

Accelerators and Coarse Grained Reconfigurable Architectures



Mark Wijtvliet
m.wijtvliet@tue.nl

General purpose processors

- **Run mixed application types**
 - Operating system
 - Webserver
 - Games
 - Office applications
- **Cheap**
 - E.g. Raspberry Pi's Broadcom processor
 - Many others
- **Generally easy to program**
 - Well developed compilers and tool-flow.
 - Programming model hardly changed over the years.



The Cortex A9 RK3188 quad-core tablet processor

Price: **US \$10.00** / piece
Bulk Price ▼

Shipping: **US \$5.41** to Netherlands via China Post Registered
Estimated Delivery Time: 15-39 days (ships out within 24 hours)

Quantity: piece (997 pieces available)

Total Price: **US \$15.41**

Buy Now

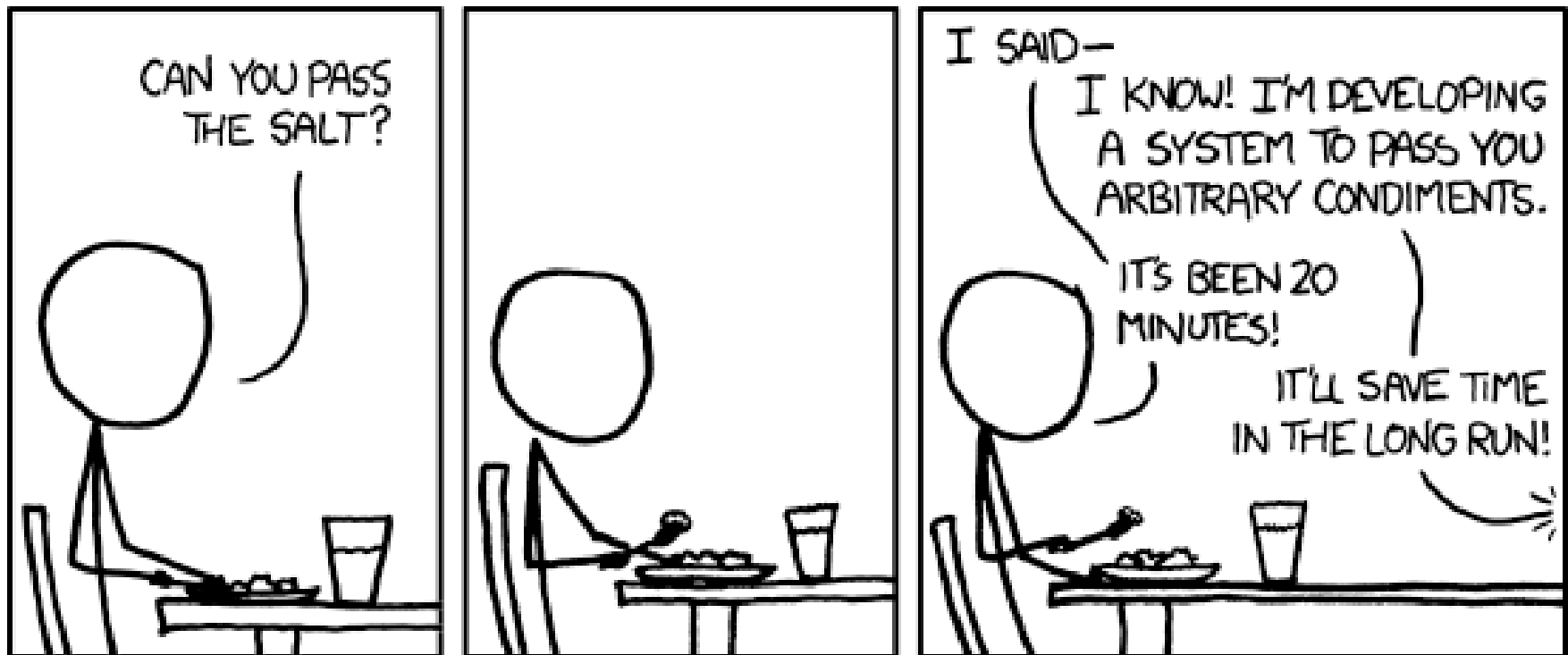
Add to Cart

♥ Add to Wish List ▼ (1 Adds) ?



General purpose processors

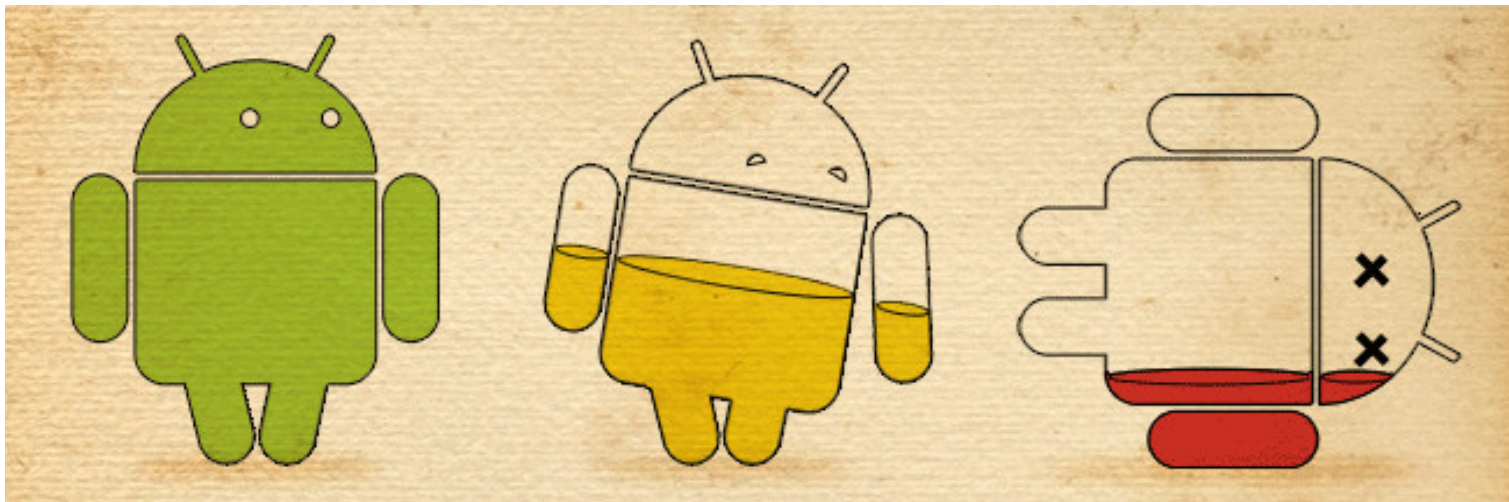
- They work great
- So why this accelerator talk?



[XKCD.com]

Example: mobile phones

- Typically a power budget of 1 Watt (1 J/Sec).
- Modern communication systems (4G) require 1000 GOPS
- That requires a compute efficiency of 1 pJ/OP



Example: mobile phones

- (Older) ARM11: 200 pJ/OP (65nm)
- A modern ARM, in 28nm
 - Scaling is $1/(S^2)$
 - Should be around 37 pJ/OP
 - Numbers not public



Example: mobile phones

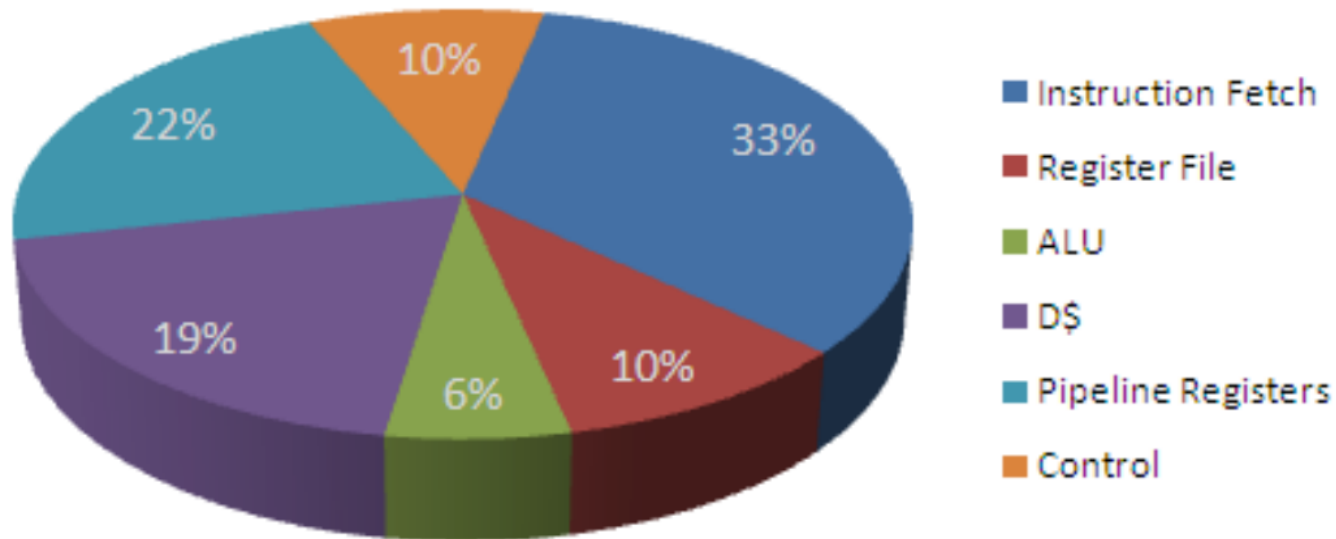
- Another problem:
 - Quad-core ARM at 2 GHz: approx. 8 GOPS
 - Orders of magnitude too low performance
 - ... And it would be nice if your phone can do something else than just 4G.



Example: mobile phones

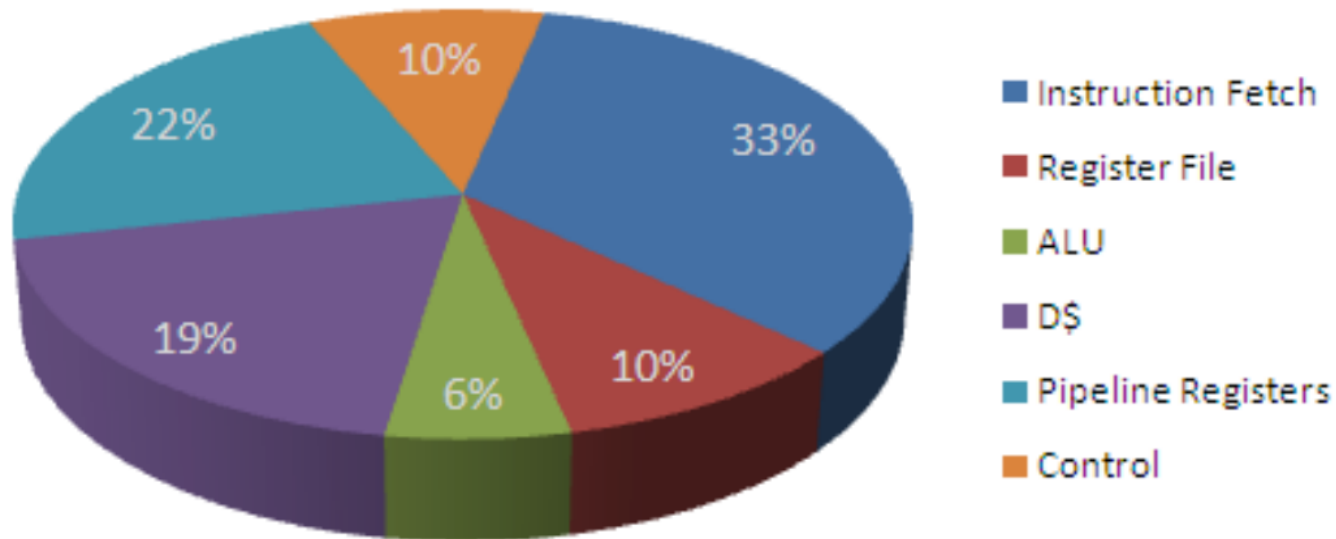
- Quick summary:
 - Compute efficiency is (at least) one order of magnitude off from 1 pJ/OP.
 - Performance is off by several orders of magnitude.

Inefficiencies in processors



[Understanding sources of inefficiency in General-Purpose Chips, Hameed et al.]

Inefficiencies in processors



- Instruction fetching and decoding
- Communication (register file, caches, etc.)
- Hardware reconfiguration (in processor pipeline)

[Understanding sources of inefficiency in General-Purpose Chips, Hameed et al.]

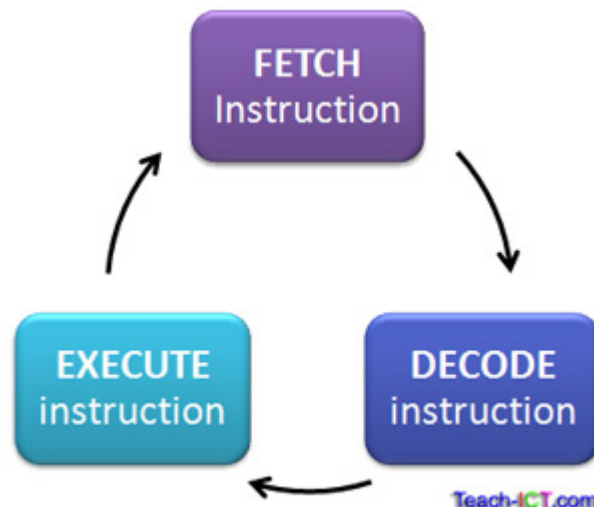
Inefficiencies in processors

- Factor 500 difference to ASIC in energy.
 - For H.264 encoding
 - For a 2.8 GHz Pentium 4 and a Tensilica Micro-processor.



Instruction fetching and decoding

- Where does the overhead come from?
 - Addressing and loading the instruction word from memory.
 - Instruction caches.
 - Decoding the instruction to decoded instruction bits that control the processing pipeline.

