control
- *Examples:* IP, X.25



Networks

Network protocols & standards

4: Transport Layer



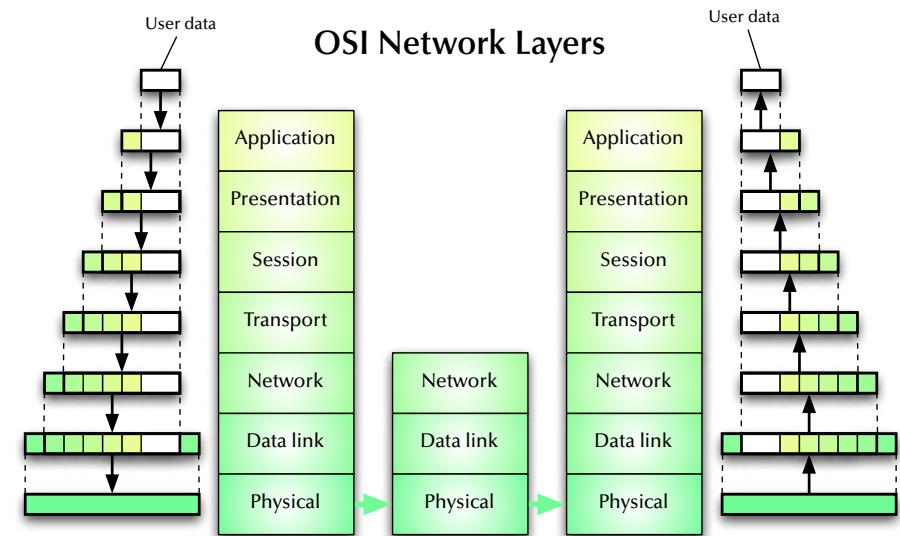
- *Service:* Transfer of data between hosts
- *Functions:* Connection establishment, management, termination, flow-control, multiplexing, error detection
- *Examples:* TCP, UDP, ISO TP0-TP4



Networks

Network protocols & standards

5: Session Layer



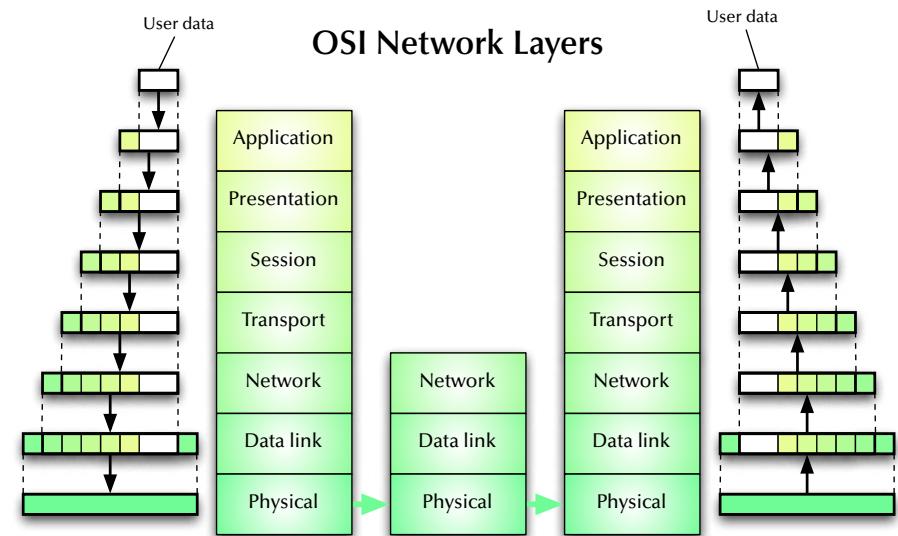
- *Service:* Coordination of the dialogue between application programs
- *Functions:* Session establishment, management, termination
- *Examples:* RPC



Networks

Network protocols & standards

6: Presentation Layer



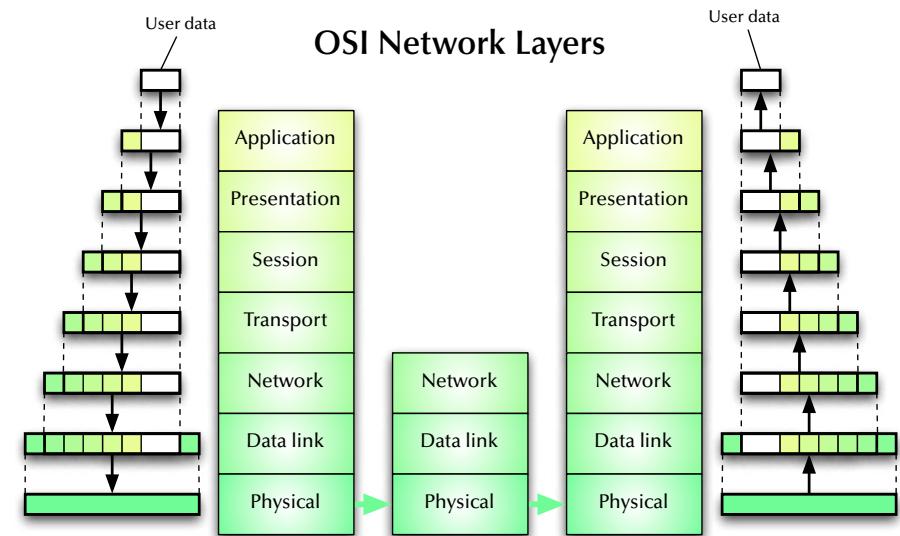
- *Service:* Provision of platform independent coding and encryption
- *Functions:* Code conversion, encryption, virtual devices
- *Examples:* ISO code conversion, PGP encryption



Networks

Network protocols & standards

7: Application Layer

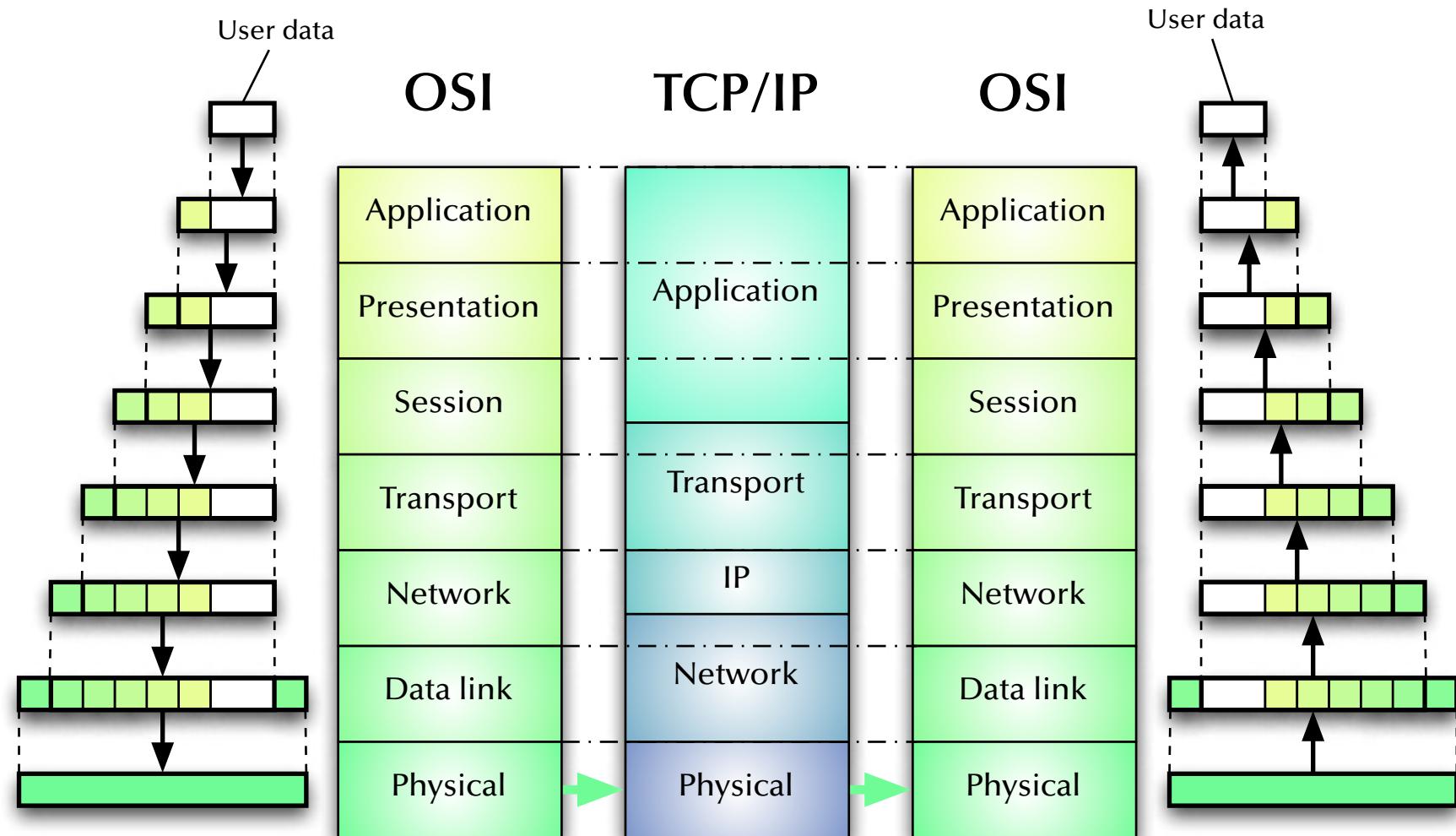


- *Service:* Network access for application programs
- *Functions:* Application/OS specific
- *Examples:* APIs for mail, ftp, ssh, scp, discovery protocols ...



