Customising RI5CY: an open-source RISC-V Core

OSHCamp 2019

Graham Markall
(SW Toolchain Engineer)

Sam Leonard
(SW/HW Intern)

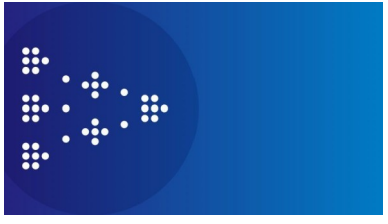
Embecosm Core Competences

Compiler Tool Chain Development

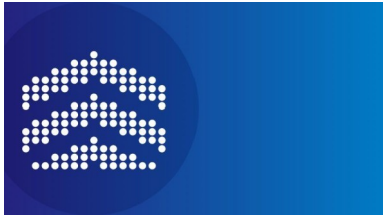
Hardware Modeling

Open Source Tool Support

Specialisms



Machine Learning Optimization



Superoptimization

Optimization for Energy Efficiency

Compilation for Security

EMBECOSM®

Outline / goals

- Talk today: groundwork
  - RISC-V ISA specifics
  - General concepts: C → Assembly → encoding
  - RI5CY microarchitecture
- Workshop tomorrow: practical
  - Practical step-by-step tutorial adding new instructions
- Choice: workshop tomorrow / online later

- RISC-V is an Instruction Set Architecture (ISA)
- Some other ISAs:

- RISC-V: Many open-source implementations (RI5CY etc.)

What's in an ISA document?

Registers:

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Instructions + Semantics

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register

Encodings

RV64I Base Instruction Set (in addition to RV32I)

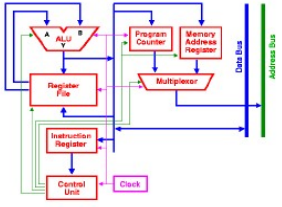
imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

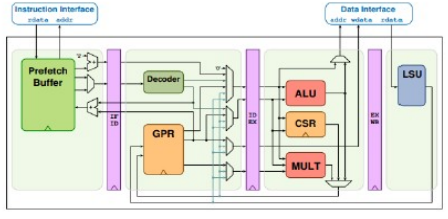
What's implemented around an ISA?

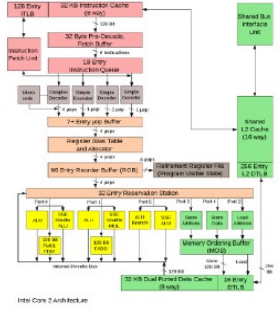
Software

ISA Specification

Hardware







(a bit of a) C program

```
int madd(int a, int x, int y)
{
  return a * x + y;
}
```

(a bit of a) C program - compiled

```
int madd(int a, int x, int y)
{
  return a * x + y;
}
```

```
madd:
        mul     a0,a0,a1
        add     a0,a0,a2
        ret
```

Assembled

```
00000000 <madd>:
   0:   02b50533          mul a0,a0,a1
   4:   00c50533          add a0,a0,a2
   8:   00008067          ret
```

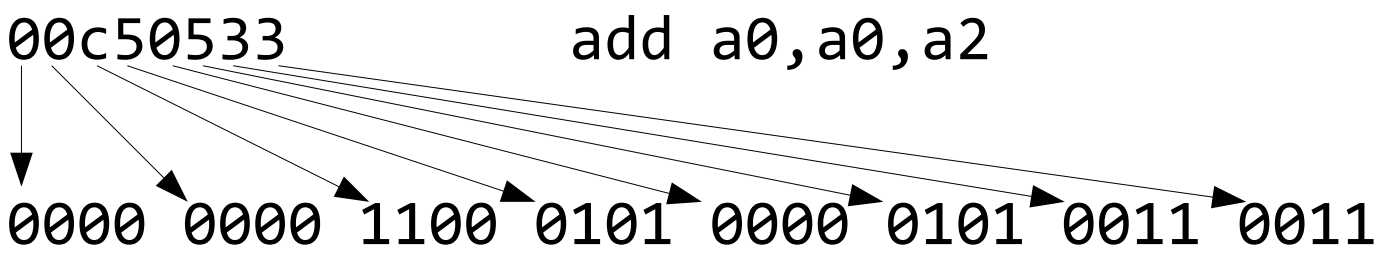
Assembled

```
00000000 <madd>:
   0:   02b50533          mul a0,a0,a1
   4:   00c50533          add a0,a0,a2
   8:   00008067          ret
```