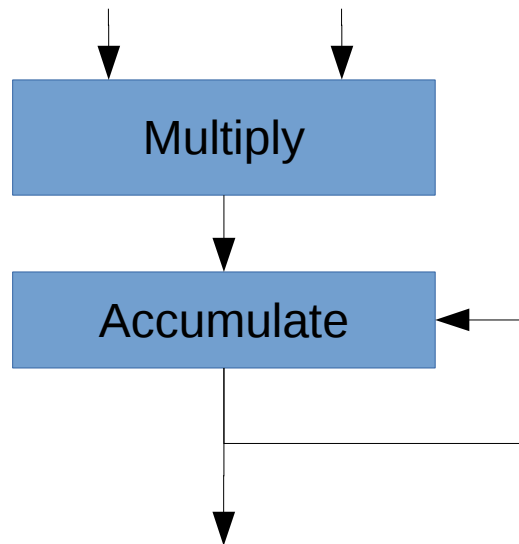


Data transport

- Where does the overhead come from?
 - Register file access
 - How do processors reduce this?
 - Data caches

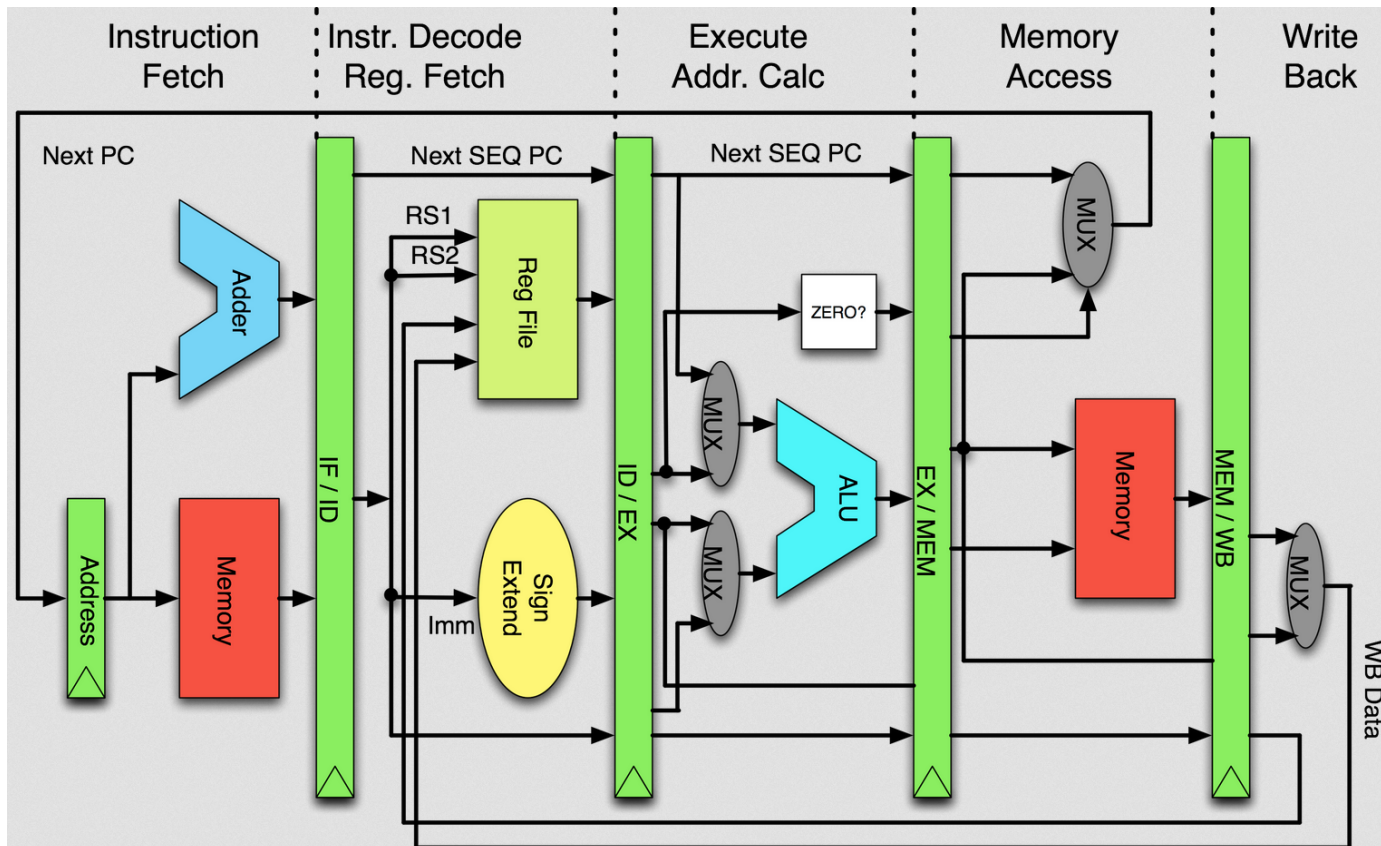
Data transport

- Where does the overhead come from?
 - Register file access
 - How do processors reduce this?
 - Data caches



Hardware reconfiguration

- Mostly multiplexers



General purpose processors

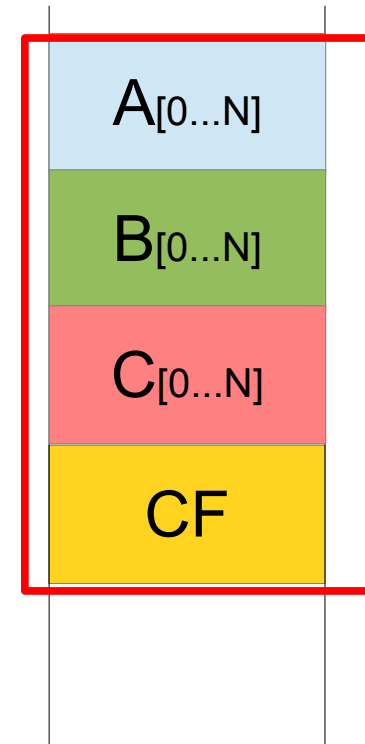
- Lessons learned:
 - Reduce instruction fetching and decoding
 - Reduce cycle-based hardware reconfiguration.
 - Reduce data transport to and from memories and RF.
 - Still needs to be programmable

Hardware acceleration

Static control

- Loops are the best candidate for static control
 - Do the same thing many times

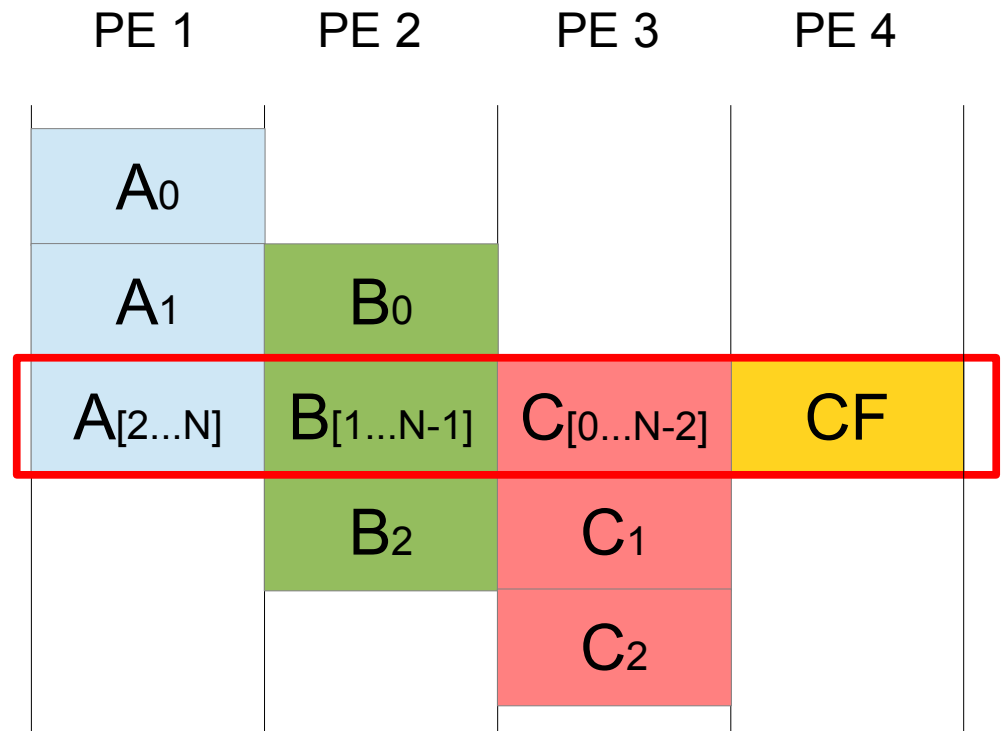
```
for i = 0 to N  
  A  
  B  
  C
```



Static control

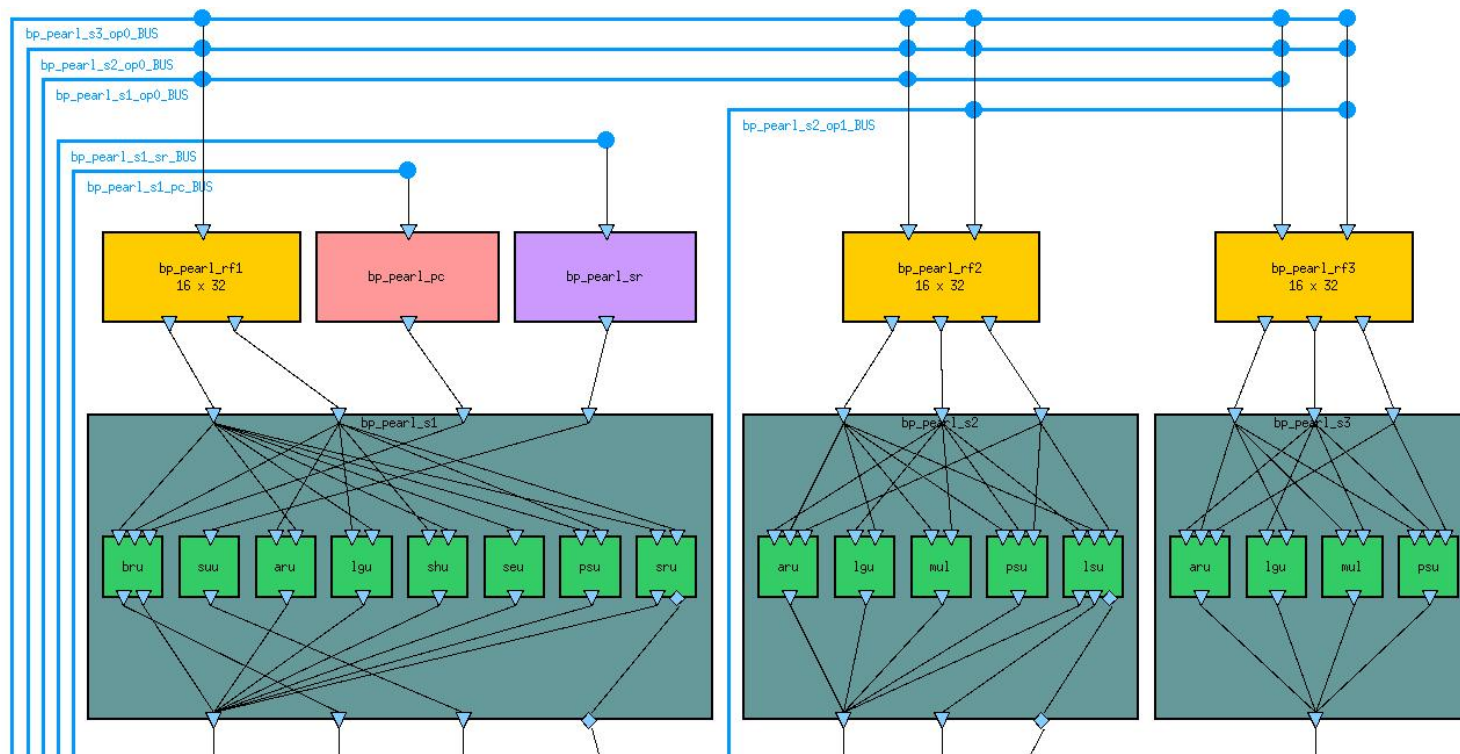
- Loops are the best candidate for static control
 - Do the same thing many times
- Software pipelining

```
for i = 0 to N
  A
  B
  C
```



Very Long Instruction Word processors (VLIW)

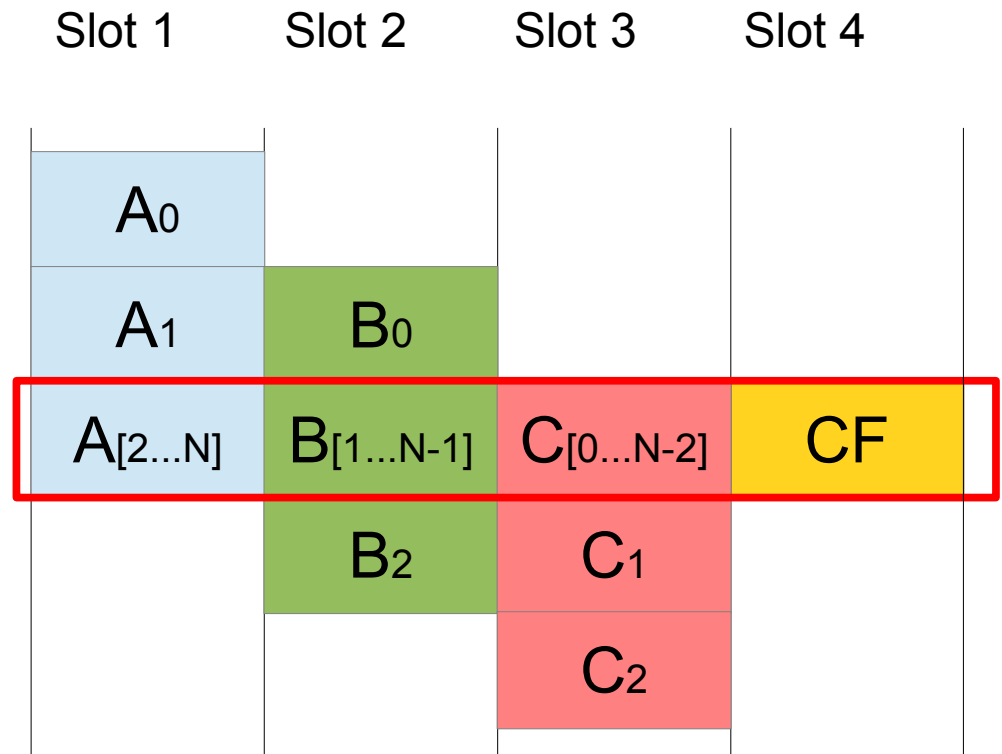
- Processor with multiple issue-slots
- One long instruction controlling them all



Very Long Instruction Word processors (VLIW)

- If we have a 4-issue VLIW we can do our loop in a single cycle.

```
for i = 0 to N  
  A  
  B  
  C
```



Very Long Instruction Word processors (VLIW)

- But once the VLIW is manufactured the number (and functionality) of the issue slots is fixed.
- Problem ...

```
for i = 0 to N  
  A  
  B  
  C  
  D
```

ASICs

- The application is completely software pipelined and implemented ('hard coded') in hardware.
 - No more instruction fetching and decoding
 - But cannot be changed anymore after production
- But ASICs can do something about RF and memory accesses...