Networks

Network protocols & standards





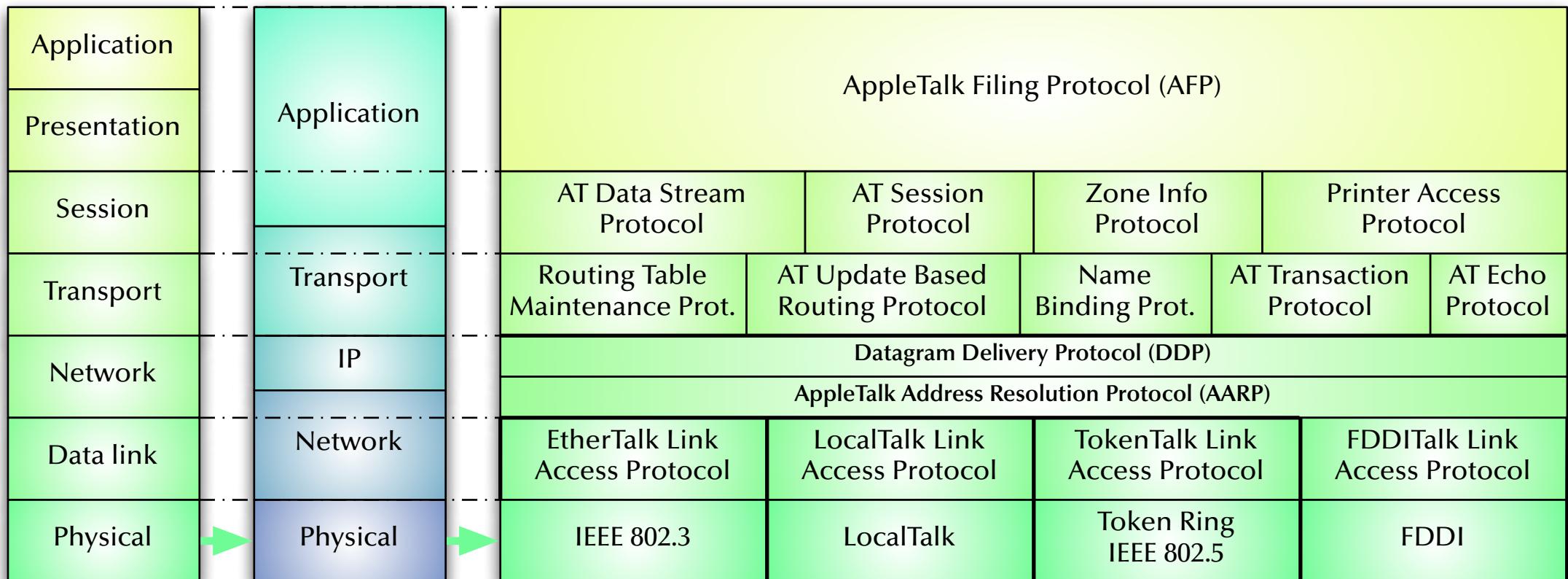
Networks

Network protocols & standards

OSI

TCP/IP

AppleTalk



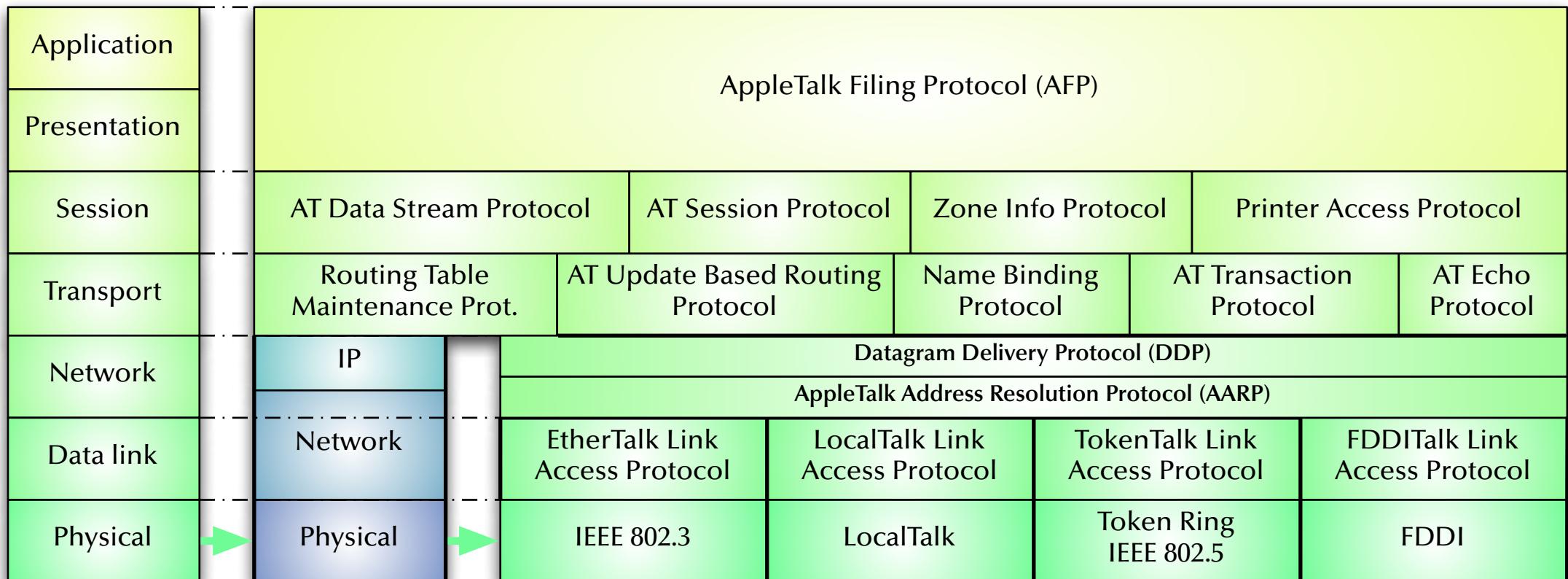


Networks

Network protocols & standards

OSI

AppleTalk over IP





Networks

Network protocols & standards

Serial Peripheral Interface (SPI)

- ☞ Used by gazillions of devices ... and it's not even a formal standard!
- ☞ Speed only limited by what both sides can survive.
- ☞ Usually push-pull drivers, i.e. fast and reliable, yet not friendly to wrong wiring/programming.



1.8" COLOR TFT LCD display from Adafruit



SanDisk marketing photo

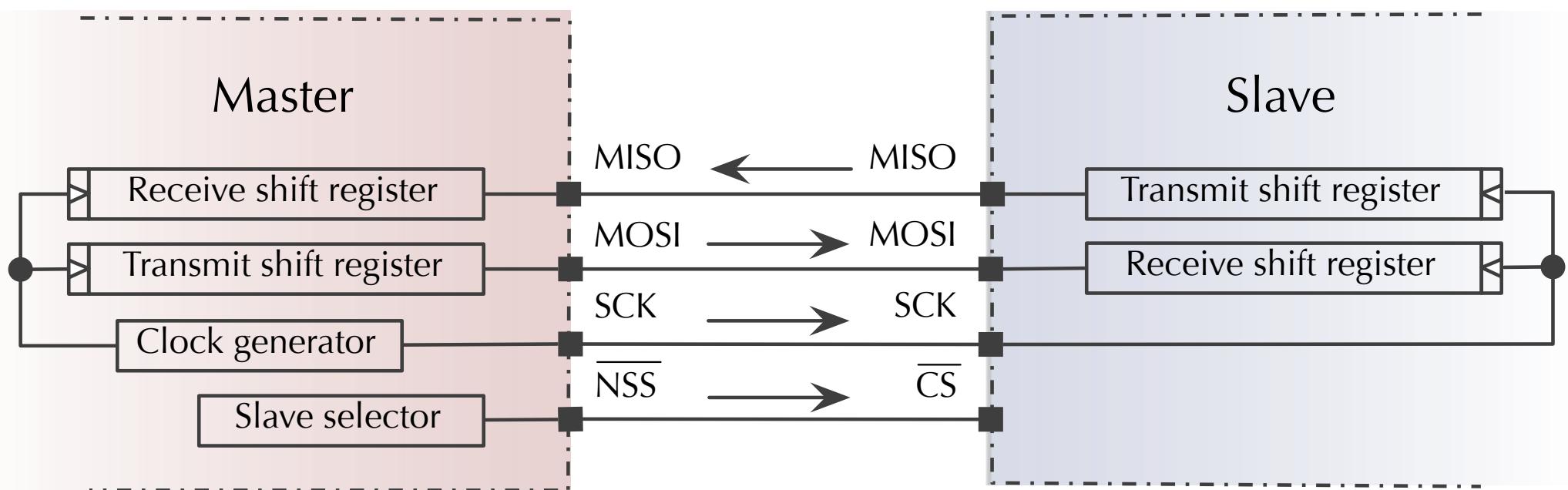


Networks

Network protocols & standards

Serial Peripheral Interface (SPI)

Full Duplex, 4-wire, flexible clock rate

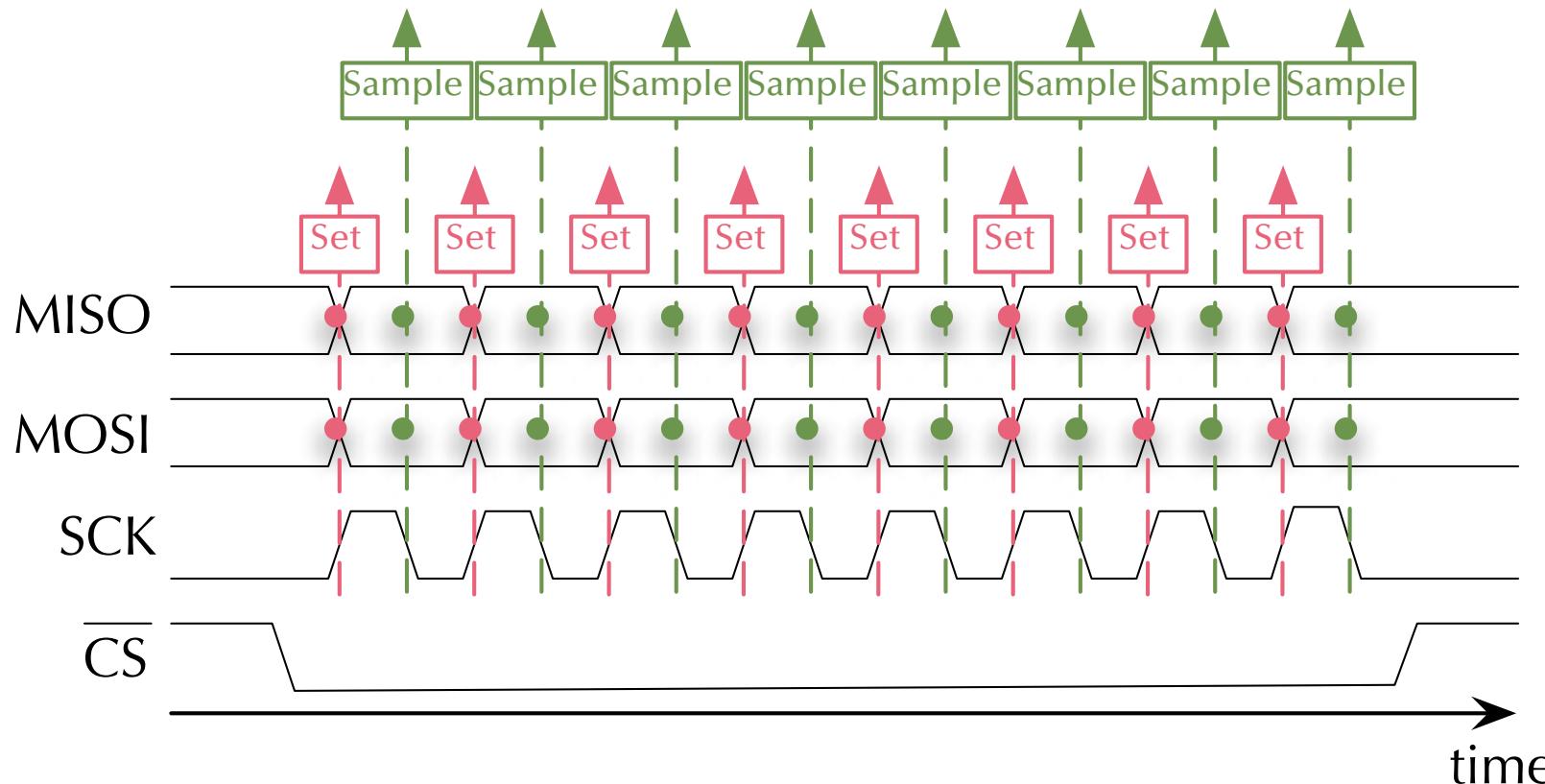
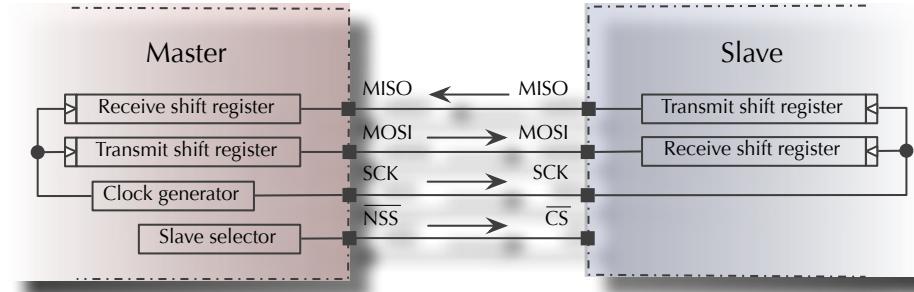




Networks

Network protocols & standards

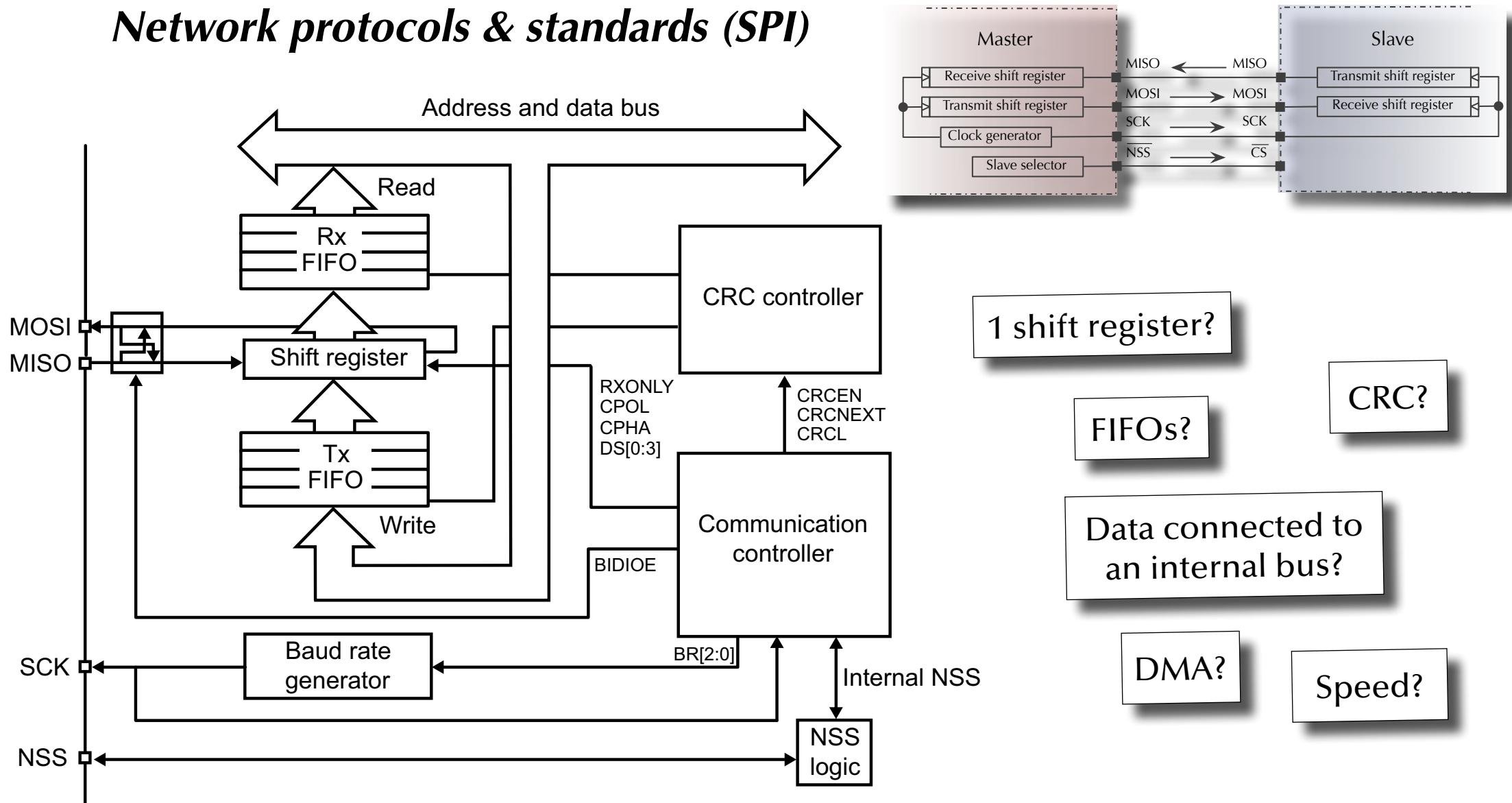
Serial Peripheral Interface (SPI)