Picking apart the encoding

```
00c50533          add a0,a0,a2
```

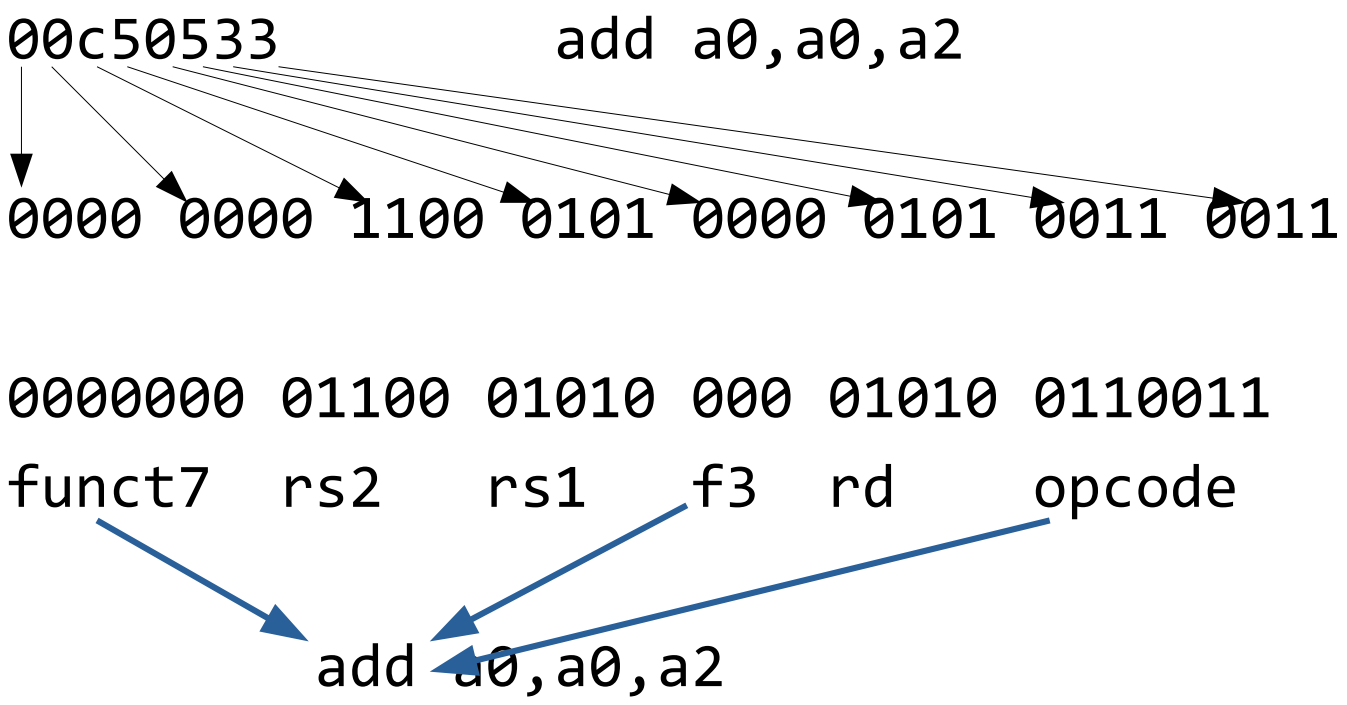
Picking apart the encoding