Spatial layout

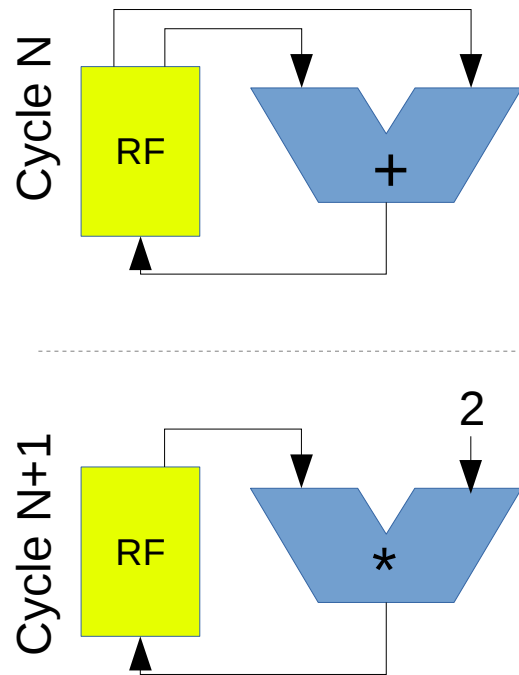
Compute: $(A+B) * 2$



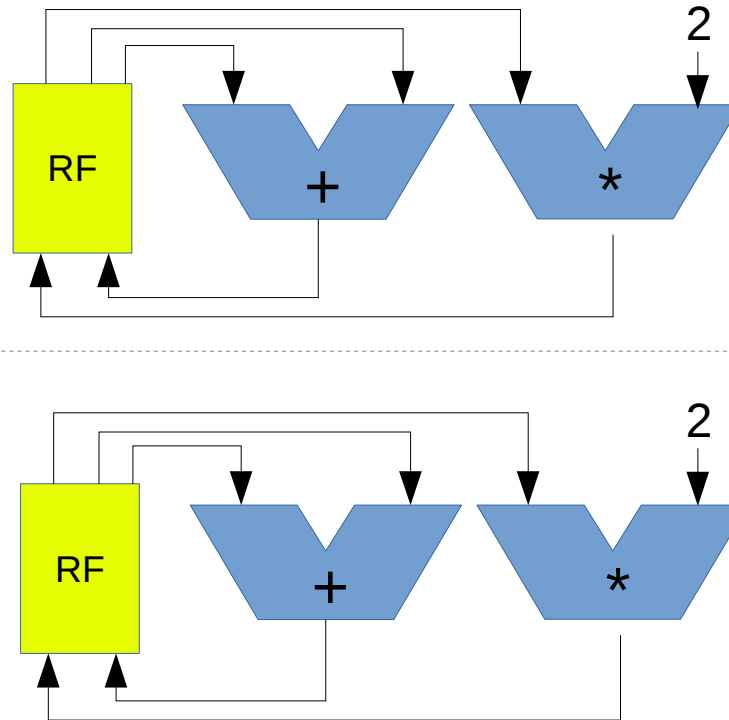
Spatial layout

Compute: $(A+B) * 2$

General purpose



VLIW



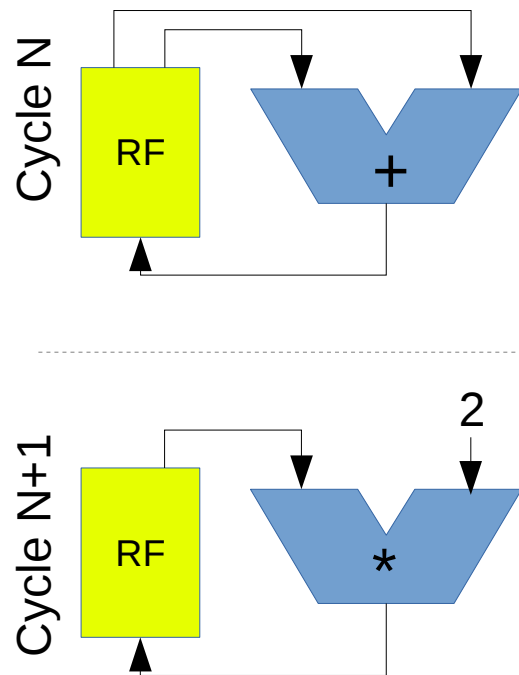
Spatial layout

2 times the throughput!

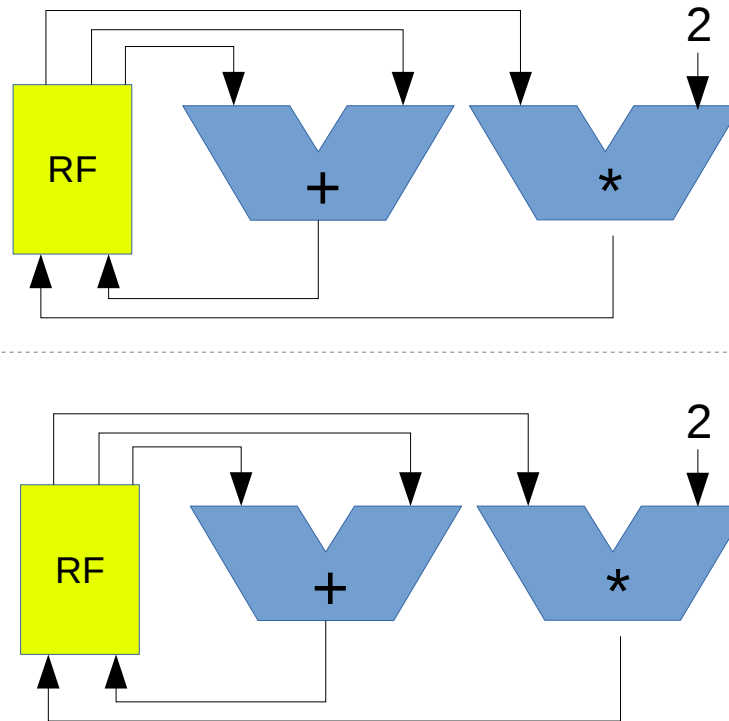
Spatial layout

Compute: $(A+B) * 2$

General purpose

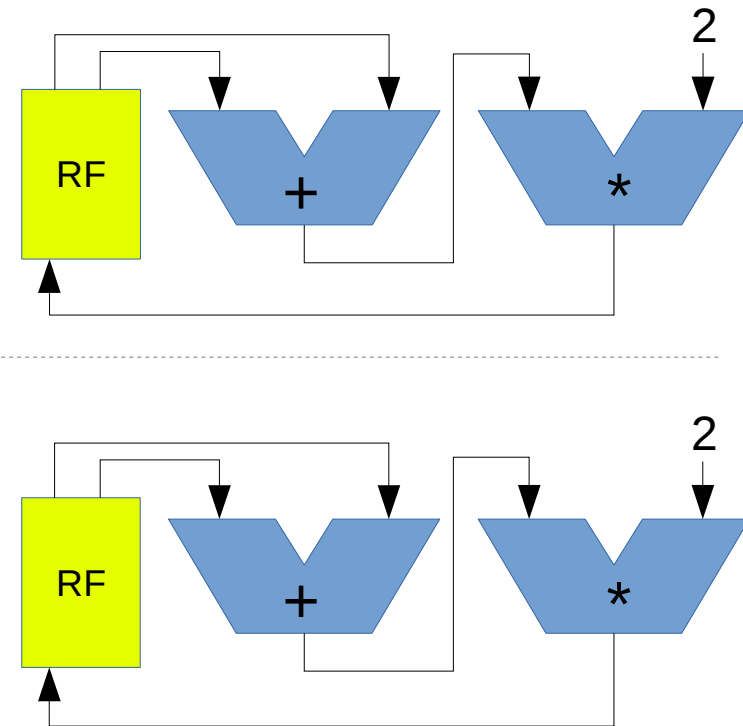


VLIW



2 times the throughput!

Spatial layout



2 times the throughput!

Spatial layout

- Essentially a software pipelined loop with direct connections between functional units.
- Specialized ASICs, e.g. dedicated H.264 decoders.
 - You probably have one in your smart-phone
- Example: H.264 decoding 1080p @ 30fps
 - ASIC: 186 mW (180nm) → 2.78mW (22nm)
 - i5 4300M: 2.84 W (22nm)

Spatial layout

- Essentially a software pipelined loop with direct connections between functional units.
- Specialized ASICs, e.g. dedicated H.264 decoders.
 - You probably have one in your smart-phone
- Example: H.264 decoding 1080p @ 30fps
 - ASIC: 186 mW (180nm) → 2.78mW (22nm)
 - i5 4300M: 2.84 W (22nm)

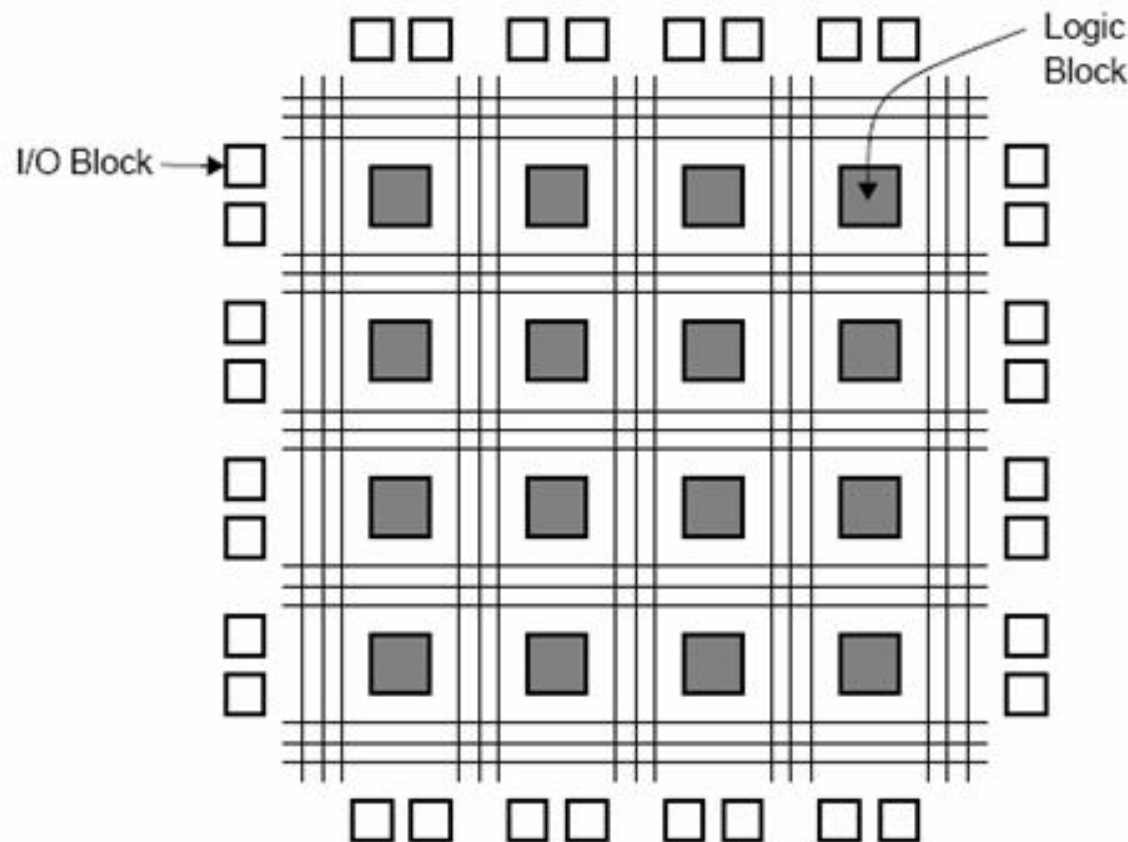
ASICs

- Very efficient
 - (Almost) no control
 - Some configuration registers
 - Fully software-pipelined hardware implementation
 - Reduce memory accesses
 - With spatial layout many register file (and memory) accesses can be avoided
- Very inflexible
 - Highly optimized for a very small application set

Reconfigurable hardware

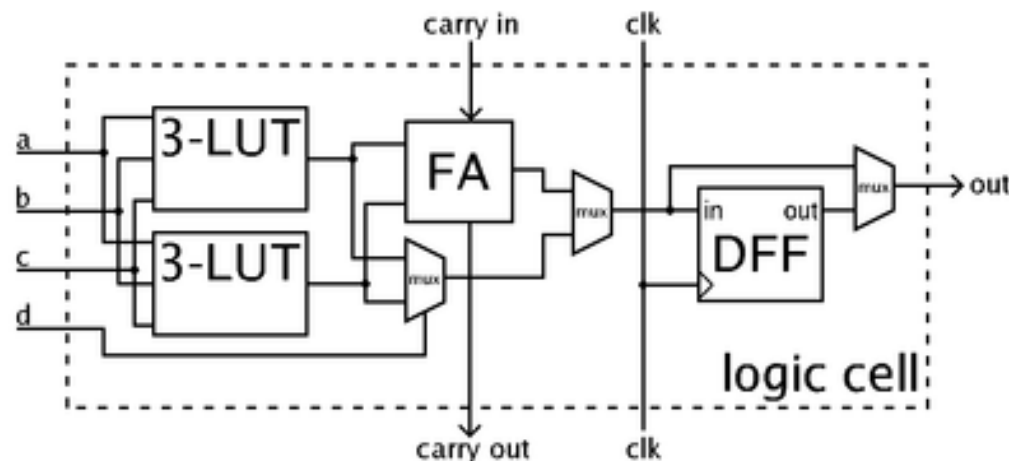
Field Programmable Gate Arrays

- Chip full of configurable logic blocks/cells



Field Programmable Gate Arrays

- What is in a logic block
 - Look-up tables
 - Full adder
 - Flip-Flop
 - Some multiplexers