Networks

Network protocols & standards (SPI)



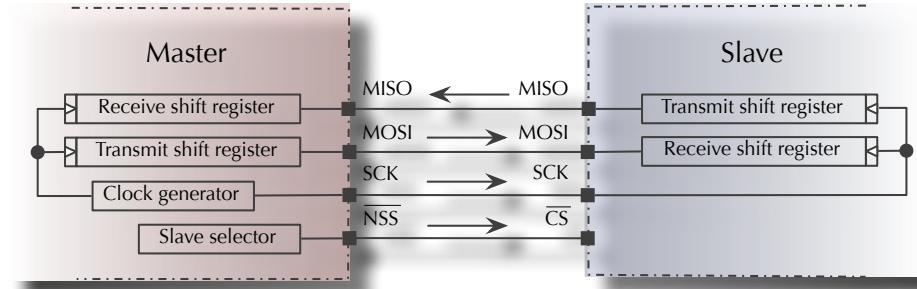
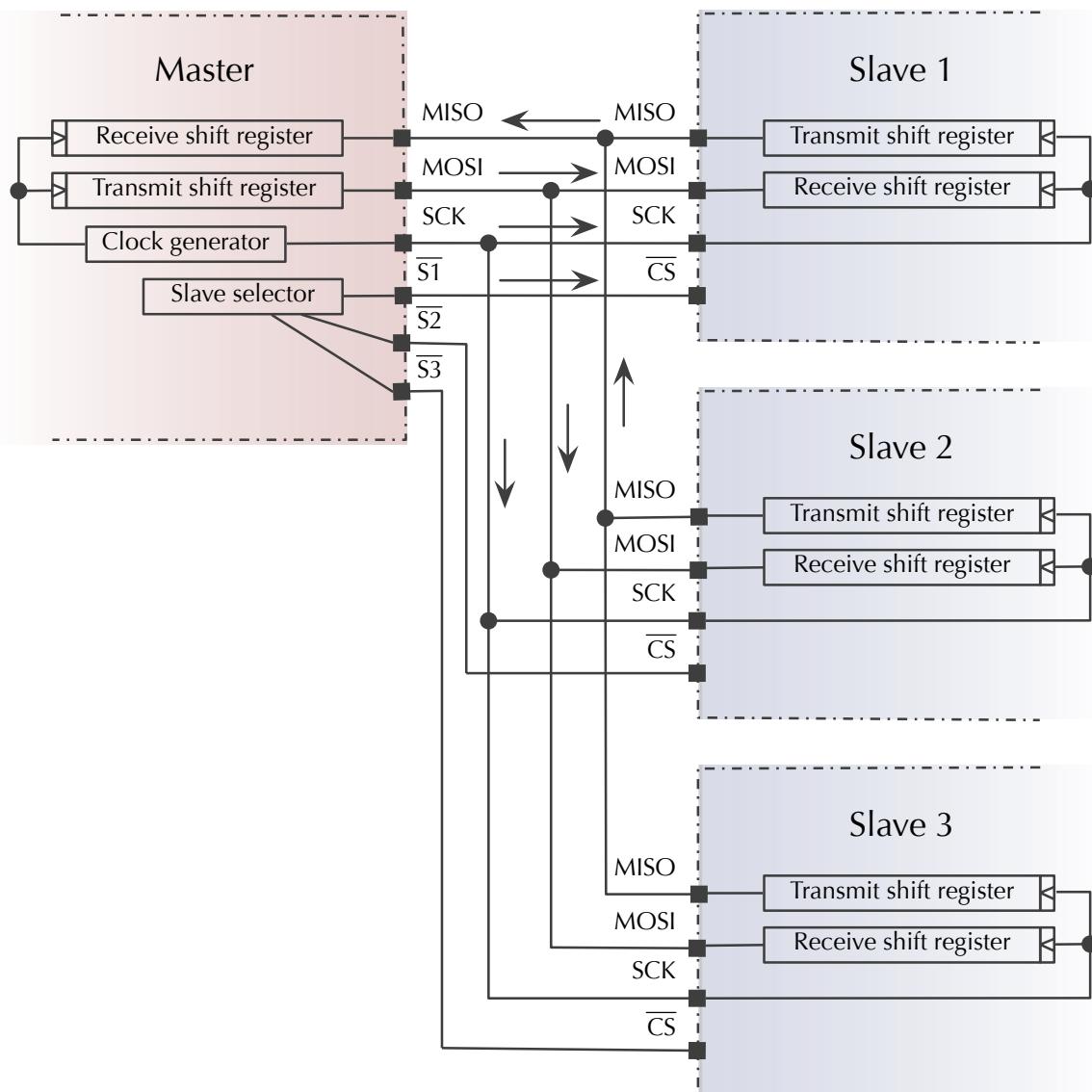
from STM32L4x6 advanced ARM®-based 32-bit MCUs reference manual: Figure 420 on page 1291

MS30117V1



Networks

Network protocols & standards (SPI)

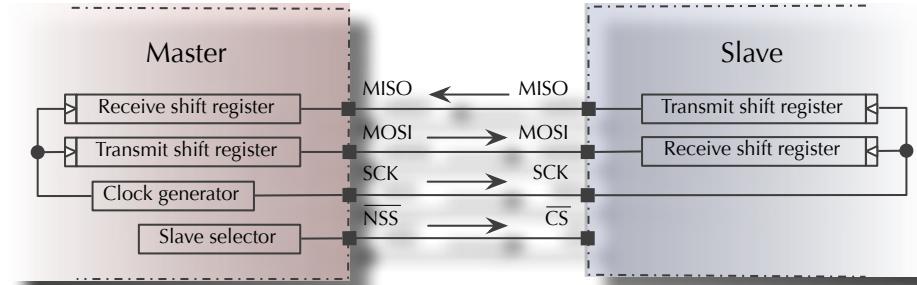
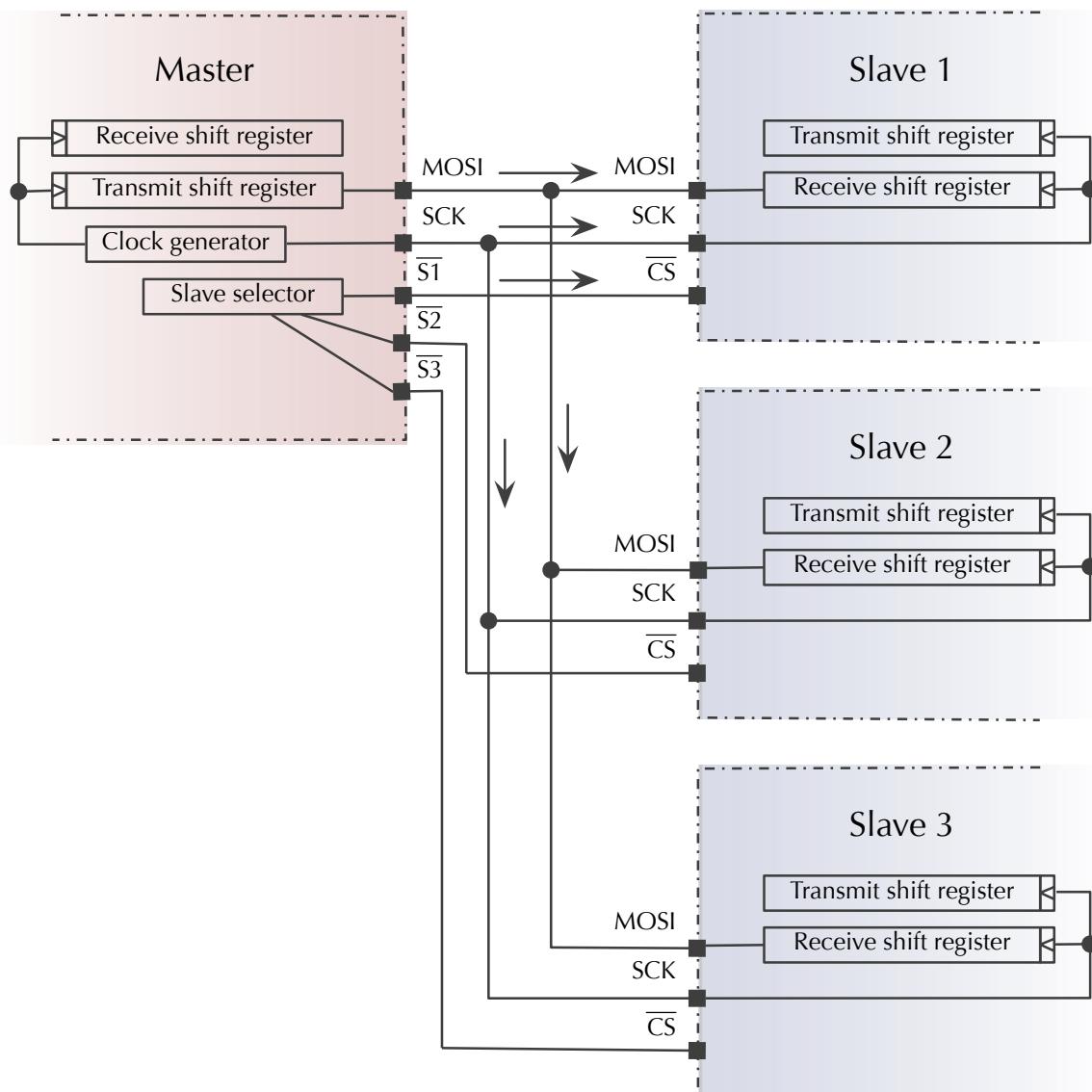


Full duplex with 1
out of x slaves



Networks

Network protocols & standards (SPI)

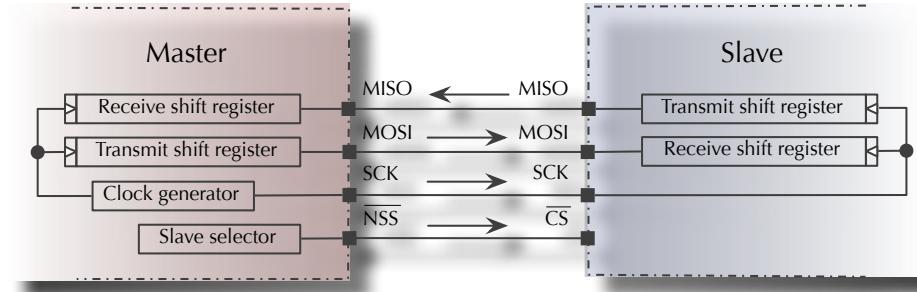
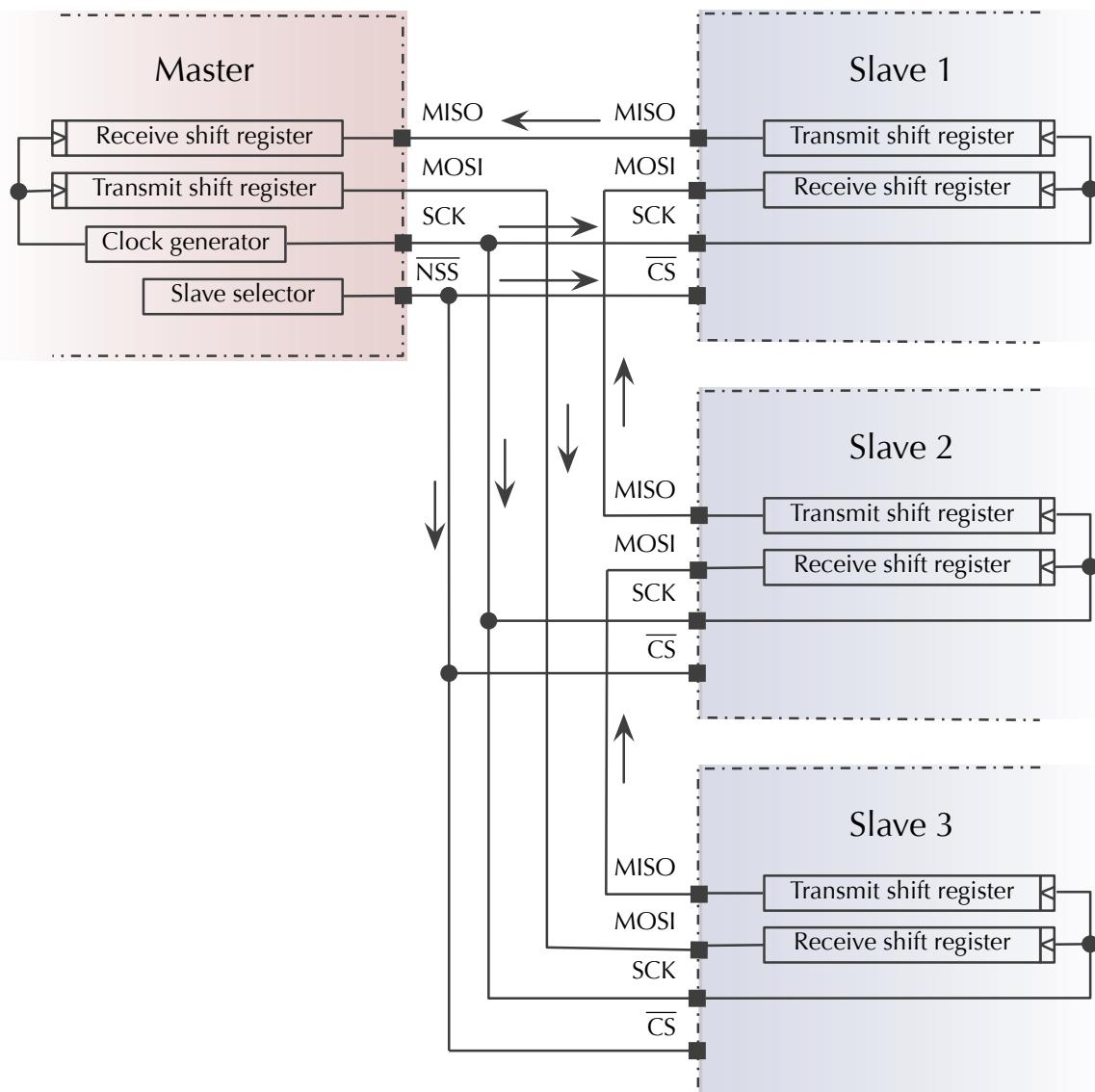