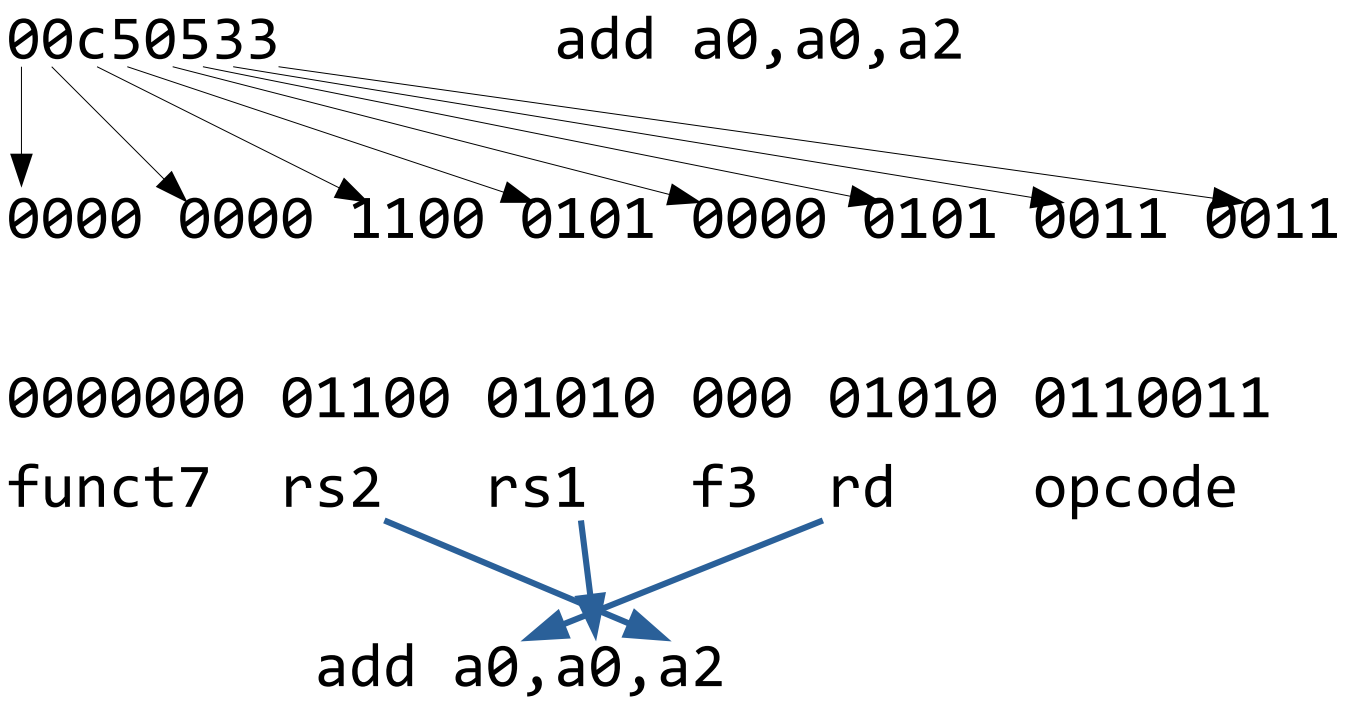
Picking apart the encoding


```
0000000 01100 01010 000 01010 0110011
funct7  rs2   rs1   f3  rd    opcode
```