Field Programmable Gate Arrays

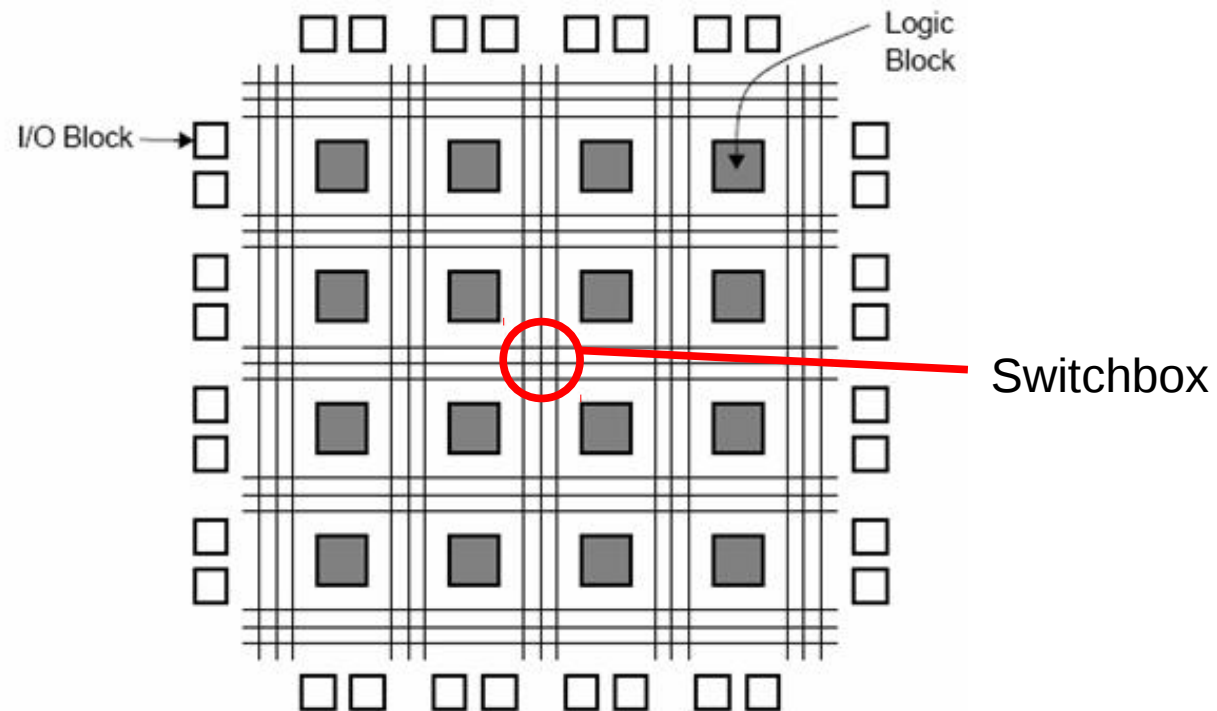
- With the exception of specialized blocks most FPGAs contain gate-level blocks.
- Allows you to build arbitrary hardware
 - Like a box full of logic gates to build circuits.
- These blocks can be connected together via the interconnect.



[chipsetc.com]

Field Programmable Gate Arrays

- The interconnect on a FPGA is static:
 - Configured at application level (typically when you power-up the FPGA).
 - Connections are (usually) fixed after that



Field Programmable Gate Arrays

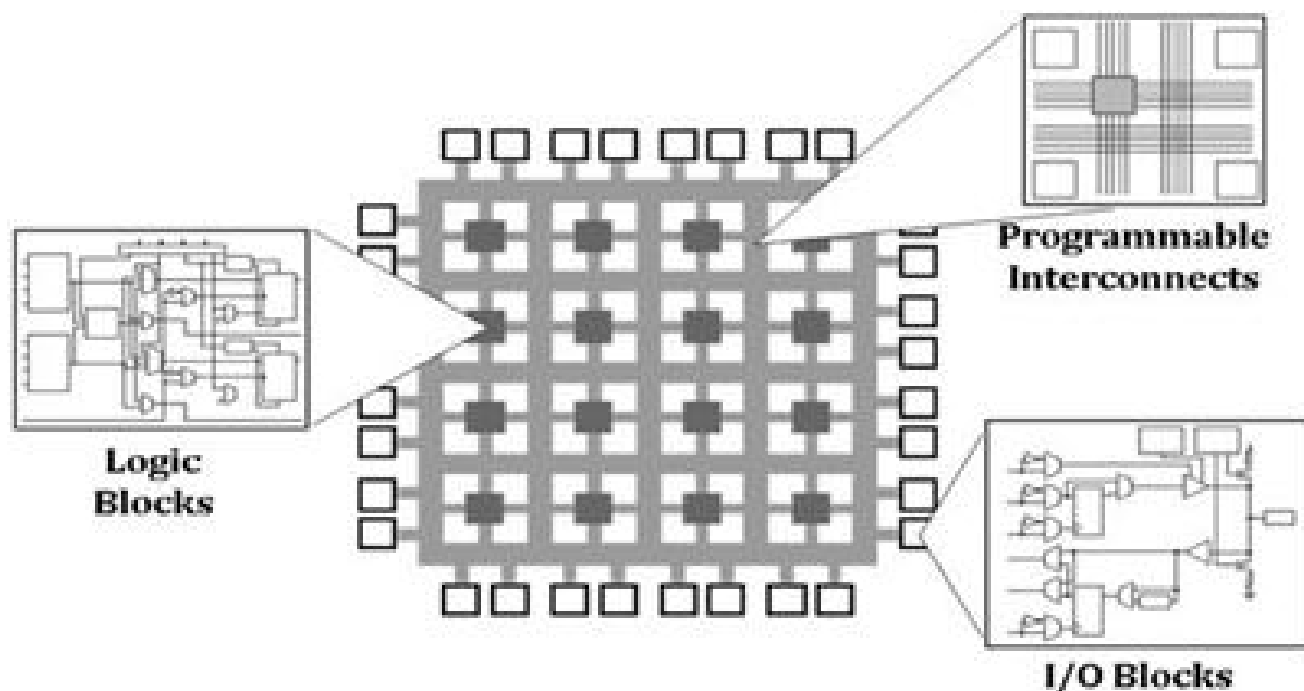
- By configuring the interconnect and the logic blocks arbitrary (digital) circuits are possible.
- This allows for building specialized circuits that implement your algorithm with:
 - Spatial layout
 - Static control
 - Or a processor that runs software if you like to...

Field Programmable Gate Arrays

- Spatial mapping and static control
- Reconfigurable
- ... no such thing as a free lunch ...



Spatial Layout in FPGA

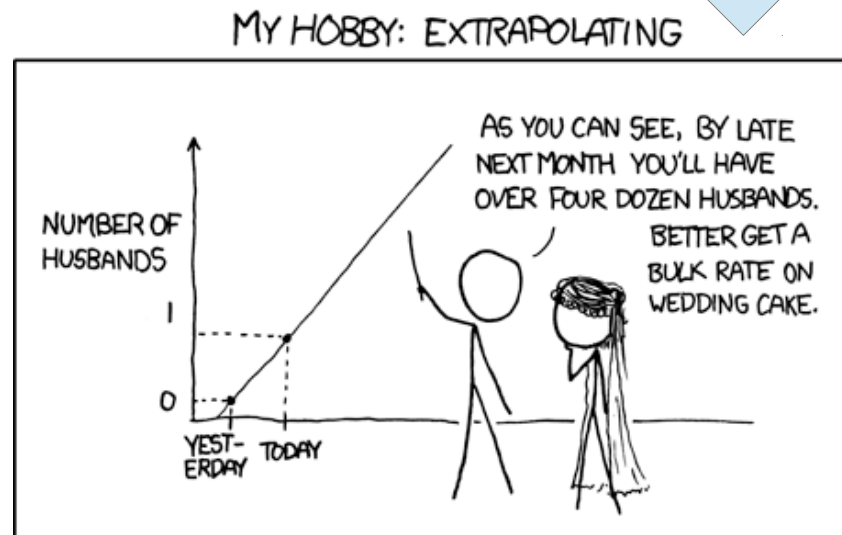


FPGA Configures at **gate level**, which incurs **large overheads**:

- Large configuration memory (SRAM leakage: high static power)
- Complex routing network (many long wires: high dynamic power)

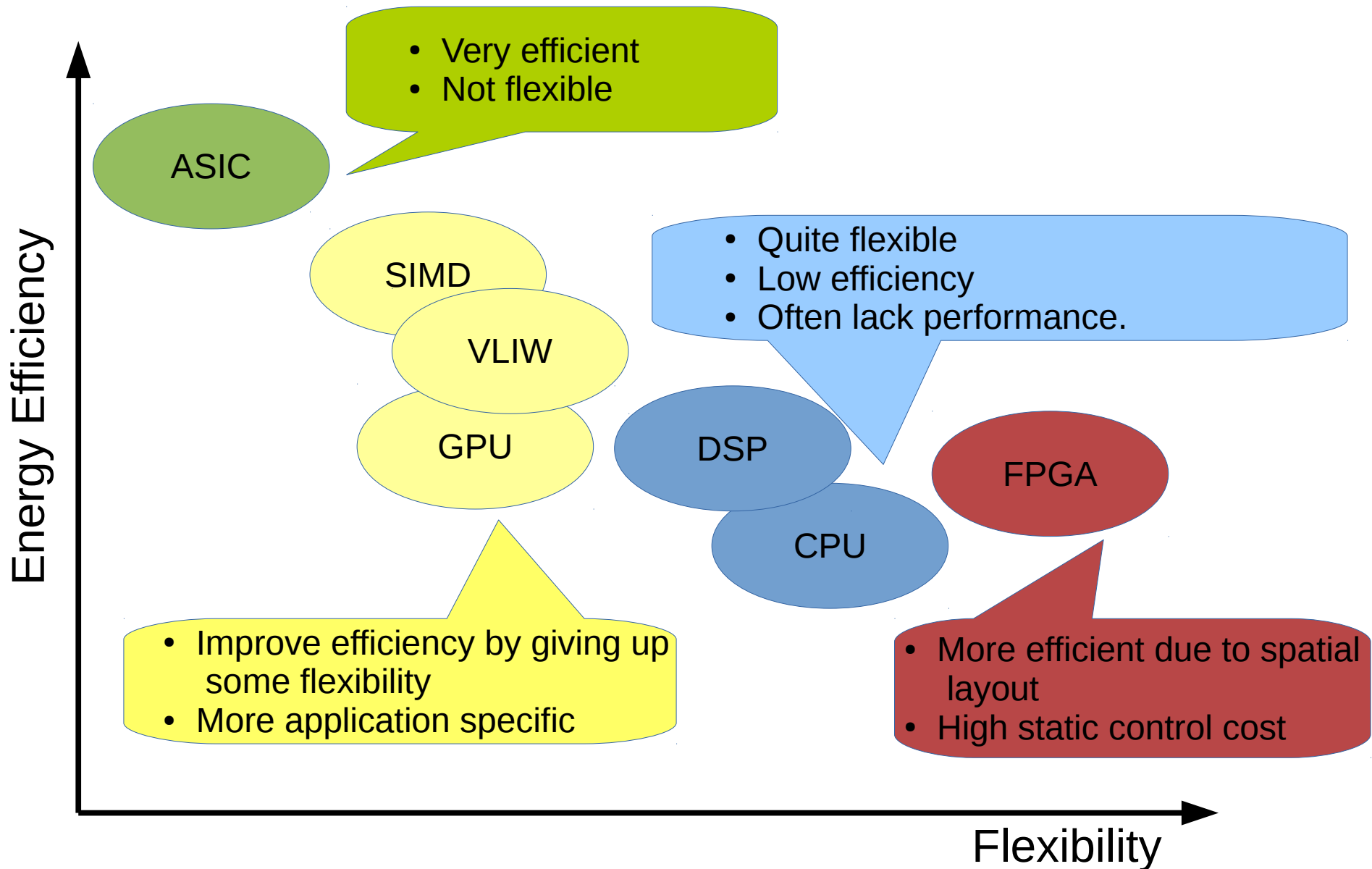
Field Programmable Gate Arrays

- Each configurable item has some configuration memory cells attached that configure it.
- Often several megabits
- Memory cells have leakage
- ... Many cells have more leakage ...
- Not trivial to program



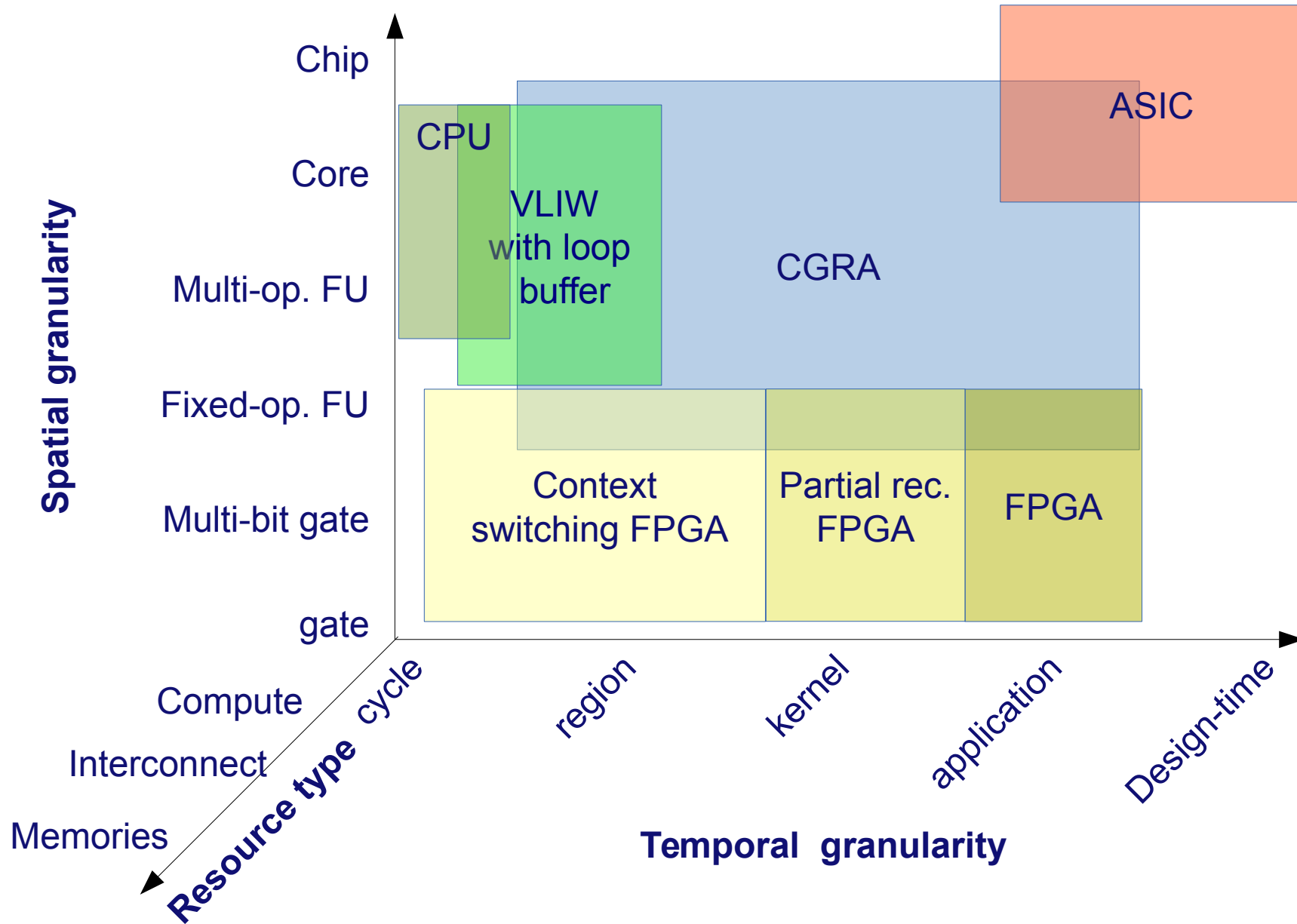
[xkcd.com]

What have we learned so far?