Concurrent simplex
with y out of x slaves



Networks

Network protocols & standards (SPI)



Concurrent
daisy chaining
with all slaves



Networks

Network protocols & standards

Ethernet / IEEE 802.3

Local area network (LAN) developed by Xerox in the 70's

- 10 Mbps specification 1.0 by DEC, Intel, & Xerox in 1980.
- First standard as IEEE 802.3 in 1983 (10 Mbps over thick co-ax cables).
- currently 1 Gbps (802.3ab) copper cable ports used in most desktops and laptops.
- currently standards up to 100 Gbps (IEEE 802.3ba 2010).
- more than 85 % of current LAN lines worldwide
(according to the International Data Corporation (IDC)).

☞ Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

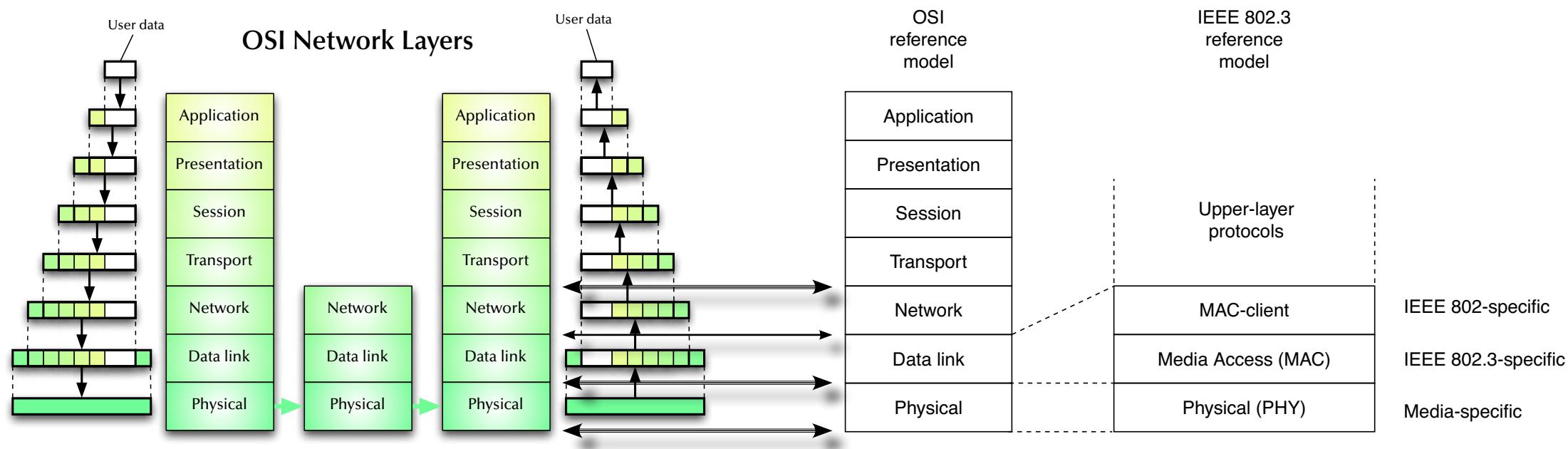


Networks

Network protocols & standards

Ethernet / IEEE 802.3

OSI relation: PHY, MAC, MAC-client



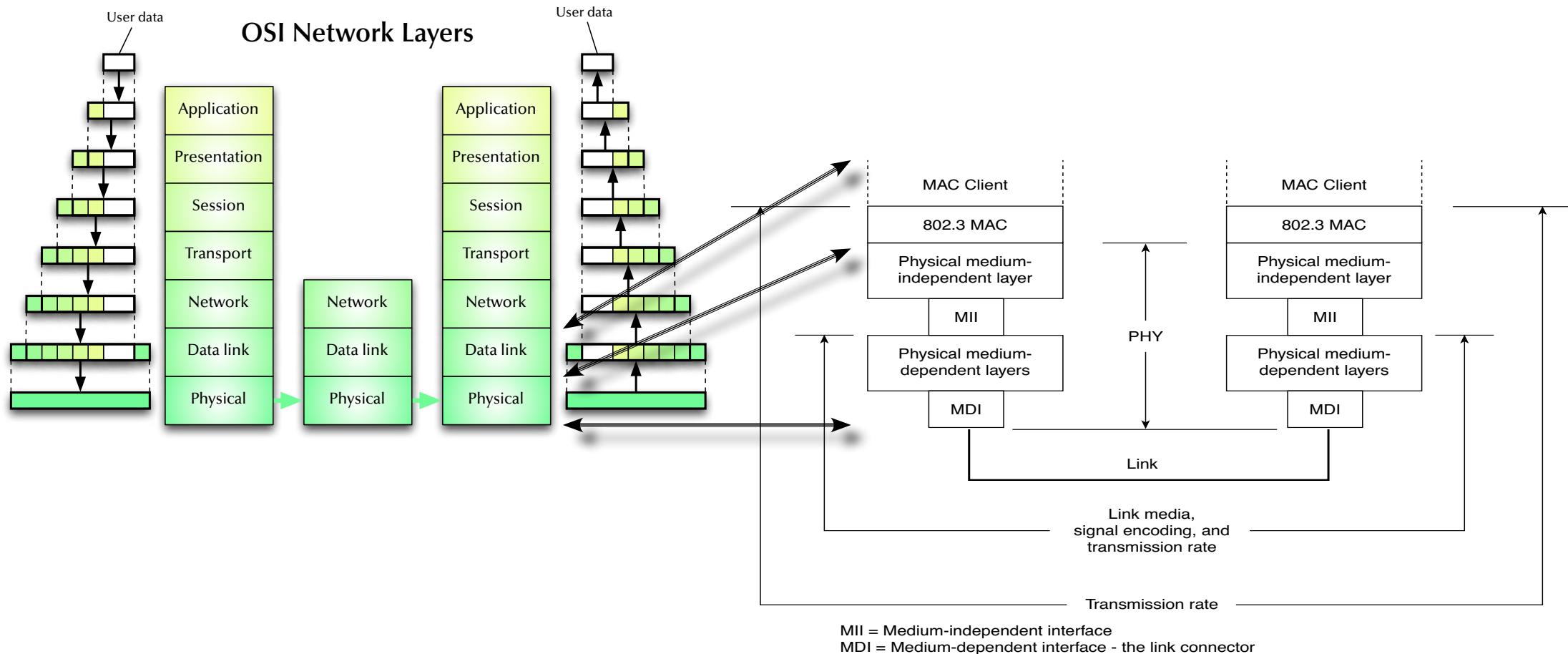


Networks

Network protocols & standards

Ethernet / IEEE 802.3

OSI relation: PHY, MAC, MAC-client





Networks

Network protocols & standards

Ethernet / IEEE 802.11

Wireless local area network (WLAN) developed in the 90's

- First standard as IEEE 802.11 in 1997 (1-2 Mbps over 2.4 GHz).
- Typical usage at 54 Mbps over 2.4 GHz carrier at 20 MHz bandwidth.
- Current standards up to 780 Mbps (802.11ac) over 5 GHz carrier at 160 MHz bandwidth.
- Future standards are designed for up to 100 Gbps over 60 GHz carrier.
- Direct relation to IEEE 802.3 and similar OSI layer association.

☞ **Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)**

☞ **Direct-Sequence Spread Spectrum (DSSS)**



Networks

Network protocols & standards

Bluetooth

Wireless local area network (WLAN) developed in the 90's with different features than 802.11:

- Lower power consumption.
- Shorter ranges.
- Lower data rates (typically < 1 Mbps).
- Ad-hoc networking (no infrastructure required).

☞ Combinations of 802.11 and Bluetooth OSI layers are possible to achieve the required features set.



Networks

Network protocols & standards

Token Ring / IEEE 802.5 / Fibre Distributed Data Interface (FDDI)

- “Token Ring” developed by IBM in the 70’s
- IEEE 802.5 standard is modelled after the IBM Token Ring architecture (specifications are slightly different, but basically compatible)
- IBM Token Ring requests are star topology as well as twisted pair cables, while IEEE 802.5 is unspecified in topology and medium
- Fibre Distributed Data Interface combines a token ring architecture with a dual-ring, fibre-optical, physical network.

👉 **Unlike CSMA/CD, Token ring is deterministic**
(with respect to its timing behaviour)

👉 **FDDI is deterministic and failure resistant**

👉 None of the above is currently used in performance oriented applications.



Networks

Network protocols & standards

Fibre Channel

- Developed in the late 80's.
- ANSI standard since 1994.
- Current standards allow for 16 Gbps per link.
- Allows for three different topologies:
 - ☞ **Point-to-point:** 2 addresses
 - ☞ **Arbitrated loop** (similar to token ring): 127 addresses ☞ deterministic, real-time capable
 - ☞ **Switched fabric:** 2^{24} addresses, many topologies and concurrent data links possible
- Defines OSI equivalent layers up to the session level.
- ☞ Mostly used in storage arrays,
but applicable to super-computers and high integrity systems as well.