Picking apart the encoding

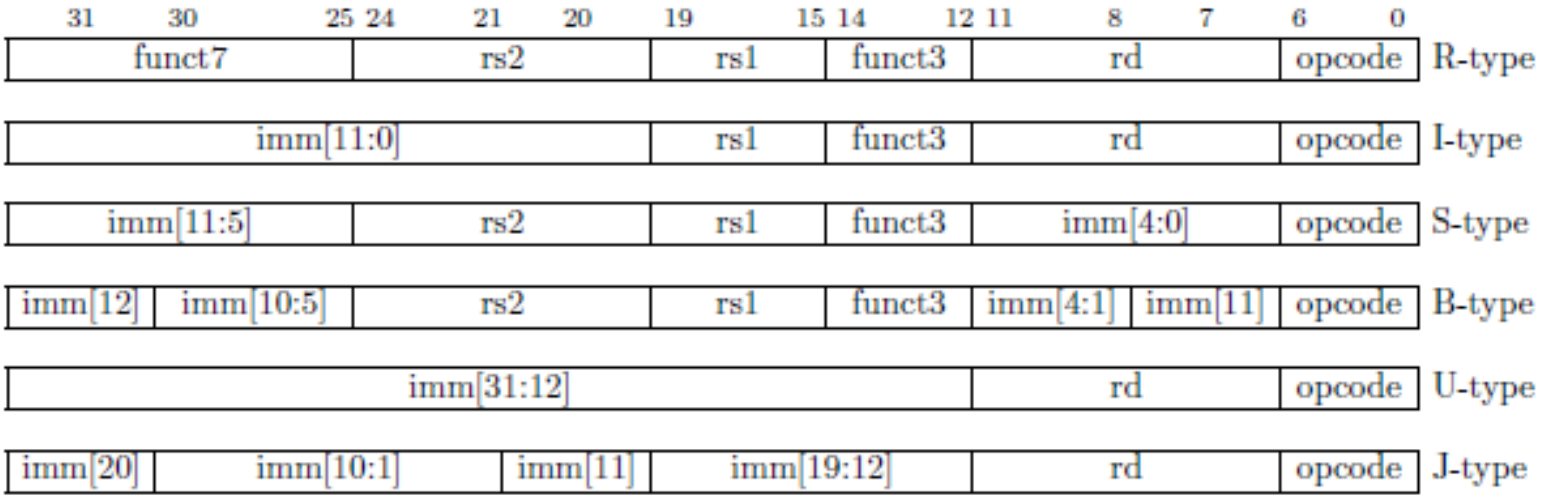


Picking apart the encoding



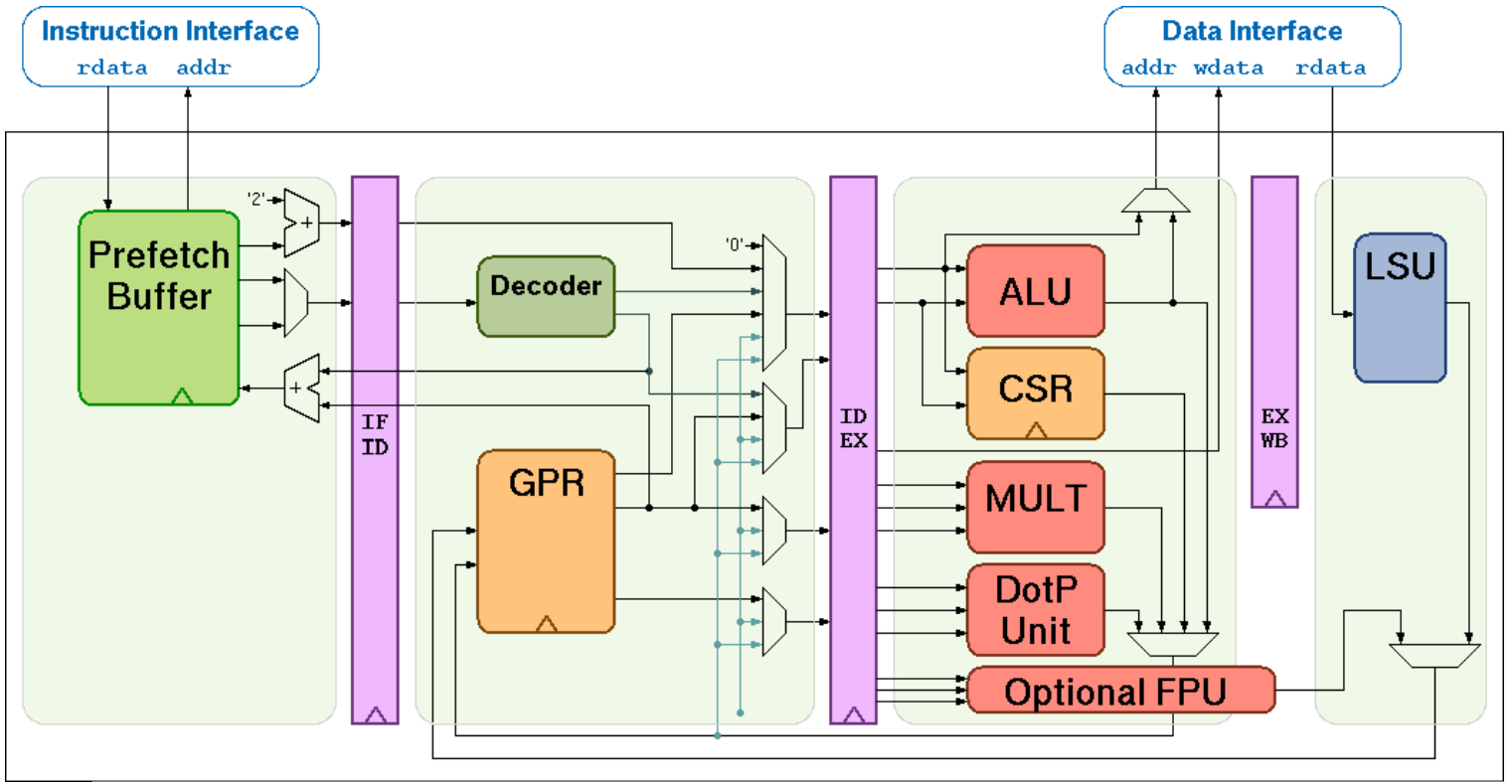
RISC-V Encoding formats

- More formats → more flexibility in specifying instructions
- More formats → greater decoder complexity

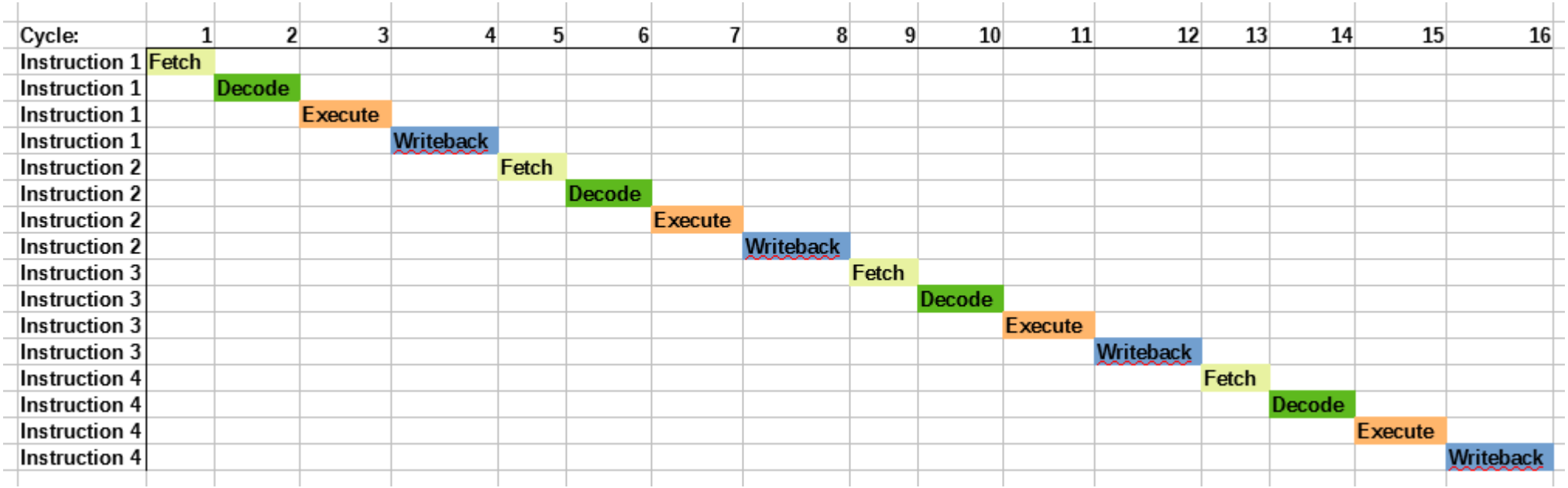


Execution – RI5CY pipeline

```
02b50533
00c50533
00008067
...
```



Executing 1 op per cycle



Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Instruction 1	Fetch															
Instruction 1		Decode														
Instruction 1			Execute													
Instruction 1				Writeback												
Instruction 2					Fetch											
Instruction 2						Decode										
Instruction 2							Execute									
Instruction 2								Writeback								
Instruction 3									Fetch							
Instruction 3										Decode						
Instruction 3											Execute					
Instruction 3												Writeback				
Instruction 4													Fetch			
Instruction 4														Decode		
Instruction 4															Execute	
Instruction 4																Writeback