Coarse Grained Reconfigurable Architecture

Coarse Grained Reconfigurable Architecture



Spatial Layout on the Cheap

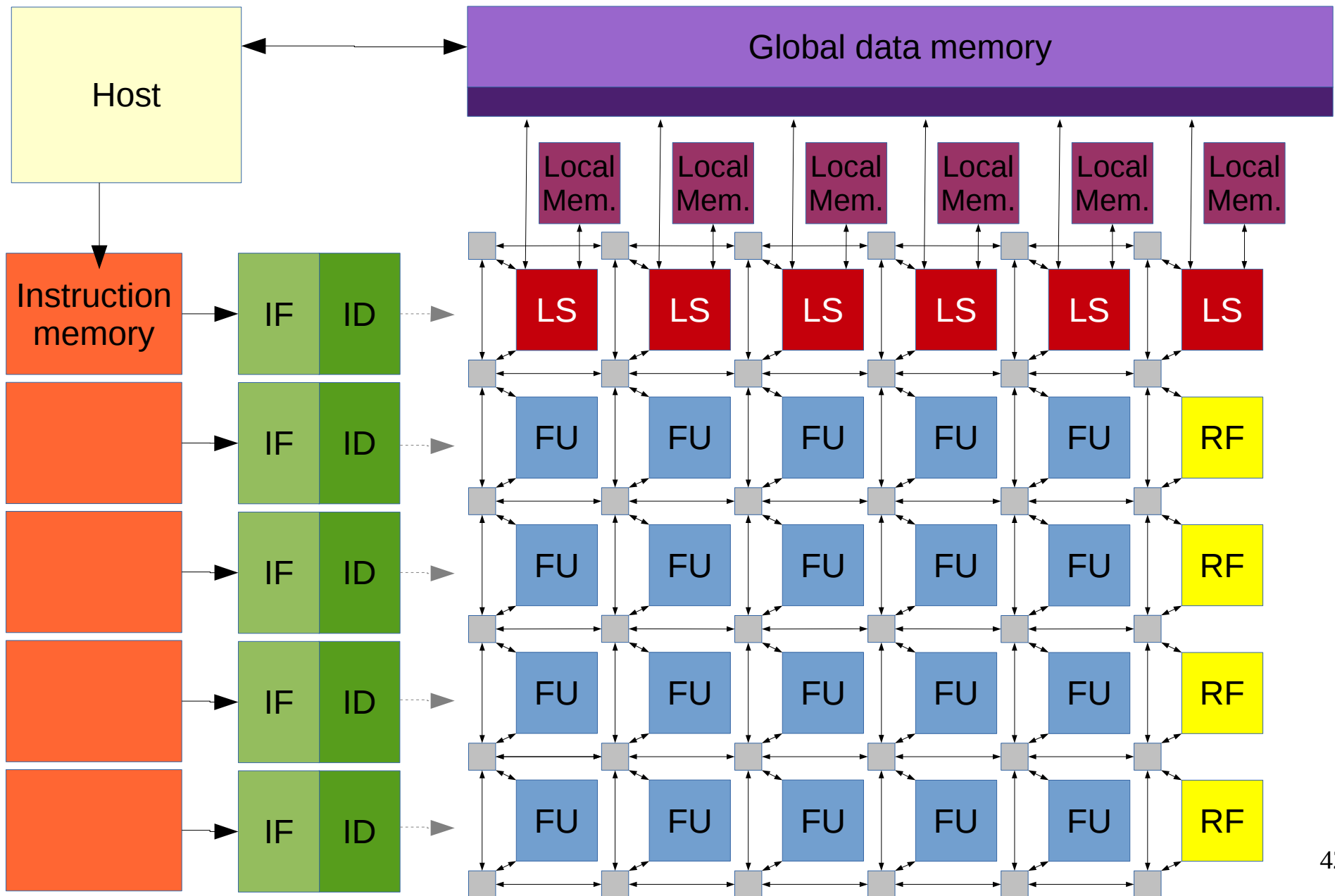
Gate-level granularity is often not required
digital signal processing

What our architecture does:

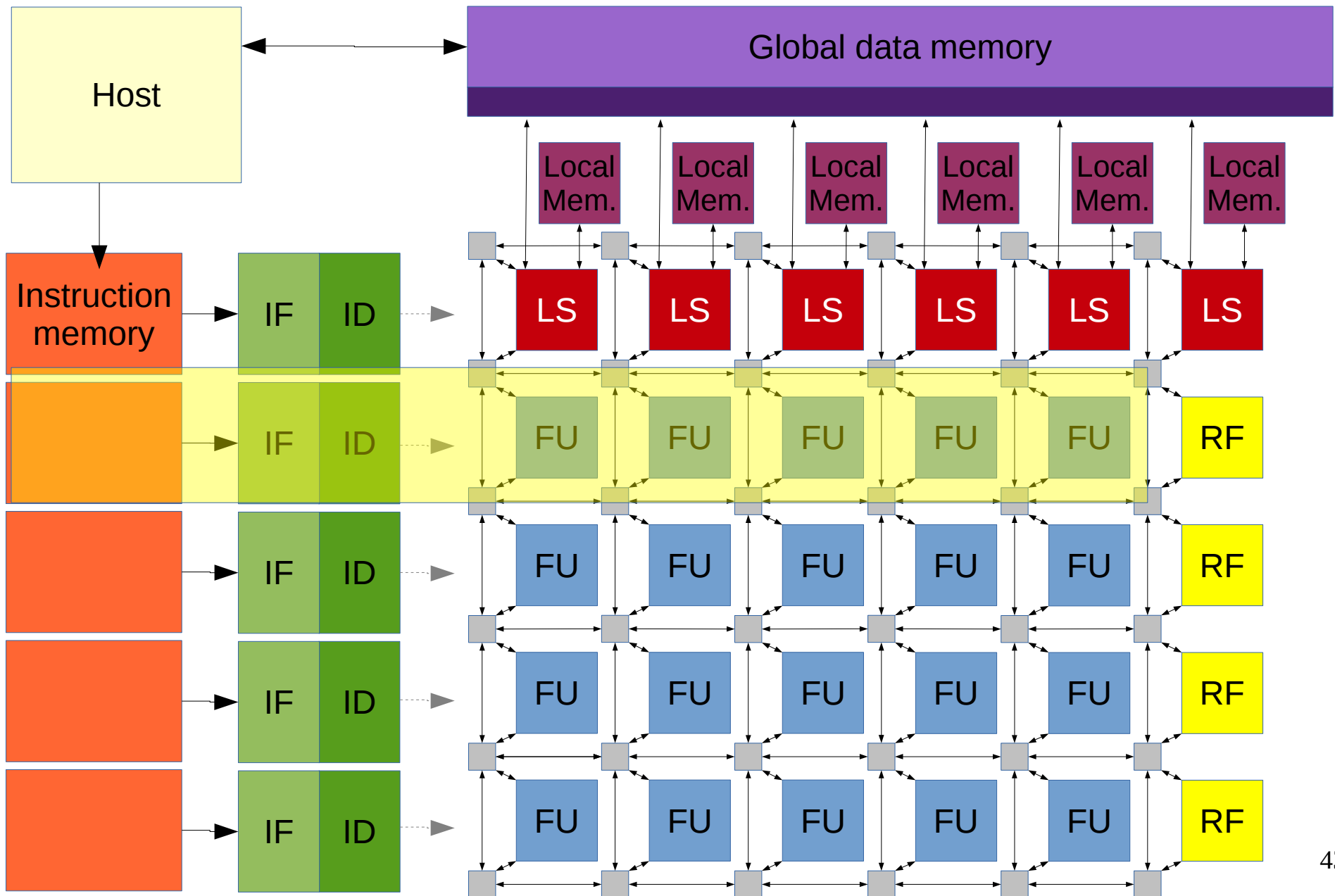
- Configure at **functional unit level**
- **Statically route data-path** (spatial layout) but allow instructions.

A specialized FPGA to build processors

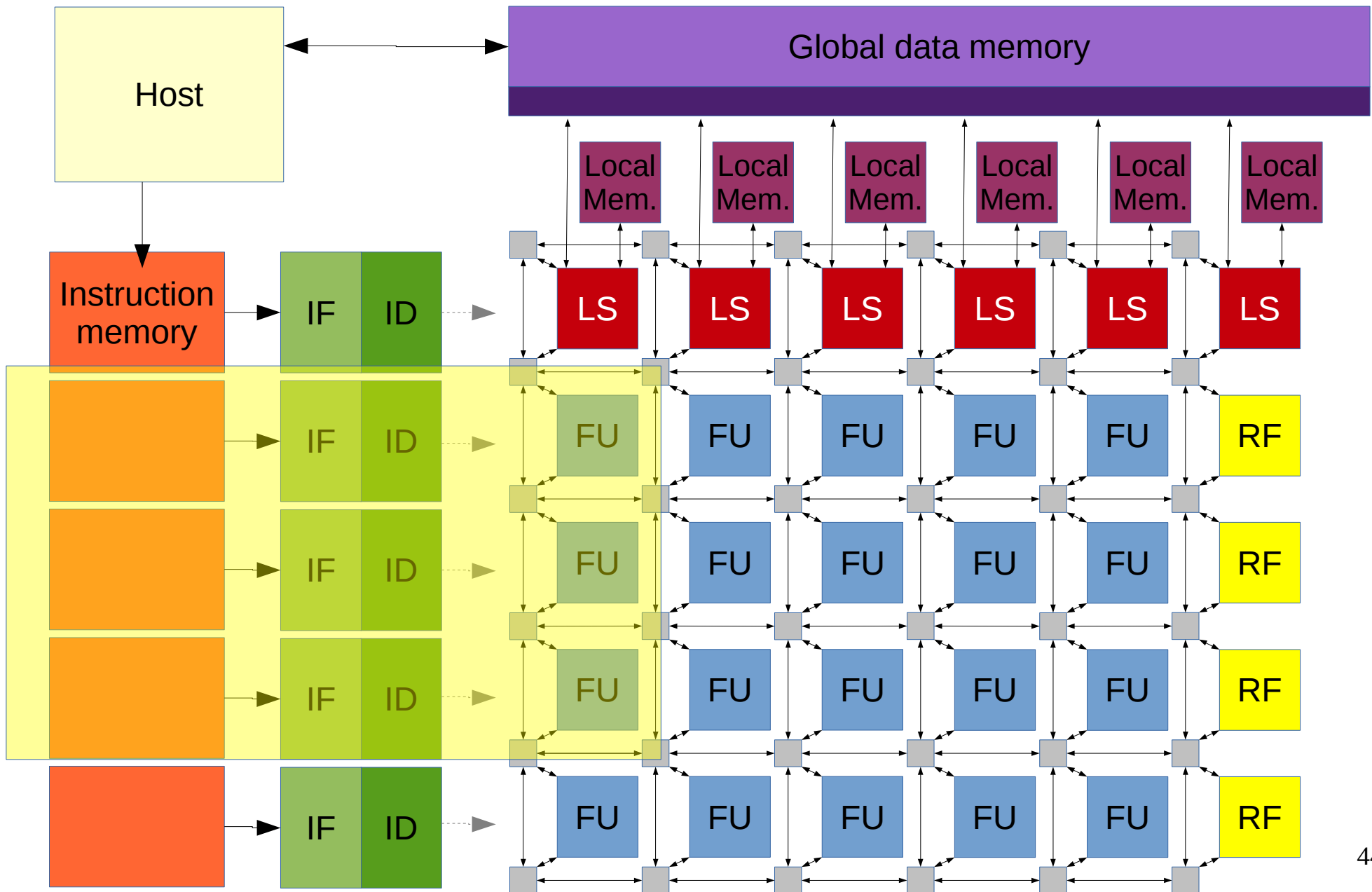
CGRA



SIMD construction



VLIW construction



Building Blocks

IF ID Instruction Fetch/Decode (IF/ID)

LS Load Store Units (LSU)

RF Register Files (RF)

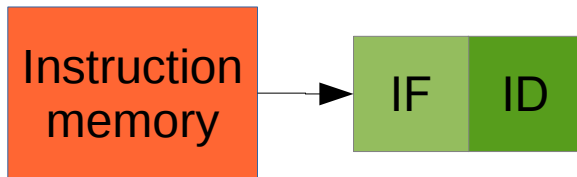
MEM Memories (MEM)

FU Functional Units (FU)

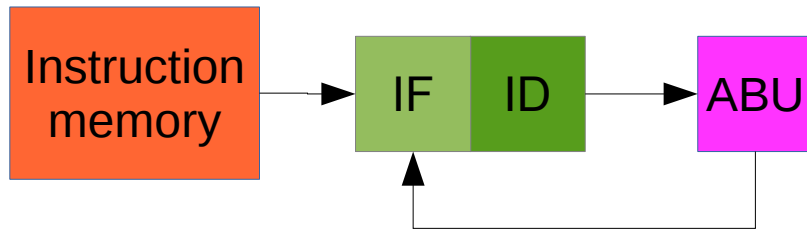
- Add, subtract, bitlevel operations
- Multiplication
- Accumulator
- ...