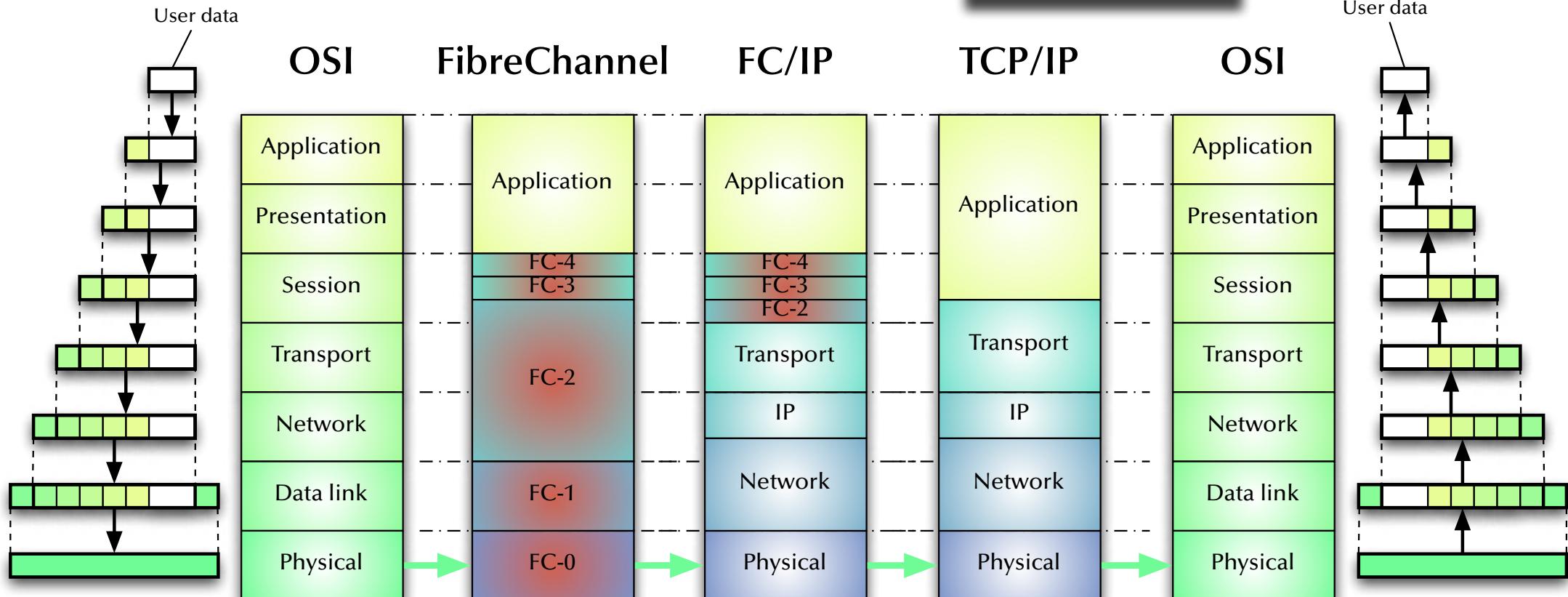
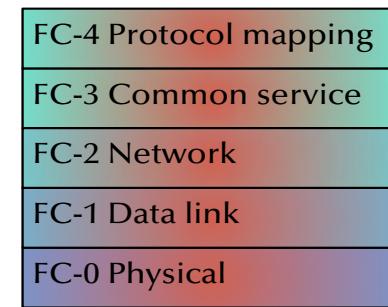


Networks

Network protocols & standards

Fibre Channel

Mapping of Fibre Channel to OSI layers:





Networks

Summary

Networks

- **Network layer models**

- Open Systems Interconnection (OSI) reference model

- **Practical network standards**

- Serial Peripheral Interface (SPI)
- Ethernet / IEEE 802.3 (CSMA/CD)
- Tokenring / IEEE 802.5 / FDDI
- Wireless networks / IEEE 802.11 (CSMA/CA, DSSS)
- Fibre Channel

Computer Organisation & Program Execution 2021



9

Architecture

Uwe R. Zimmer - The Australian National University



Architecture

References for this chapter

[Patterson17]

David A. Patterson & John L. Hennessy

Computer Organization and Design – The Hardware/Software Interface

Chapter 4 “The Processor”,

Chapter 6 “Parallel Processors from Client to Cloud”

ARM edition, Morgan Kaufmann 2017



Architecture





Architecture



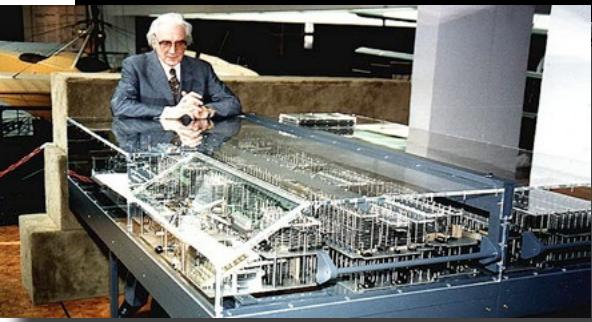
Definition: Processor
Hardware origins
18th century machines

L'Ecrivain
1770
Programmable,
yet not a computer in today's
definition (not Turing complete)

L'Ecrivain (1770)
Pierre Jaquet-Droz, Henri-Louis Jaquet-Droz & Jean-Frédéric Lescho



Architecture

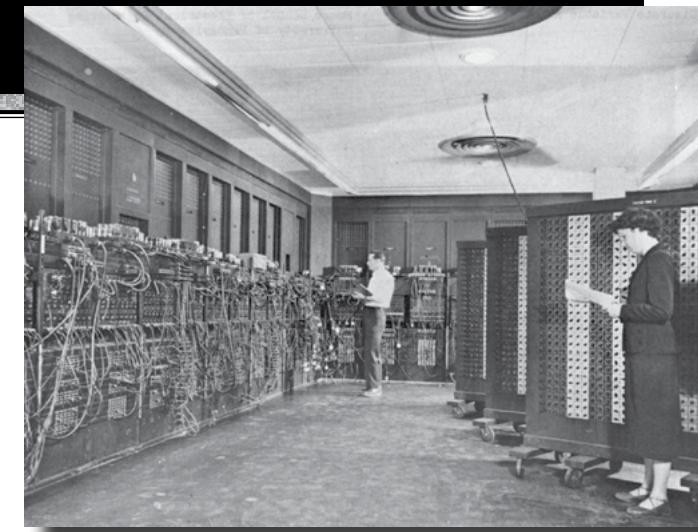


Konrad Zuse with Z1, © Dr. Horst Zuse
(replica of the 1937 computer)

Definition: Processor *Digital Computers*

Hardware origins

- **Patents by Konrad Zuse (Germany), 1936.**
- *First digital computer: Z1 (Germany), 1937:* Relays, programmable via punch tape, clock: 1 Hz, 64 words memory à 22-bit, 2 registers, floating point unit, weight: 1 t.
- *First freely programmable (Turing complete) relays computer: Z3 (Germany), 1941:* 5.3 Hz
- **Atanasoff Berry Computer (US) 1942:** Vacuum tubes, (not Turing complete).
- **Colossus Mark 1 (UK) 1944:** Vacuum tubes (not Turing complete).
- “*First Draft of a Report on the EDVAC*” (*Electronic Discrete Variable Automatic Computer*) by **John von Neumann** (US), 1945: Influential article about core elements of a computer: **Arithmetic unit, control unit** (Sequencer), **memory** (holding data and program), and **I/O**.
- First high level programming language: **Plankalkül** (“Plan Calculus”) by Konrad Zuse, 1945.
- **ENIAC (Electronic Numerical Integrator And Computer) (US) 1946:** programmed by plugboard, *First Turing complete vacuum tubes based computer, clock: 100 kHz, weight: 27 t on 167 m².*



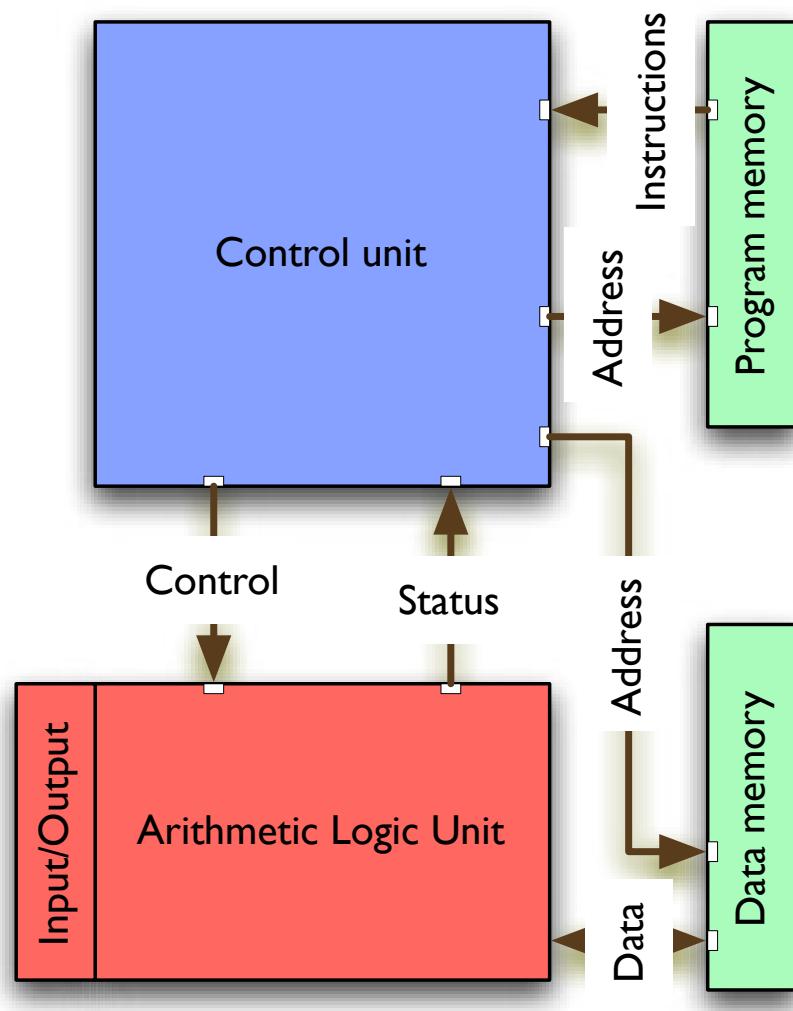
ENIAC 1946,
Glen Beck (background), Betty Jennings (foreground)



Architecture

Computer Architectures

Harvard Architecture

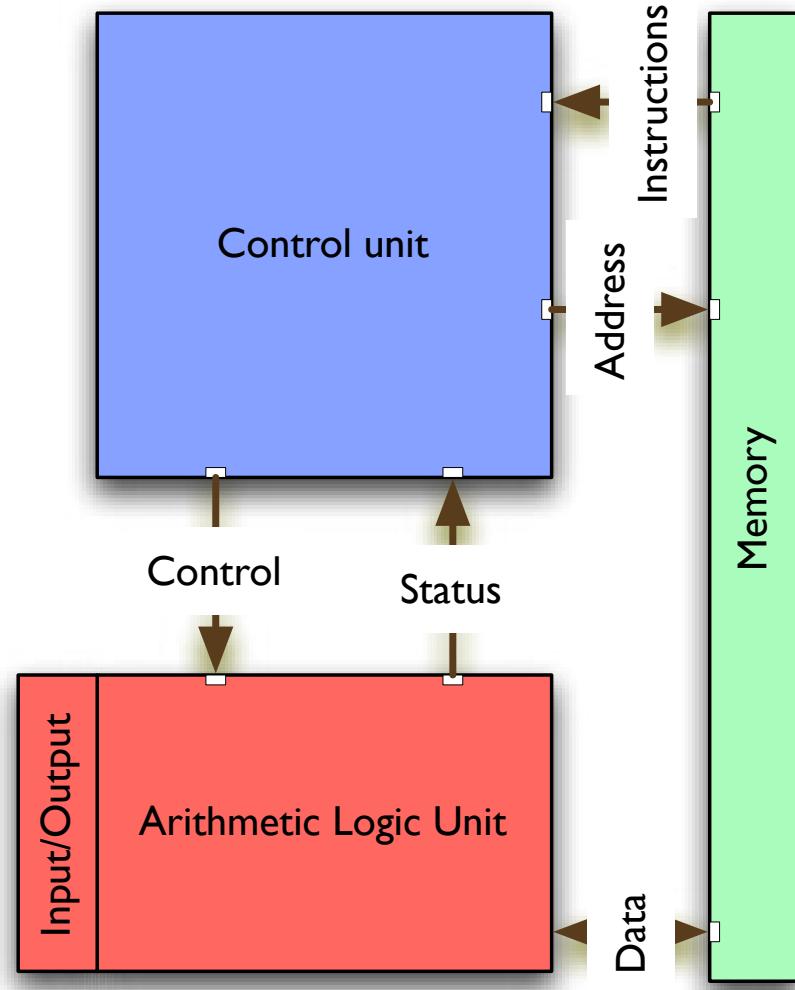


- **Control unit**
Concurrently addresses program and data memory and fetches next instruction.
Controls next ALU operations and determines the next instruction (based on ALU status).
- **Arithmetic Logic Unit (ALU)**
Fetches data from memory.
Executes arithmetic/logic operation.
Writes data to memory.
- **Input/Output**
- **Program memory**
- **Data memory**



Architecture

Computer Architectures



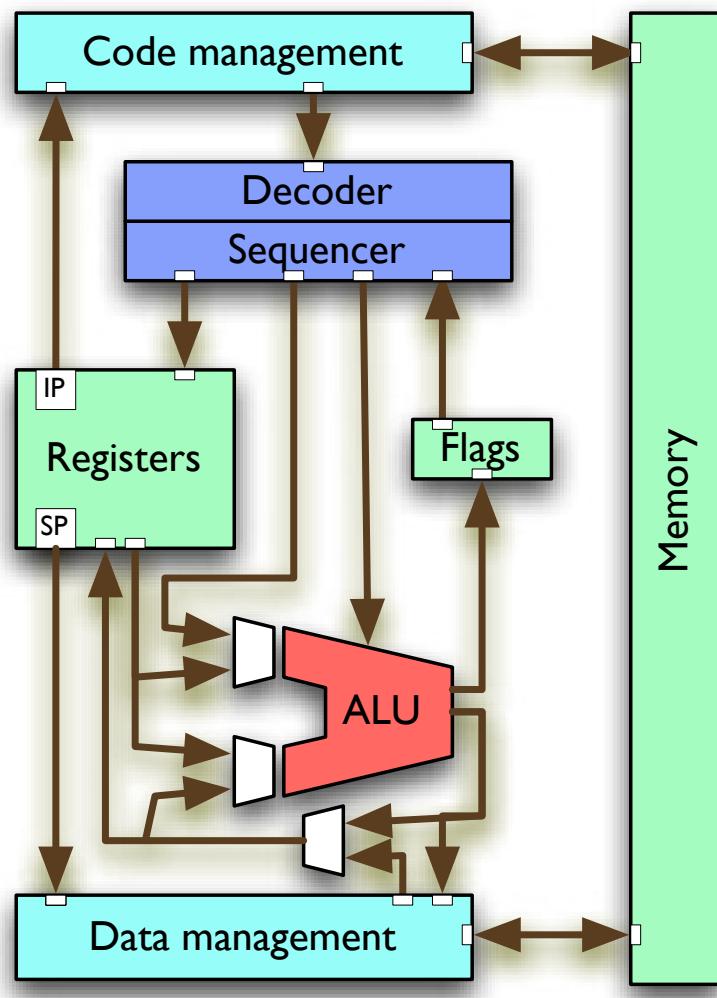
von Neumann Architecture

- **Control unit**
Sequentially addresses program and data memory and fetches next instruction.
Controls next ALU operations and determines the next instruction (based on ALU status).
- **Arithmetic Logic Unit (ALU)**
Fetches data from memory.
Executes arithmetic/logic operation.
Writes data to memory.
- **Input/Output**
- **Memory**
Program and data is not distinguished
☞ Programs can change themselves.