- 1 operation per cycle
- 4 cycles per instruction

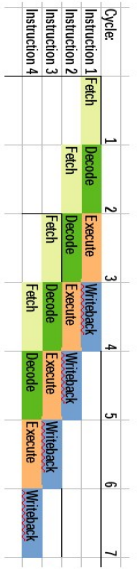
EMBECOSM®

Pipelined execution

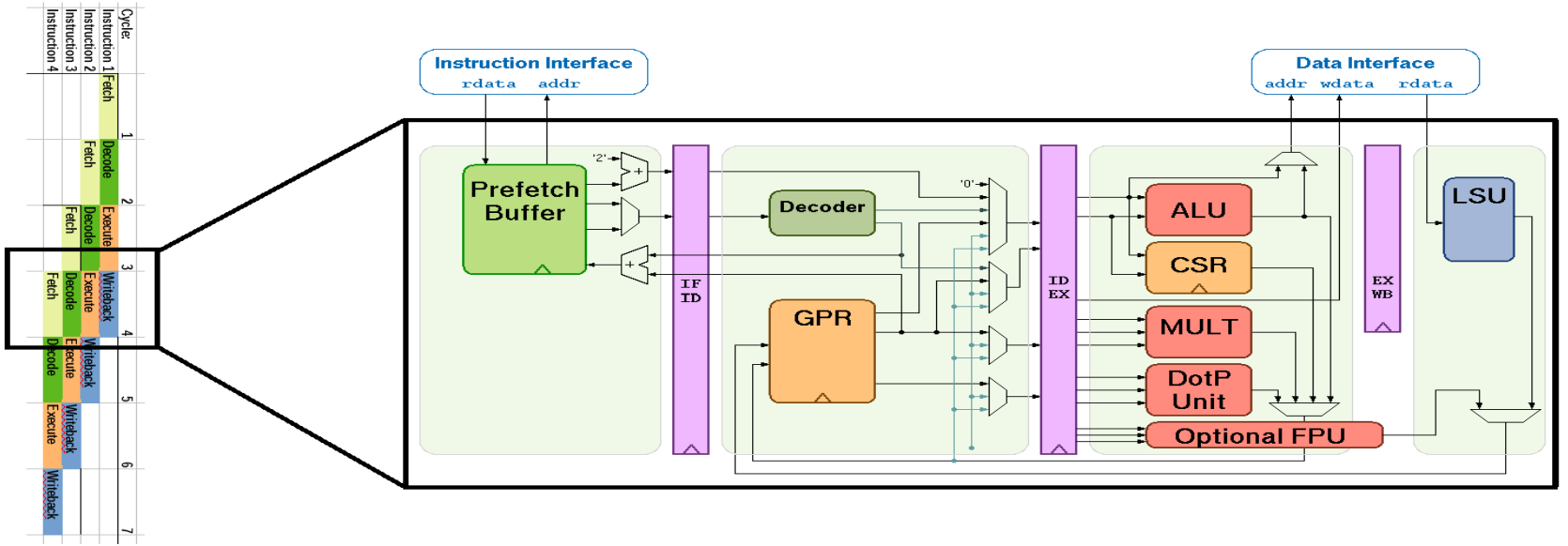
Cycle:	1	2	3	4	5	6	7
Instruction 1	Fetch	Decode	Execute	Writeback			
Instruction 2		Fetch	Decode	Execute	Writeback		
Instruction 3			Fetch	Decode	Execute	Writeback	
Instruction 4				Fetch	Decode	Execute	Writeback