■ Switchboxes

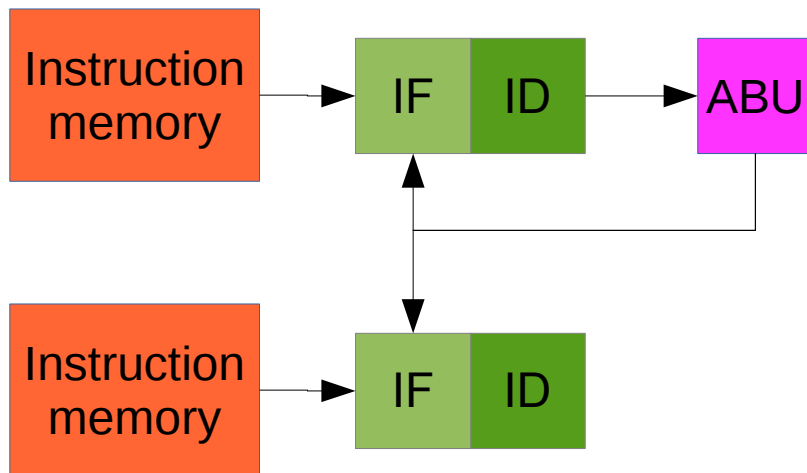
Building a processor



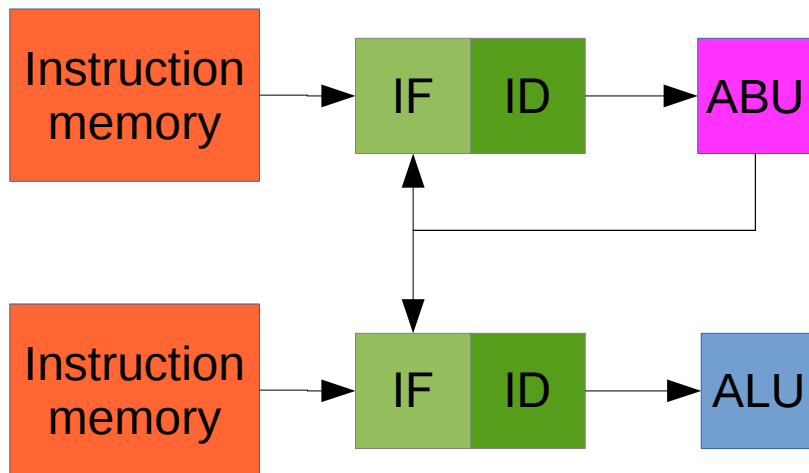
Building a processor



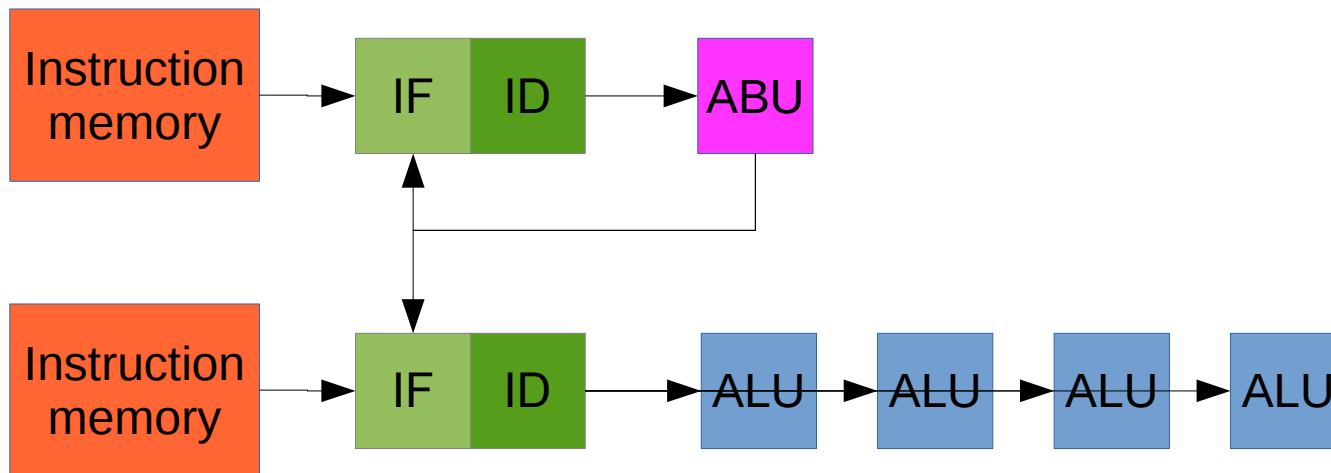
Building a processor



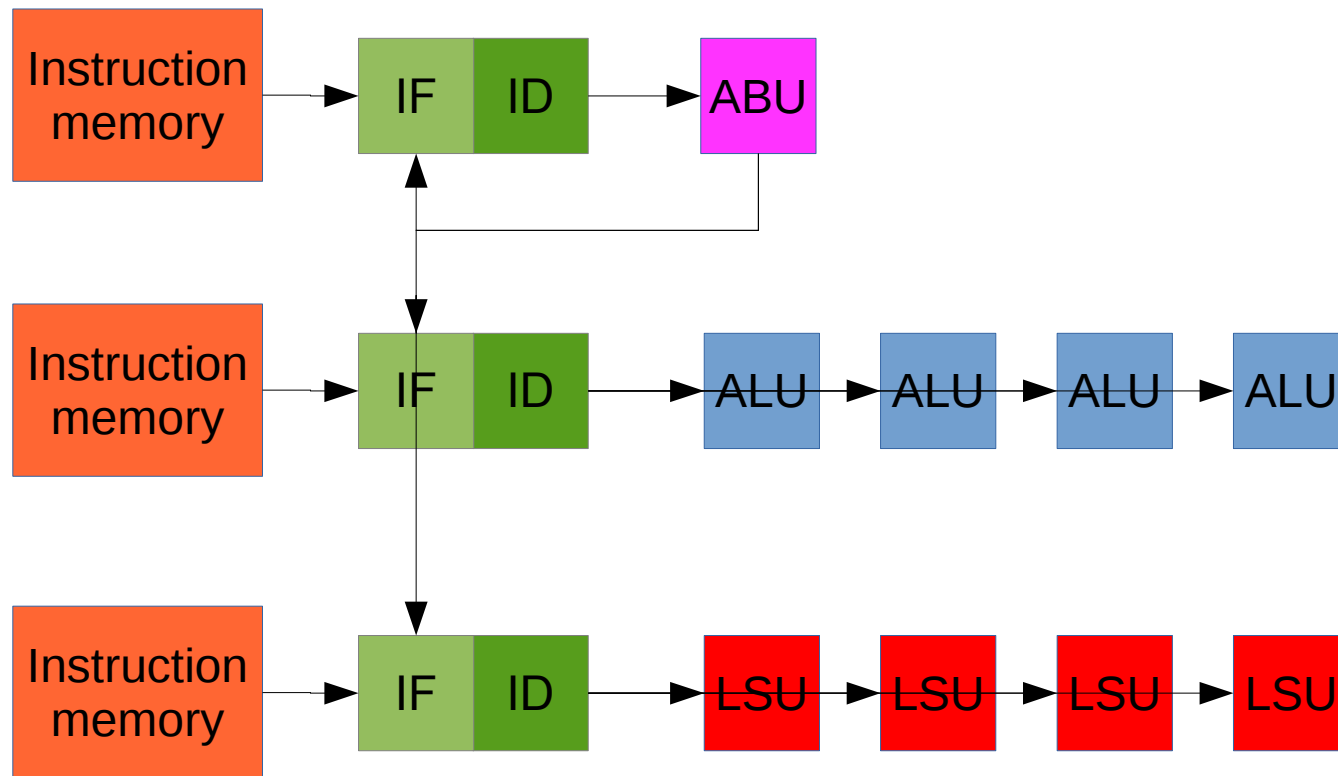
Building a processor



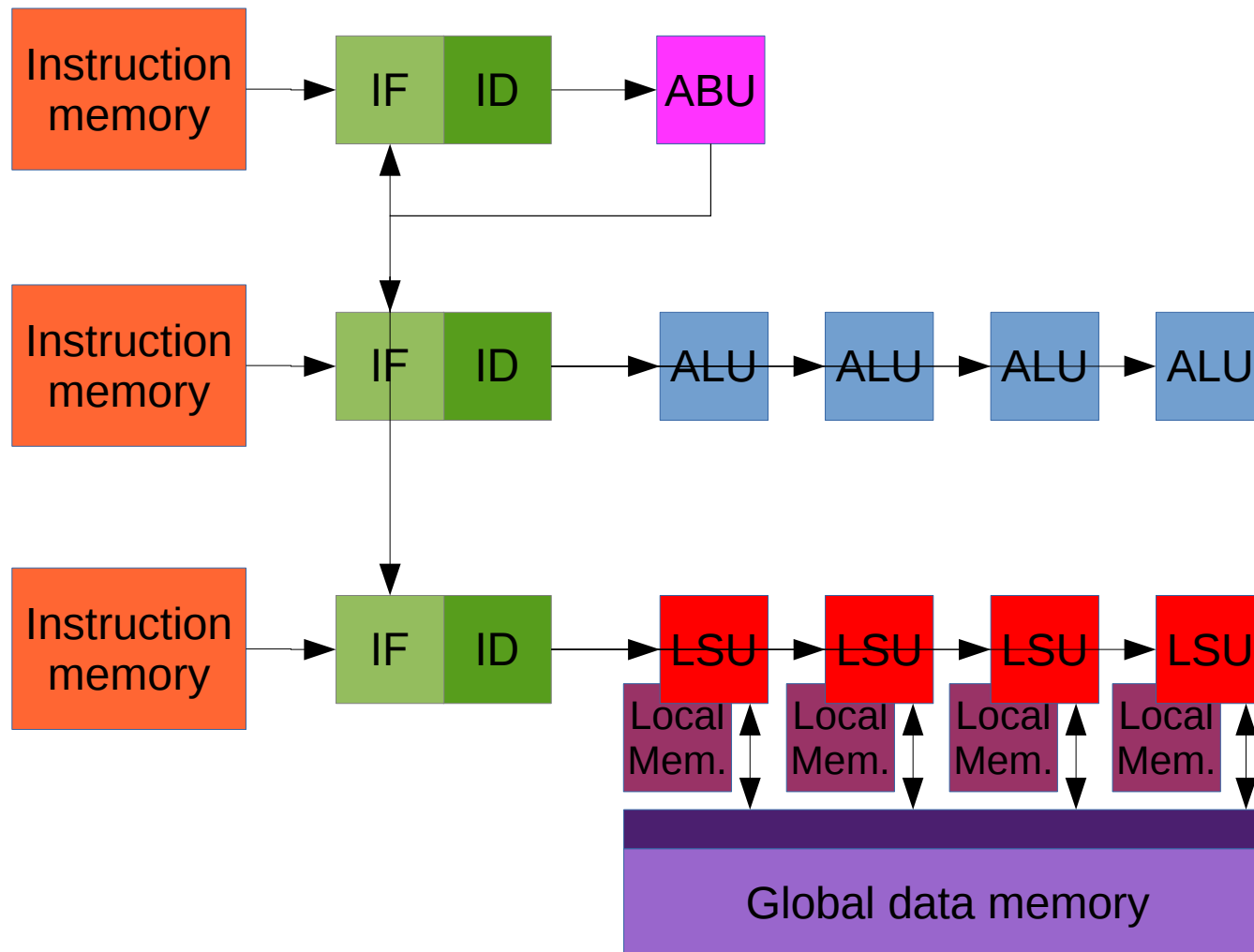
Building a processor



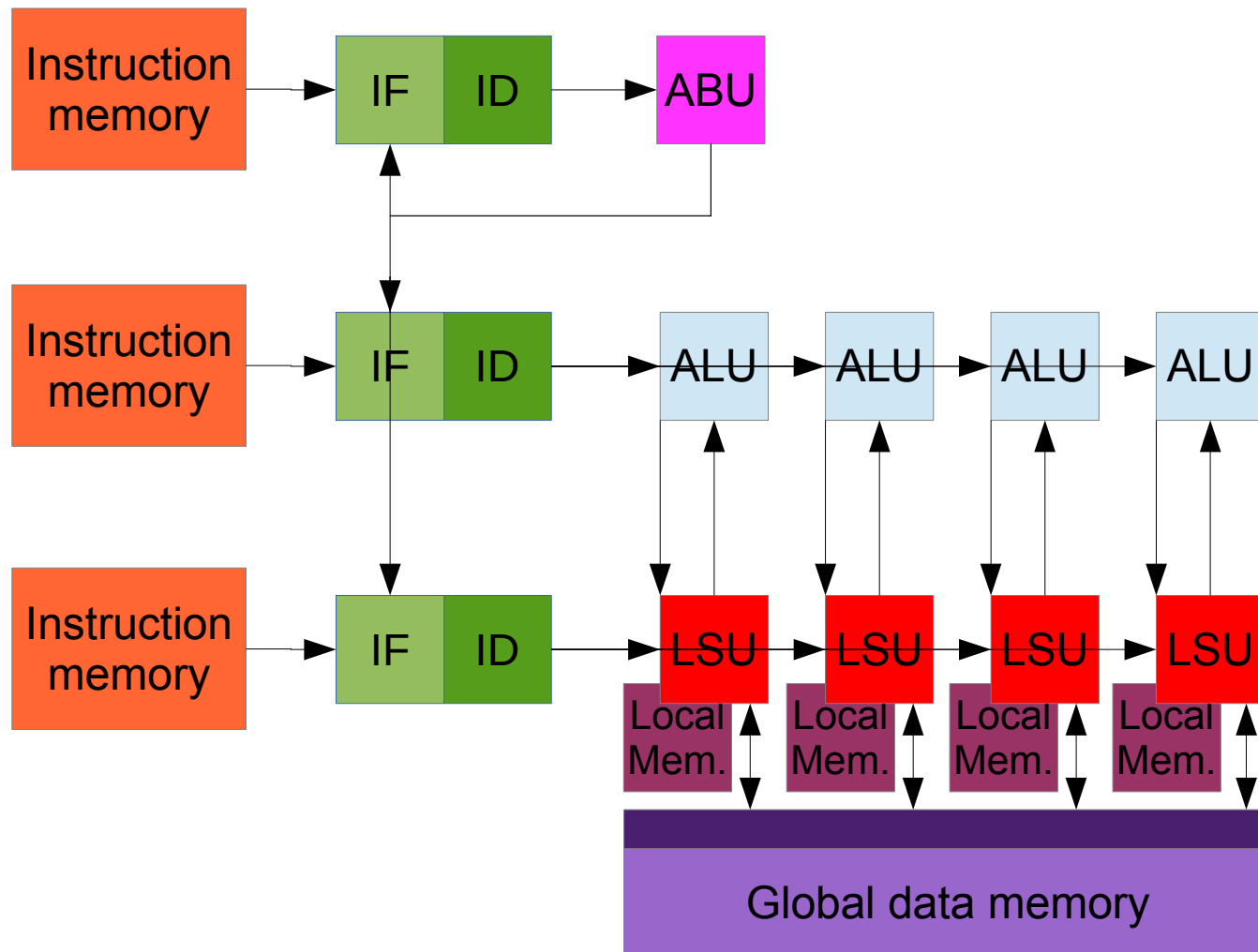
Building a processor



Building a processor



Building a processor



Building a processor

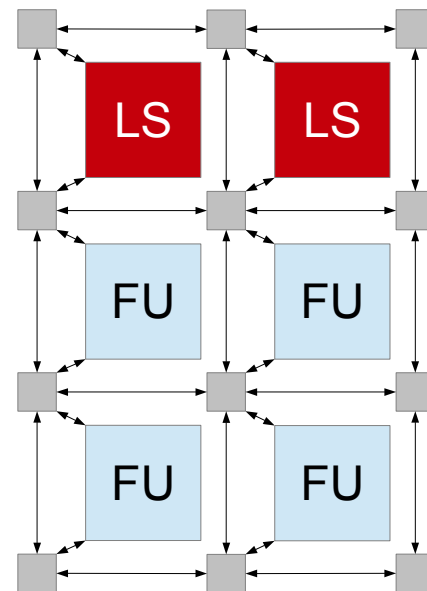
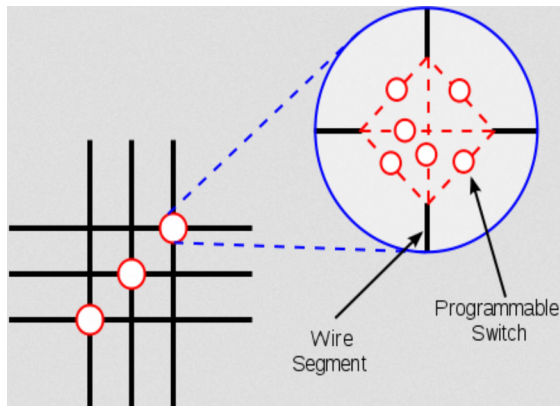
- We can have multiple ABUs
 - What does this mean?

Building a processor

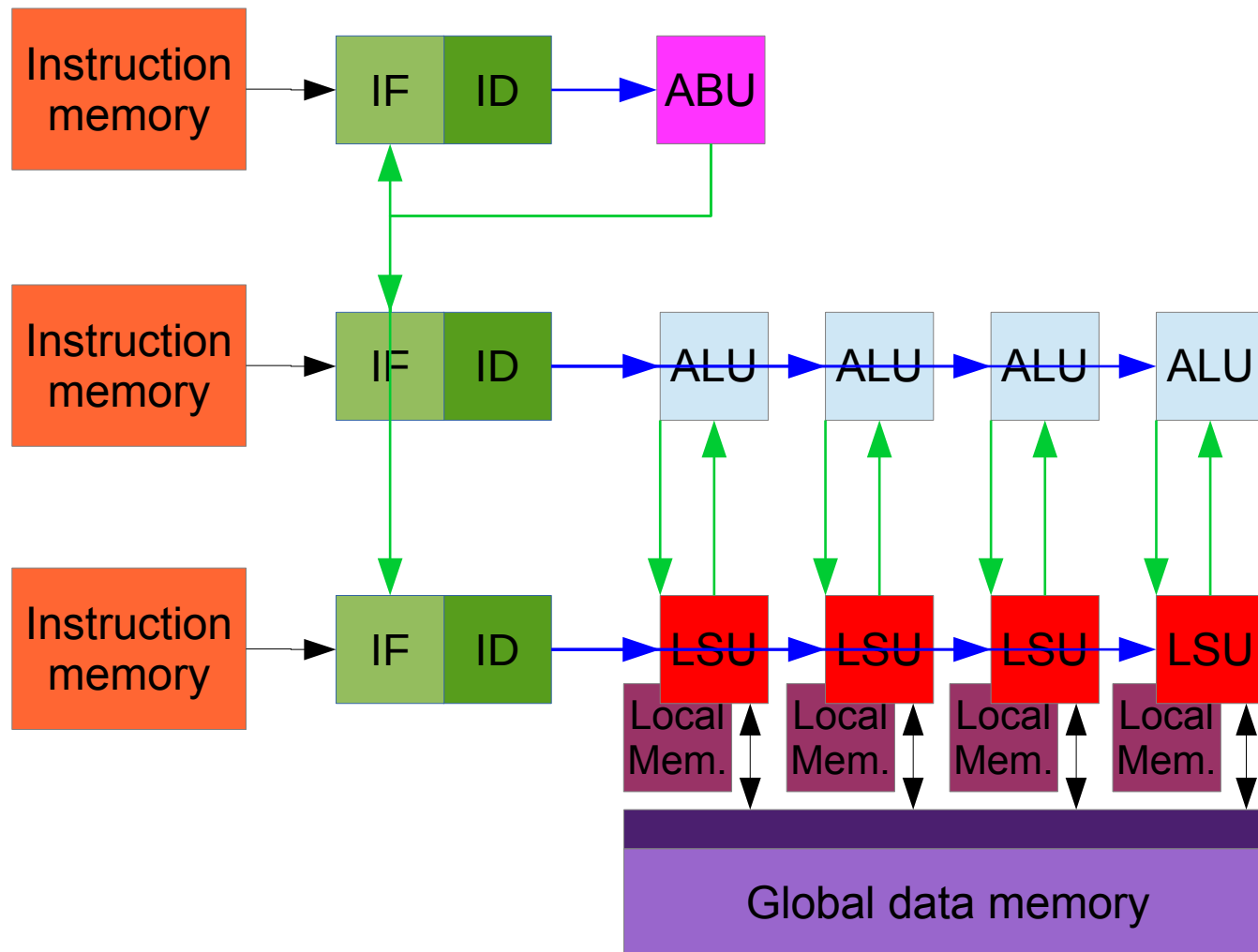