Architecture

Computer Architectures



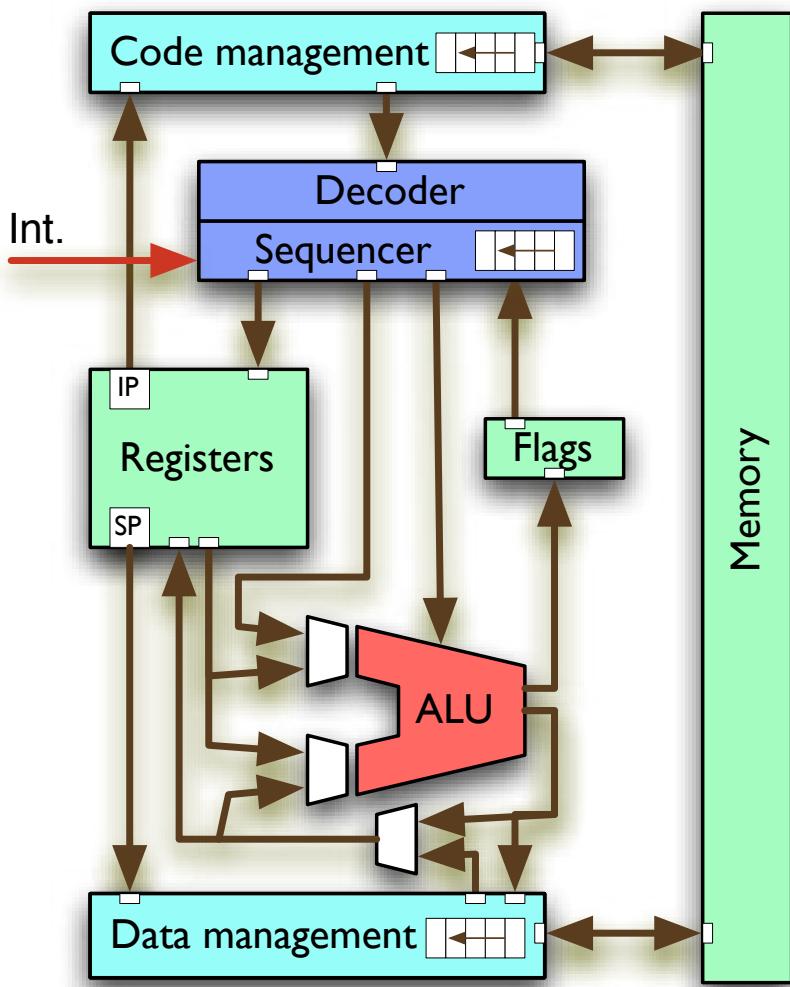
A simple processor (CPU)

- **Decoder/Sequencer**
Can be a machine in itself which breaks CPU instructions into *concurrent micro code*.
- **Execution Unit / Arithmetic-Logic-Unit (ALU)**
A collection of transformational logic.
- **Memory**
- **Registers**
Instruction pointer, stack pointer, general purpose and specialized registers.
- **Flags**
Indicating the states of the latest calculations.
- **Code/Data management**
Fetching, Caching, Storing.



Architecture

Processor Architectures



Pipeline

Some CPU actions are naturally sequential (e.g. instructions need to be first loaded, then decoded before they can be executed).

More fine grained sequences can be introduced by breaking CPU instructions into micro code.

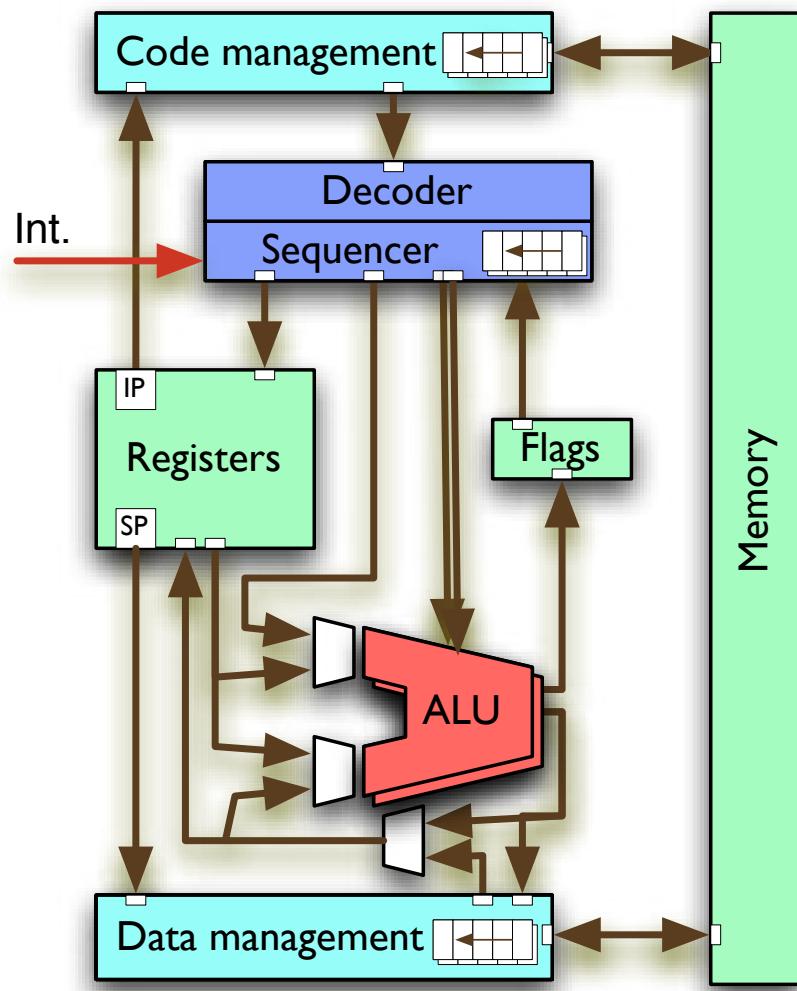
- ☞ Overlapping those sequences in time will lead to the concept of pipelines.
- ☞ Same latency, yet higher throughput.
- ☞ (Conditional) branches might break the pipelines
- ☞ Branch predictors become essential.



Architecture

Processor Architectures

Parallel pipelines



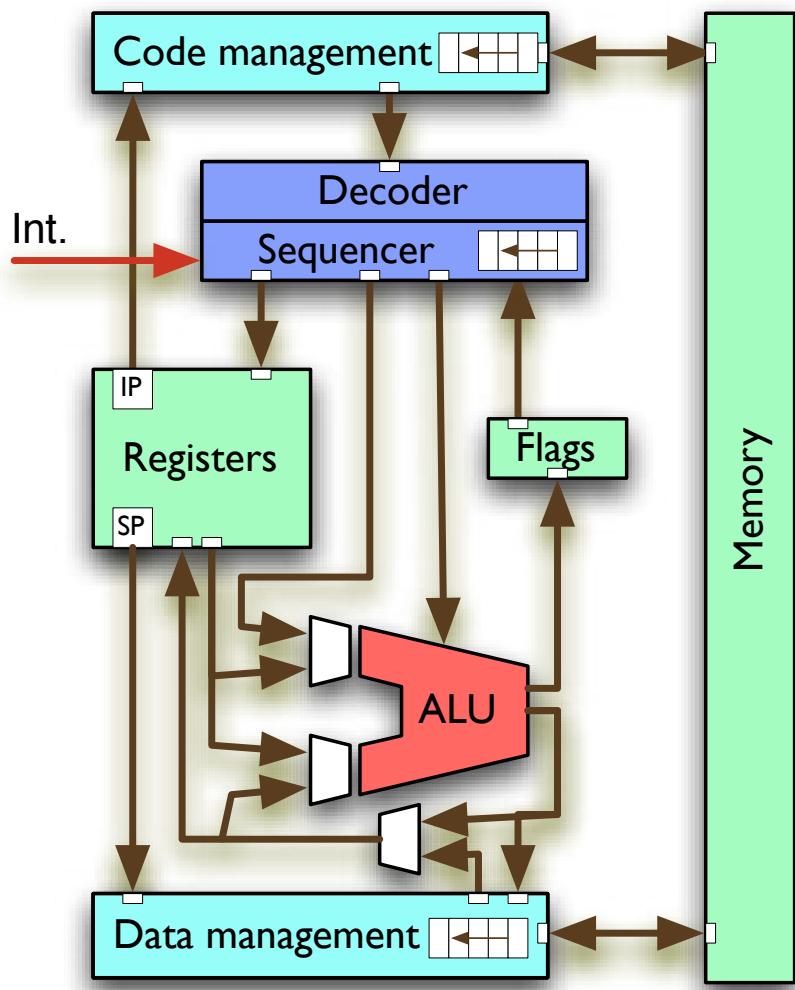
Filling parallel pipelines
(by alternating incoming commands between
pipelines) may employ multiple ALU's.

- ☞ (Conditional) branches might again break the pipelines.
- ☞ Interdependencies might limit the degree of concurrency.
- ☞ Same latency, yet even higher throughput.
- ☞ Compilers need to be aware of the options.



Architecture

Processor Architectures



Pipeline hazards

Structural hazard

- ☞ Lack of **hardware** to run operations in parallel,
... e.g. load an new instruction and load new data in parallel.

Control hazard

- ☞ A **decision** depends on the previous instruction.
... e.g. a conditional branch based on the flags from the previous instruction.

Data hazard

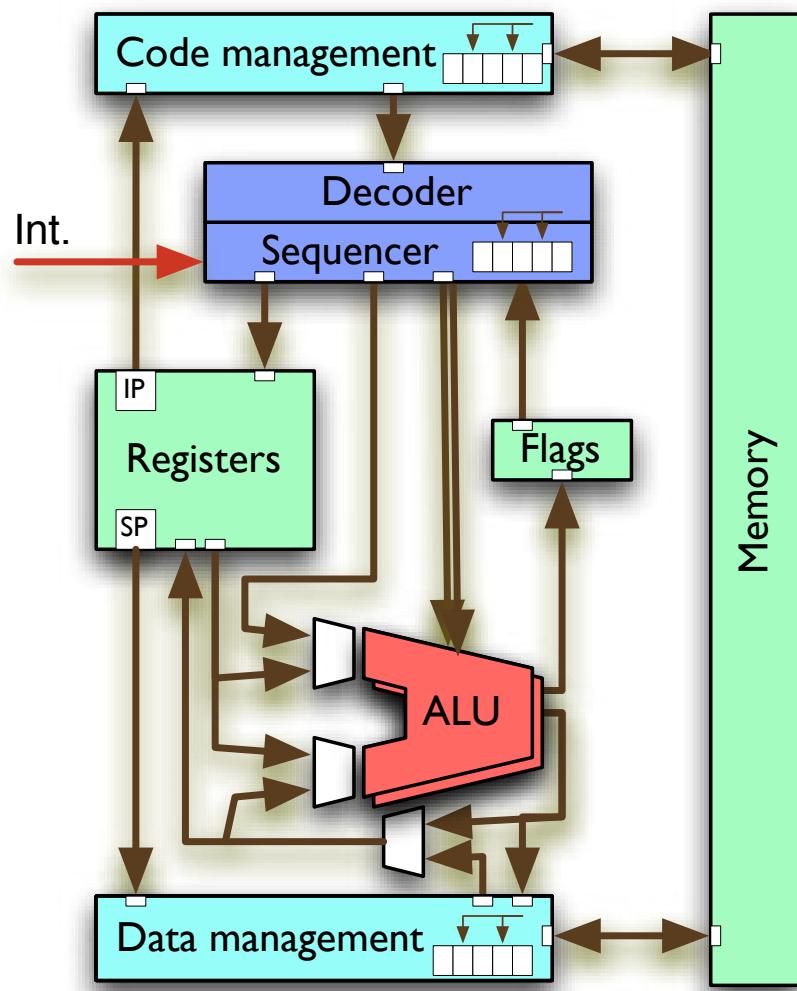
- ☞ Needed **data** is not yet available
... e.g. the result of an arithmetic operation is needed in the next instruction.