- 4 operations per cycle
- 1 cycle per instruction

Pipelined execution



Pipelined execution



- Each unit kept busy on each cycle

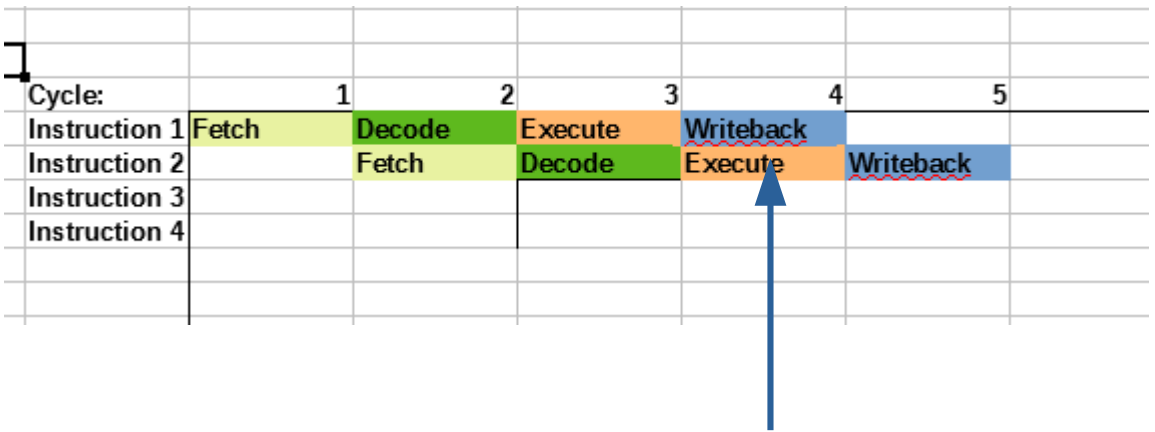
Hazards

```
lw a1, 0(a2)      # 1. load from address in a2
add a3, a1, a0    # 2. a1 + a0
```

Cycle:	1	2	3	4	5
Instruction 1	Fetch	Decode	Execute	Writeback	
Instruction 2		Fetch	Decode	Execute	Writeback
Instruction 3					
Instruction 4					

Hazards

```
lw a1, 0(a2)      # 1. load from address in a2
add a3, a1, a0    # 2. a1 + a0
```

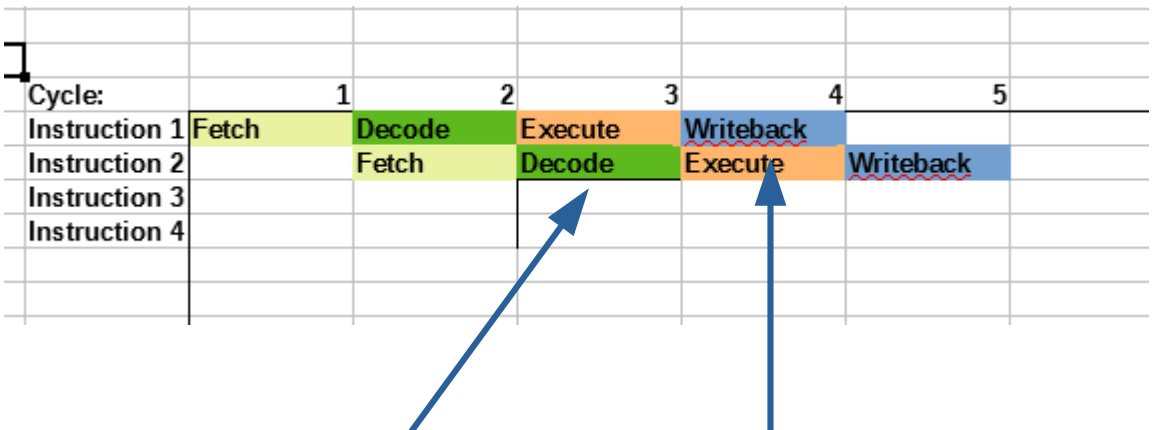


Cycle:	1	2	3	4	5
Instruction 1	Fetch	Decode	Execute	Writeback	
Instruction 2		Fetch	Decode	Execute	Writeback
Instruction 3					
Instruction 4					

Instr 1: Contents of a1 loaded on cycle 4, but...

Hazards

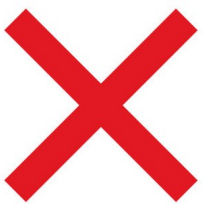
```
lw a1, 0(a2)      # 1. load from address in a2
add a3, a1, a0    # 2. a1 + a0
```



Cycle:	1	2	3	4	5
Instruction 1	Fetch	Decode	Execute	Writeback	
Instruction 2		Fetch	Decode	Execute	Writeback
Instruction 3					
Instruction 4					

Instr 1: Contents of a1 loaded on cycle 4, but...