- Architecture is specified with XML file
- Two versions:
 - Static: all connections are fixed at design time
 - Dynamic: connections can be configured at runtime
- The assignment will use the static version

Dynamic CGRA

- Connections can be changed at run-time
 - Very similar to FPGA



Dynamic CGRA

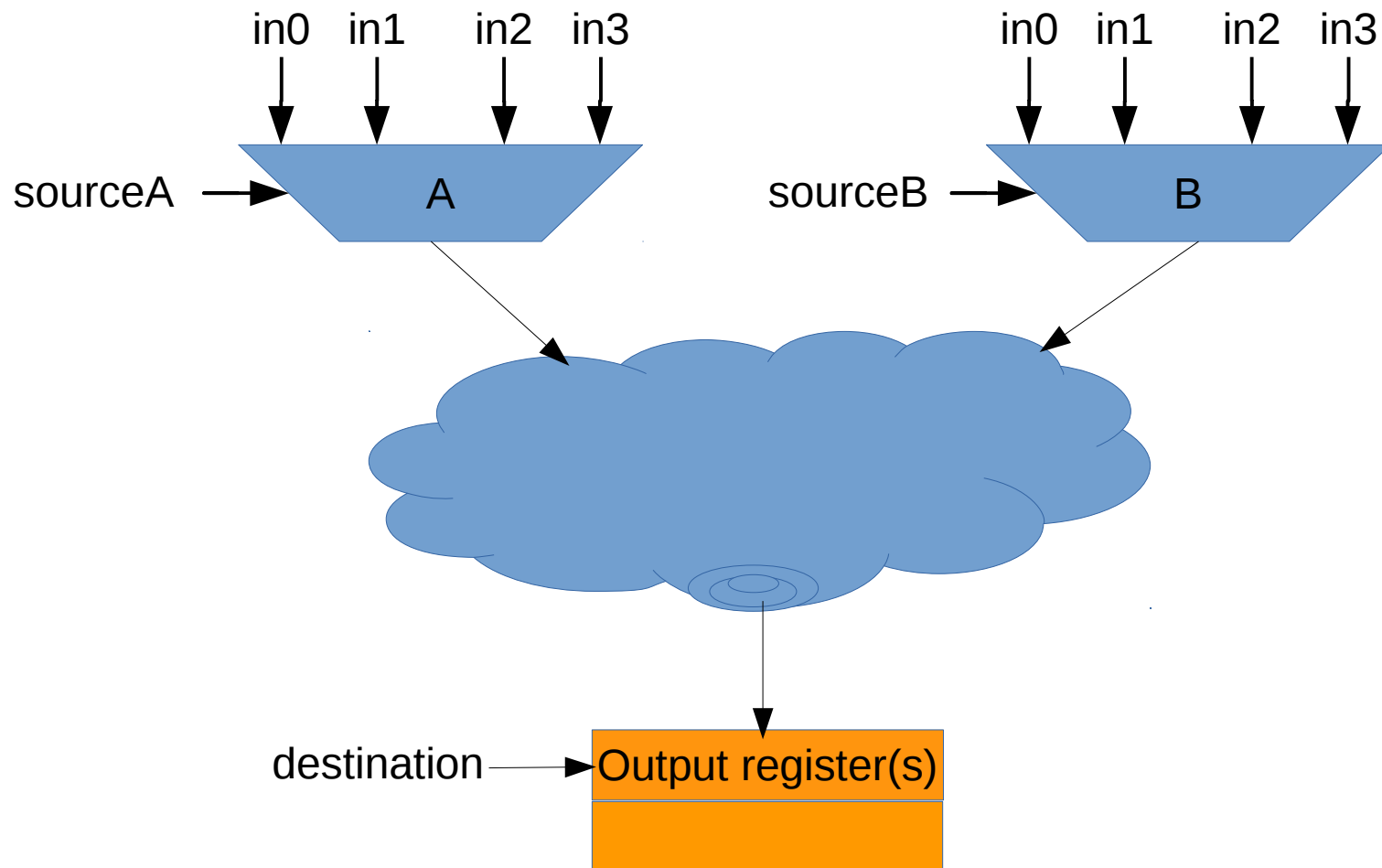


Functional units

- As mentioned before, we have several:
 - ALU Arithmetic Logic Unit
 - RF Register File
 - LSU Load-Store Unit
 - ABU Accumulate Branch Unit
 - MUL Multiplier
 - IU Immediate Unit

Functional units

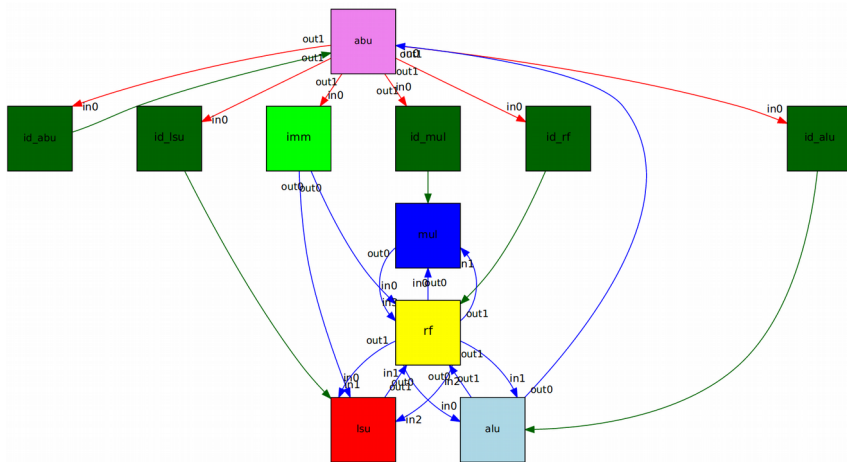
- Most units have 4 inputs and 2 outputs