Architecture

Processor Architectures

Out of order execution



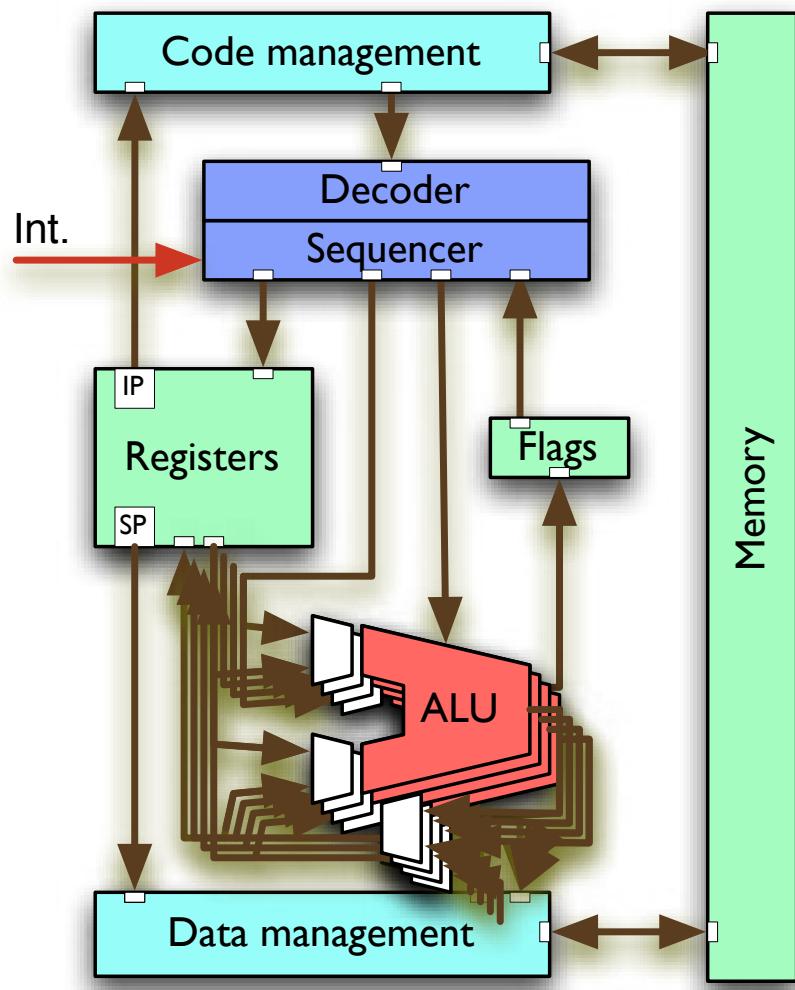
Breaking the sequence inside each pipeline leads to ‘out of order’ CPU designs.

- ☞ Replace pipelines with hardware scheduler.
- ☞ Results need to be “re-sequentialized” or possibly discarded.
- ☞ “Conditional branch prediction” executes the most likely branch or multiple branches.
- ☞ Works better if the presented code sequence has more independent instructions and fewer conditional branches.
- ☞ This hardware will require (extensive) code optimization to be fully utilized.



Architecture

Processor Architectures



SIMD ALU units

Provides the facility to apply the same instruction to multiple data concurrently.
Also referred to as “vector units”.

Examples: Altivec, MMX, SSE[2|3|4], ...

☞ Requires specialized compilers or programming languages with implicit concurrency.

GPU processing

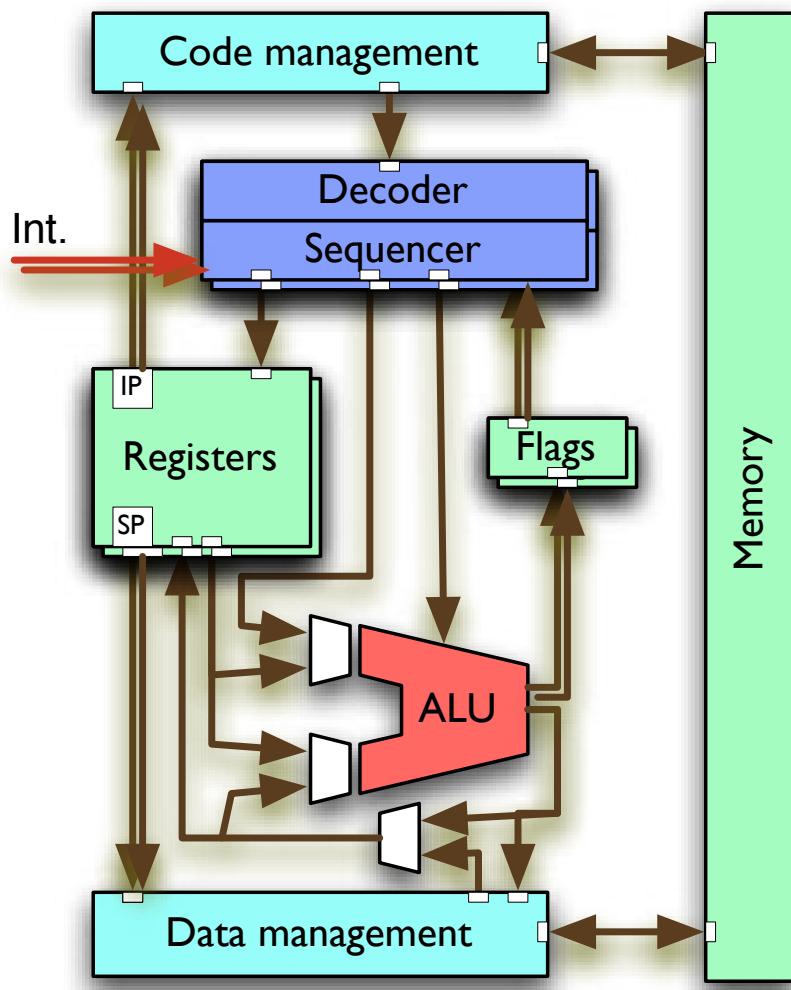
Graphics processor as a vector unit.

☞ Unifying architecture languages are used (OpenCL, CUDA, GPGPU).



Architecture

Processor Architectures



Hyper-threading

Emulates multiple virtual CPU cores by means of replication of:

- Register sets
- Sequencer
- Flags
- Interrupt logic

while keeping the “expensive” resources like the ALU central yet accessible by multiple hyper-threads concurrently.

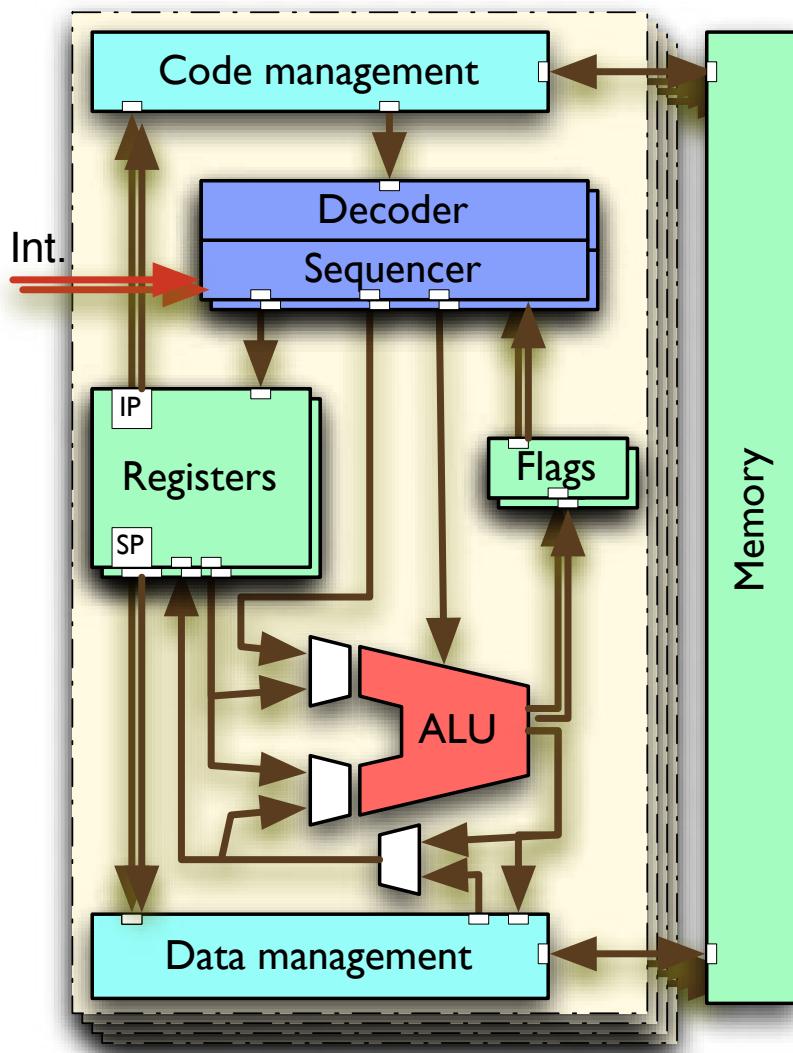
☞ Requires programming languages with implicit or explicit concurrency.

Examples: Intel Pentium 4, Core i5/i7, Xeon, Atom, Sun UltraSPARC T2 (8 threads per core)



Architecture

Processor Architectures



Multi-core CPUs

Full replication of multiple CPU cores on the same chip package.

- Often combined with hyper-threading and/or multiple other means (as introduced above) on each core.
- Cleanest and most explicit implementation of concurrency on the CPU level.

☞ Requires synchronized atomic operations.

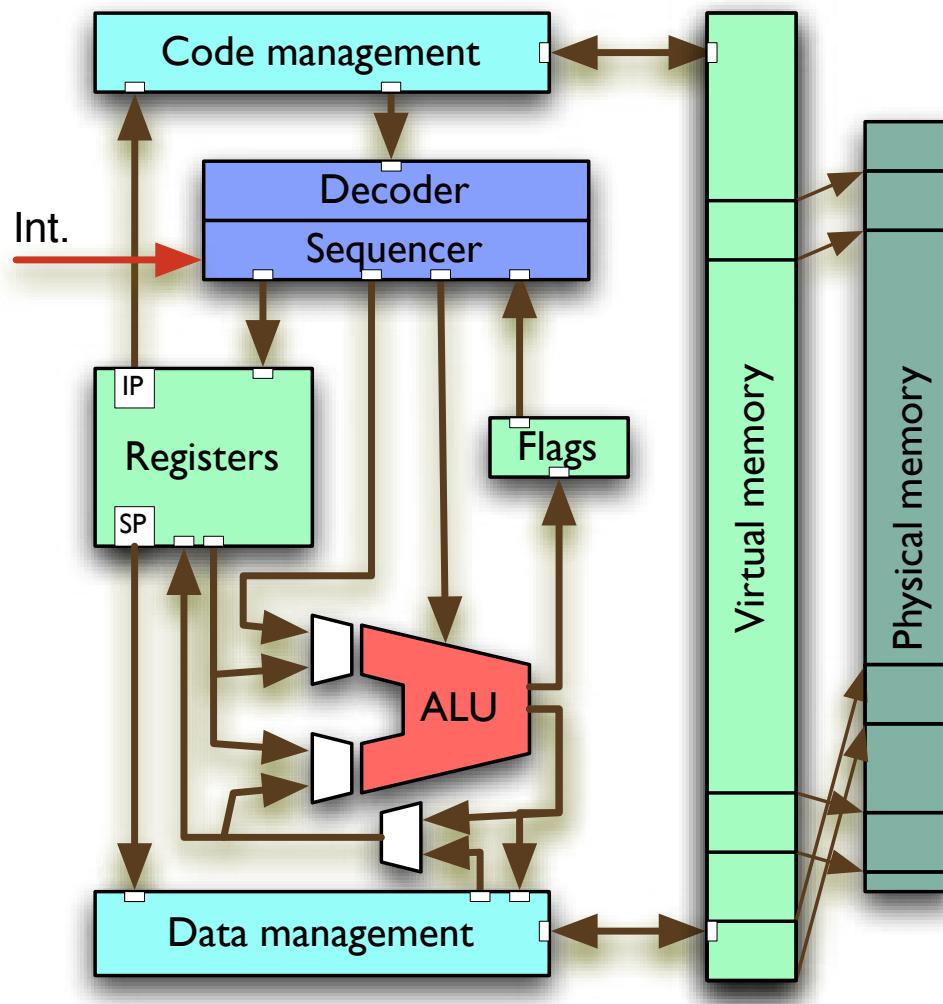
☞ Requires programming languages with implicit or explicit concurrency.

Historically the introduction of multi-core CPUs ended the “GHz race” in the early 2000’s.



Architecture

Processor Architectures



Virtual memory

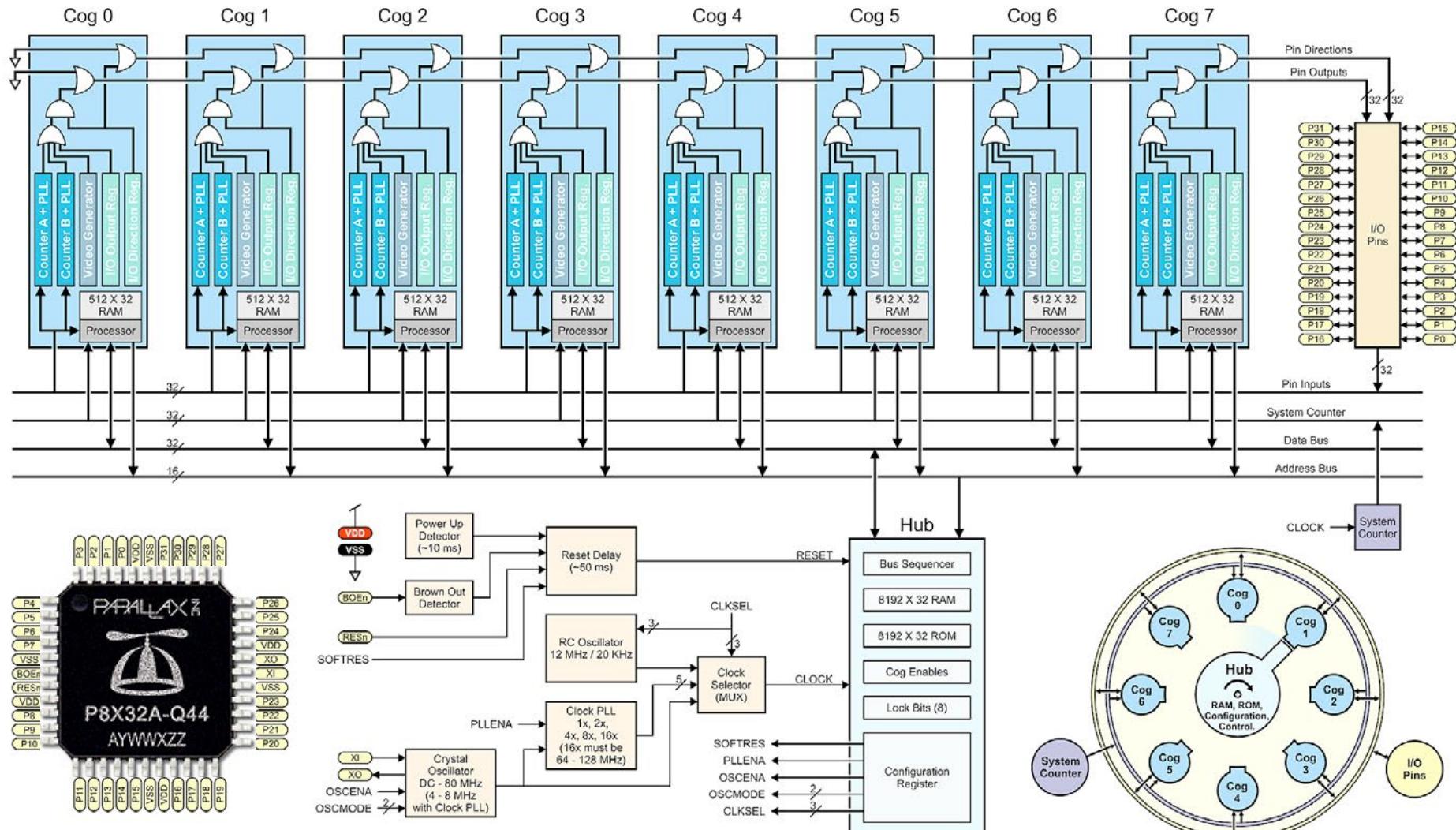
Translates logical memory addresses into physical memory addresses and provides memory protection features.

- Does not introduce concurrency by itself.
- ☞ Is still essential for concurrent programming as hardware memory protection guarantees memory integrity for individual processes / threads.



Architecture

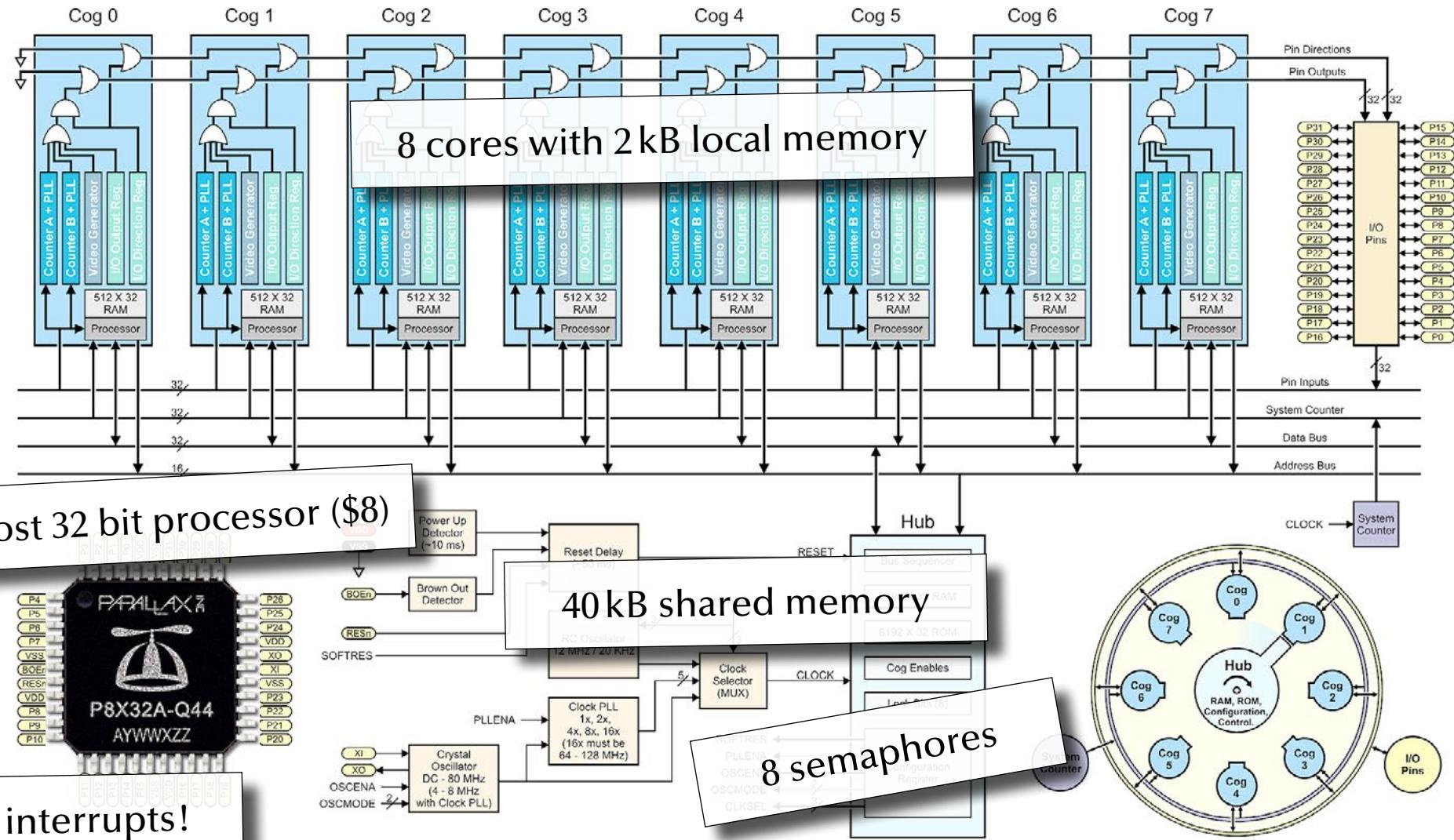
Alternative Processor Architectures: Parallax Propeller





Architecture

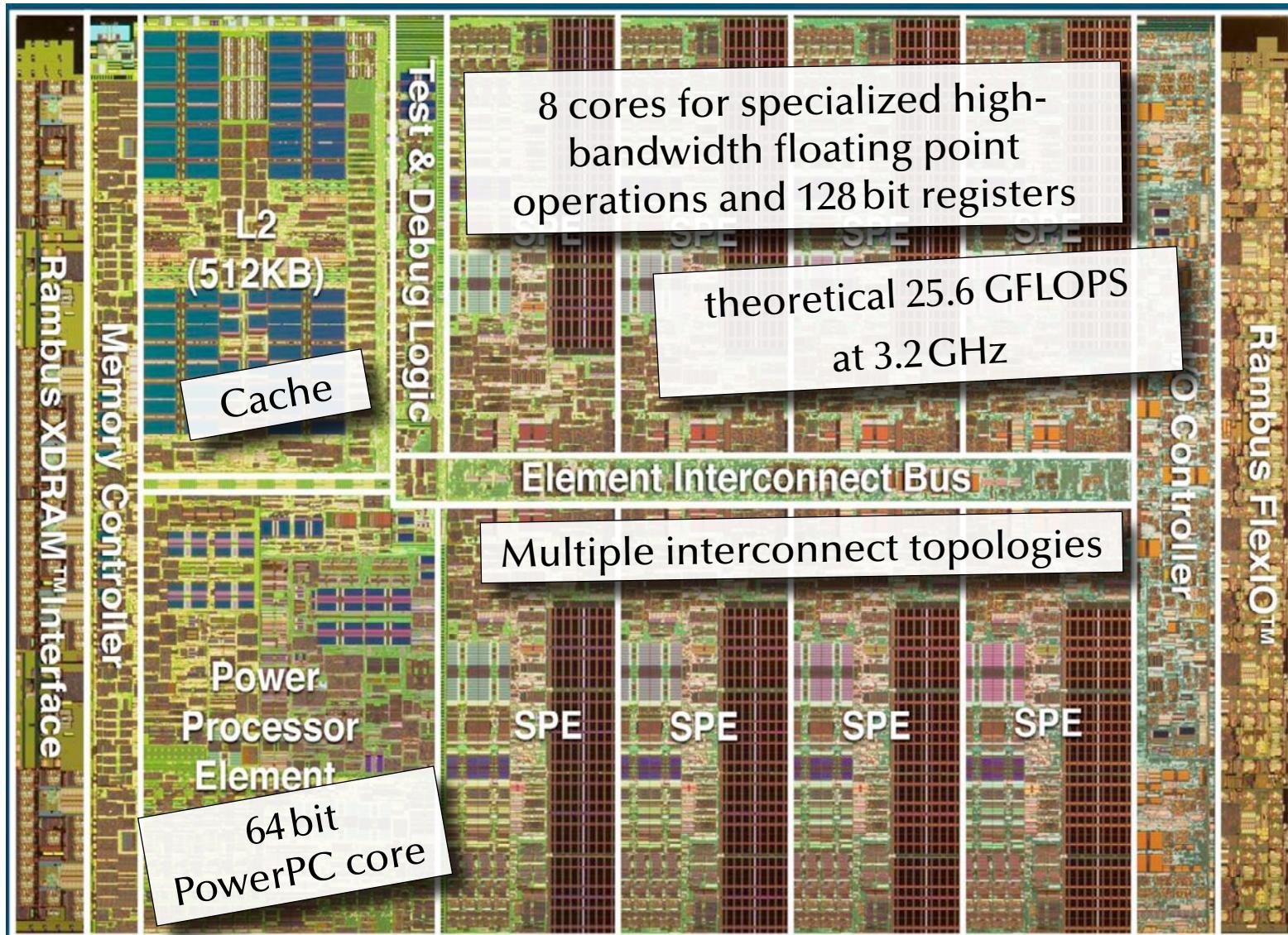
Alternative Processor Architectures: Parallax Propeller (2006)





Architecture

Alternative Processor Architectures: IBM Cell processor (2001)





Architecture

Multi-CPU systems

Scaling up:

- Multi-CPU on the same memory
 - multiple CPUs on same motherboard and memory bus, e.g. servers, workstations
- Multi-CPU with high-speed interconnects
 - various supercomputer architectures, e.g. Cray XE6:
 - 12-core AMD Opteron, up to 192 per cabinet (2304 cores)
 - 3D torus interconnect (160 GB/sec capacity, 48 ports per node)
- Cluster computer (Multi-CPU over network)
 - multiple computers connected by network interface,
e.g. Sun Constellation Cluster at ANU:
 - 1492 nodes, each: 2x Quad core Intel Nehalem, 24 GB RAM
 - QDR Infiniband network, 2.6 GB/sec





Architecture

Summary

Architecture

- **History**
- **Architectures**
 - Pipelines
 - Parallel pipelines
 - Out of order execution
 - Vector machines
 - Multi-core CPUs
 - Virtual memory



10

Summary

Uwe R. Zimmer - The Australian National University



Summary

Exam preparations

Helpful

- **Distinguish** central aspects from excursions, examples & implementations.
- **Gain** full understanding of all central aspects.
- Be able to **categorize** any given example under a general theme discussed in the lecture.
- **Explain** to and **discuss** the topics with other (preferably better) students.
- Try whether you can **connect** aspects from different parts of the lecture.

Not helpful

- Remembering the slides word by word.
- Learn the ARM reference manuals page by page.