Functional units

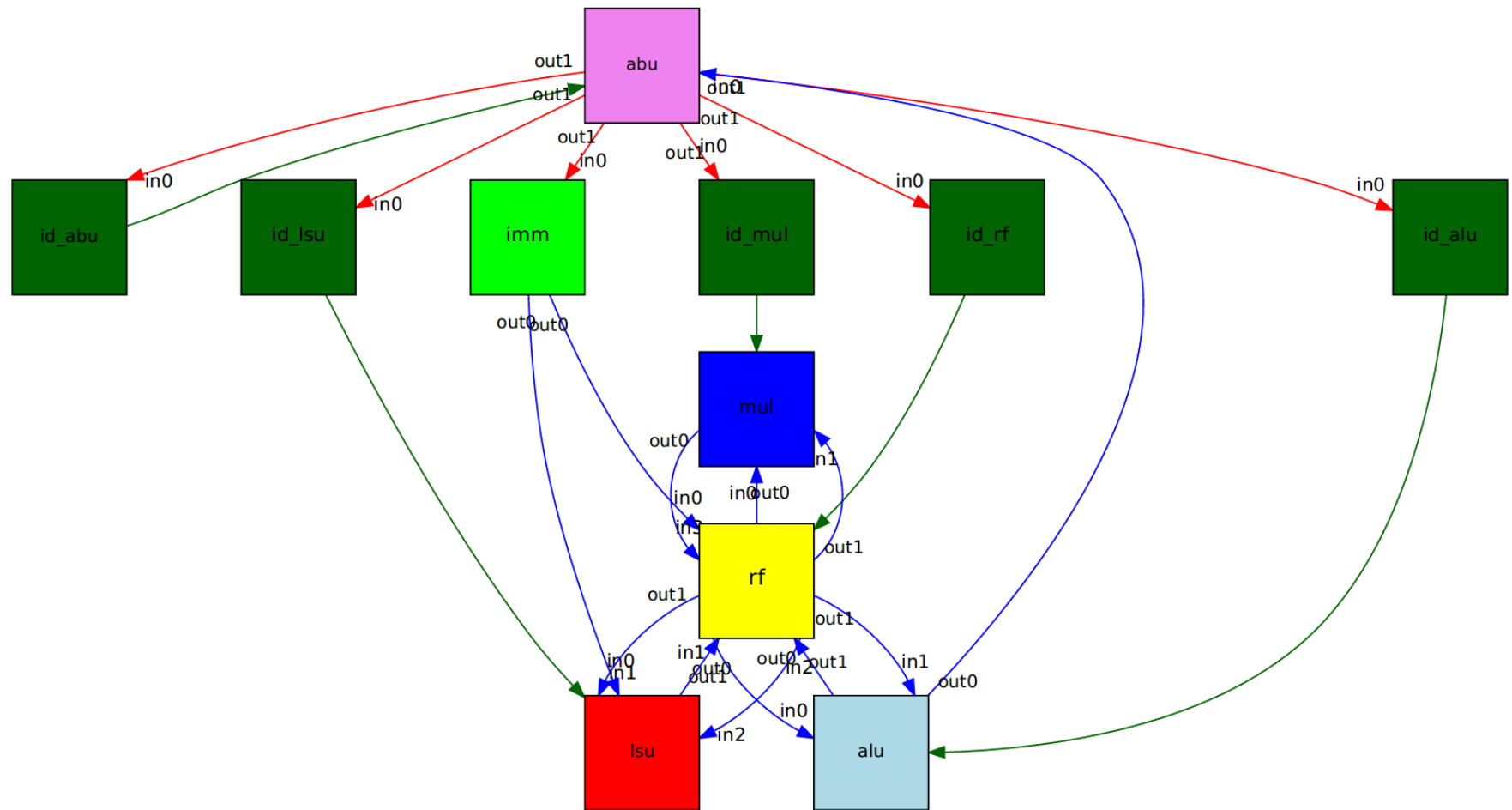
- Source inputs and destination outputs specified in instruction.
- Instruction usually in the form:
 opcode dest, inA, inB
- Each output register is a source on the network.

- Two things are required:
 - Architecture description
 - Program



id	id_sbu	id_ff	id_lsu	ins	text	id_sbu
1	text	text	text	text	text	
2	nop	nop	nopi	ins1	nop	
3	str r1, #0	store wsize[0] in r1	ins1	1 : weight[0][0]	nop	
4	str r2, #0	store wsize[1] in r2	ins1	2 : weight[0][1]	nop	
5	str r3, #0	store wsize[2] in r3	ins1	3 : weight[0][2]	nop	
6	str r4, #0	store wsize[3] in r4	ins1	4 : weight[0][3]	nop	
7	str r5, #0	store wsize[4] in r5	ins1	5 : weight[0][4]	nop	
8	str r6, #0	store wsize[5] in r6	ins1	6 : weight[0][5]	nop	
9	str r7, #0	store wsize[6] in r7	ins1	7 : weight[0][6]	nop	
10	str r8, #0	store wsize[7] in r8	ins1	8 : weight[0][7]	nop	
11	str r9, #0	store wsize[8] in r9	ins1	9 : initial value of 1	nop	
12	branch target k-loop (12)	str r10, #0	ins1	1 : initial value of k	nop	
13	branch target k-loop (14)	str r11, #0	ins1	2 : initial value of k	nop	
14	branch target k-loop (17)	str r12, #0	ins1	3 : initial value of k	nop	
15	branch target k-loop (19)	str r13, #0	ins1	4 : initial value of k	nop	
16		str r14, #0	ins1	5 : initial value of l	nop	
17		str r15, #0	ins1	6 : initial value of l	nop	
18		str r16, #0	ins1	7 : initial value of l	nop	
19		str r17, #0	ins1	8 : initial value of l	nop	
20		str r18, #0	ins1	9 : initial value of l	nop	
21		str r19, #0	ins1	10 : initial value of l	nop	
22		str r20, #0	ins1	11 : initial value of l	nop	
23		str r21, #0	ins1	12 : initial value of l	nop	
24		str r22, #0	ins1	13 : initial value of l	nop	
25		str r23, #0	ins1	14 : initial value of l	nop	
26		str r24, #0	ins1	15 : initial value of l	nop	
27		str r25, #0	ins1	16 : initial value of l	nop	
28		str r26, #0	ins1	17 : initial value of l	nop	
29		str r27, #0	ins1	18 : initial value of l	nop	
30		str r28, #0	ins1	19 : initial value of l	nop	
31		str r29, #0	ins1	20 : initial value of l	nop	
32		str r30, #0	ins1	21 : initial value of l	nop	
33		str r31, #0	ins1	22 : initial value of l	nop	
34		str r32, #0	ins1	23 : initial value of l	nop	
35		str r33, #0	ins1	24 : initial value of l	nop	
36		str r34, #0	ins1	25 : initial value of l	nop	
37		str r35, #0	ins1	26 : initial value of l	nop	
38		str r36, #0	ins1	27 : initial value of l	nop	
39		str r37, #0	ins1	28 : initial value of l	nop	
40		str r38, #0	ins1	29 : initial value of l	nop	
41		str r39, #0	ins1	30 : initial value of l	nop	
42		str r40, #0	ins1	31 : initial value of l	nop	
43		str r41, #0	ins1	32 : initial value of l	nop	
44		str r42, #0	ins1	33 : initial value of l	nop	
45		str r43, #0	ins1	34 : initial value of l	nop	
46		str r44, #0	ins1	35 : initial value of l	nop	
47		str r45, #0	ins1	36 : initial value of l	nop	
48		str r46, #0	ins1	37 : initial value of l	nop	
49		str r47, #0	ins1	38 : initial value of l	nop	
50		str r48, #0	ins1	39 : initial value of l	nop	
51		str r49, #0	ins1	40 : initial value of l	nop	
52		str r50, #0	ins1	41 : initial value of l	nop	
53		str r51, #0	ins1	42 : initial value of l	nop	
54		str r52, #0	ins1	43 : initial value of l	nop	
55		str r53, #0	ins1	44 : initial value of l	nop	
56		str r54, #0	ins1	45 : initial value of l	nop	
57		str r55, #0	ins1	46 : initial value of l	nop	
58		str r56, #0	ins1	47 : initial value of l	nop	
59		str r57, #0	ins1	48 : initial value of l	nop	
60		str r58, #0	ins1	49 : initial value of l	nop	
61		str r59, #0	ins1	50 : initial value of l	nop	
62		str r60, #0	ins1	51 : initial value of l	nop	
63		str r61, #0	ins1	52 : initial value of l	nop	
64		str r62, #0	ins1	53 : initial value of l	nop	
65		str r63, #0	ins1	54 : initial value of l	nop	
66		str r64, #0	ins1	55 : initial value of l	nop	
67		str r65, #0	ins1	56 : initial value of l	nop	
68		str r66, #0	ins1	57 : initial value of l	nop	
69		str r67, #0	ins1	58 : initial value of l	nop	
70		str r6				

Architecture description



Programming

- We would liked to give you a compiler...
 - Still in development.
 - Works but not yet using the architecture very efficiently.
- Programming is done in an assembler dialect
 - We call it PASM: Parallel Assembler

Some results

- Post place & route results for 2 benchmark applications
 - 40nm commercial library
 - Targeted at 100 MHz

Benchmark	Architecture	Cycles	Power [mW]	Energy [nJ]
Binarization	Cortex-M0	115007	1.57	1806
	CGRA scalar	8209	1.07	88
	CGRA vector 4	2069	2.34	48
FIR	Cortex-M0	665618	0.98	6523
	CGRA	2224	9.48	211

37x

31x

Reconfigurable architectures

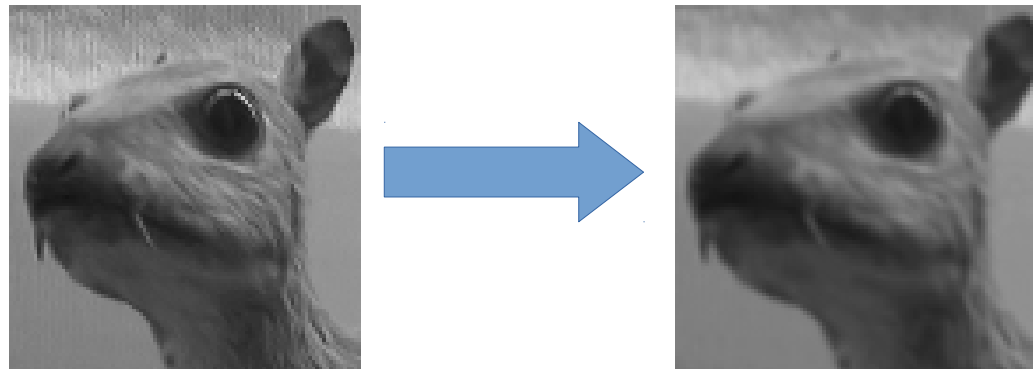
- Lower static control power than FPGA
 - Higher granularity means less control bits
 - Reconfiguration is faster
- Can adapt better to the application than VLIW
 - Only use the number of issue slots really required
 - Support spatial mapping and single-cycle loops
 - Unused units can be switched off.
- High number of operations per cycle
 - But as much as possible: the same instruction

Current developments

- Single cycle loop support
- Debug support
- Approximate compute platform
- Compiler (LLVM, Roel Jordans, talk tomorrow)
- Tape-out plans:
 - Small design in October
 - More complex/optimized design in May 2018.

The assignment

- You will get a naive implementation for a Gaussian blur convolution kernel.



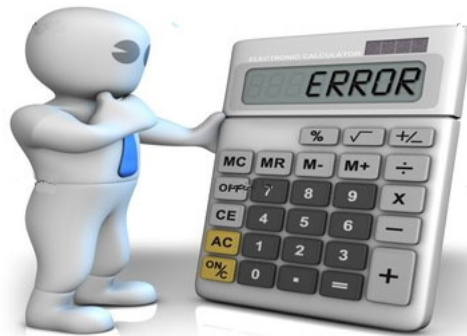
- Your job is to make a trade-off between energy, area and performance

Your assignment

- You can:
 - Modify the architecture:
 - Implement data-level parallelism
 - Implement instruction-level parallelism
 - Use bypassing
 - Use other nice hardware features
 - Modify the application:
 - There are algorithm level optimizations possible
 - To make use of the architecture changes

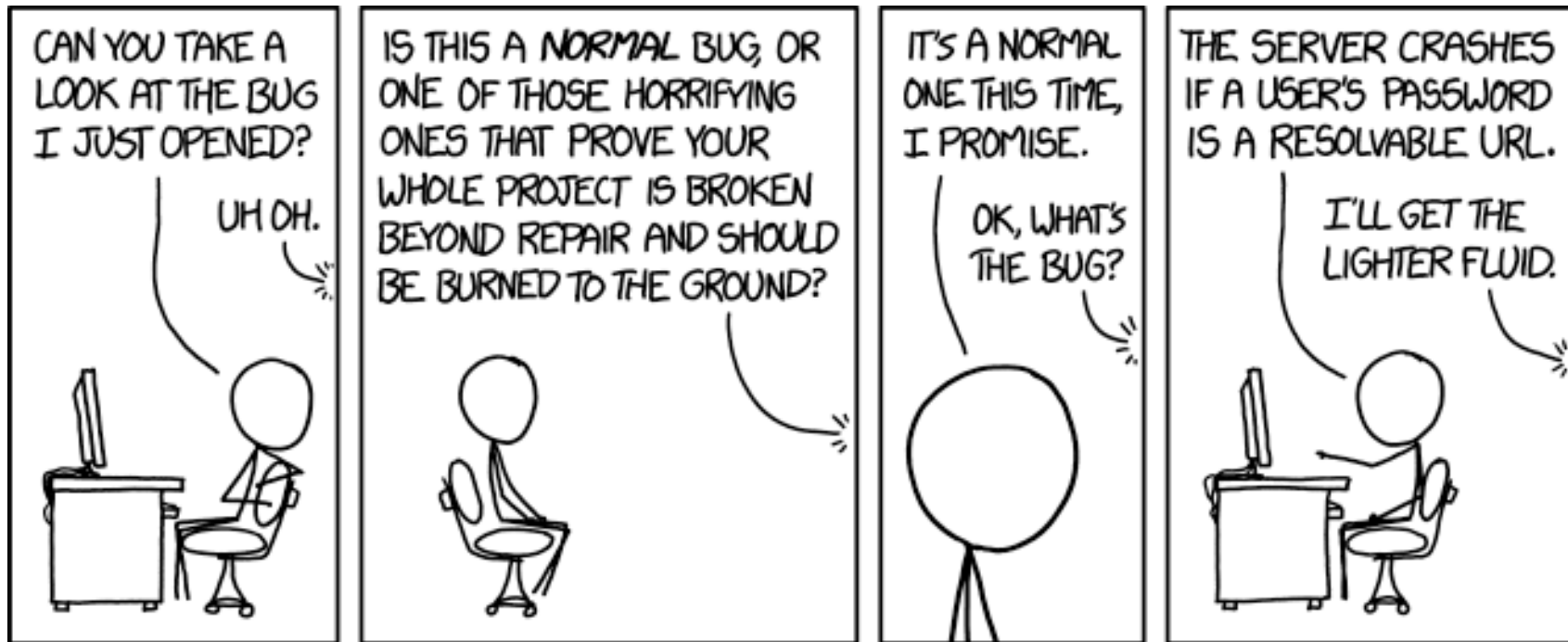
Your assignment

- The assignment document will describe everything in more detail.
 - Additional documentation and files can be found on asci.cgra.nl
- Tools are available to make energy, area and performance estimates.



One more thing...

- This is a research architecture...
 - Bugs will be present.
 - You will be among the first users.



Want to do the assignment?

Mark Wijtvliet (m.wijtvliet@tue.nl)
(or register at the forum at asci.cgra.nl and start directly)