Instr 2: need contents of a1 at cycle 3!



EMBECOSM®

Pipeline stall on hazard

```
lw a1, 0(a2)      # 1. load from address in a2
add a3, a1, a0    # 2. a1 + a0
```

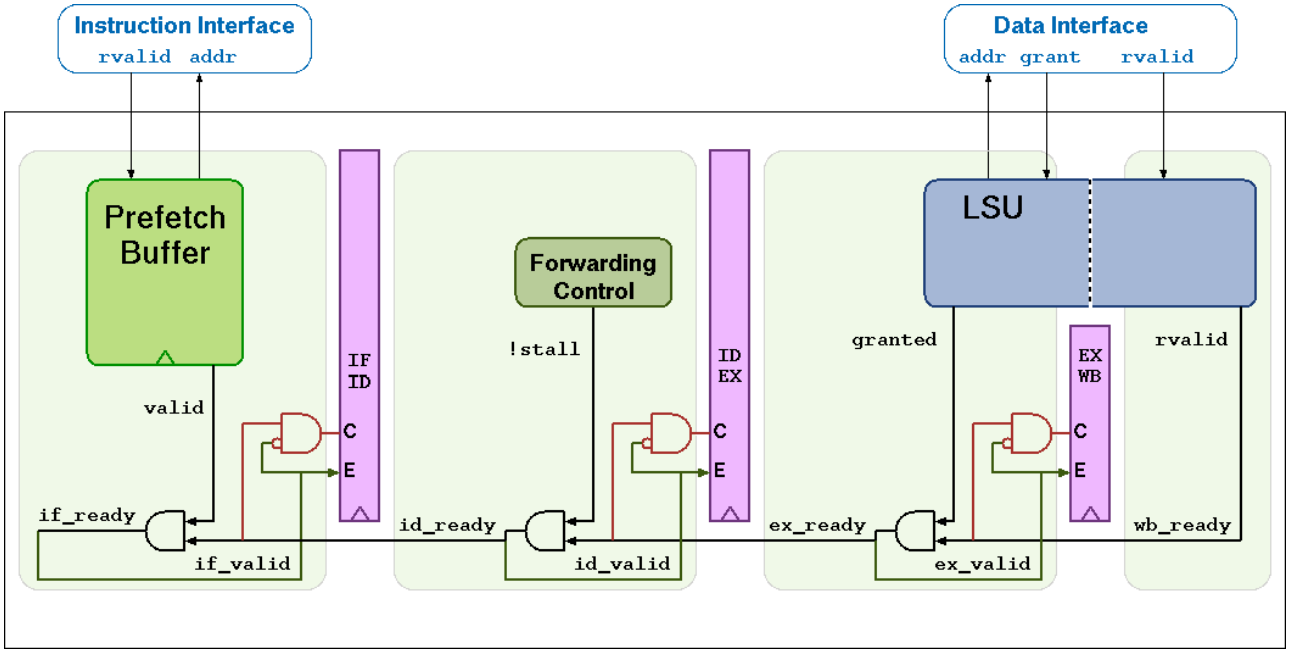
Cycle:	1	2	3	4	5	6	7
Instruction 1	Fetch	Decode	Execute	Writeback			
Instruction 2		Fetch	STALL	STALL	Decode	Execute	Writeback
Instruction 3							
Instruction 4							

Pipeline stall on multi cycle instruction

```
mul a3, a1, a2
add a5, a6, a7
```

Cycle:	1	2	3	4	5	6	7	8
Instruction 1	Fetch	Decode	Execute				Writeback	
Instruction 2		Fetch	Decode	STALL	STALL	STALL	Execute	Writeback
Instruction 3								
Instruction 4								

How does RI5CY stall?



- Signals between each stage to indicate ready / valid

Adding a new instruction

- Decide on syntax, semantics, and encoding
- Implement decoding
- Add an execution unit
- Hook up decoder to execution
- Hook up control signals
- Worked example through all these in the workshop
  - Step by step using simulation

Workshop options

- Workshop materials:
  - https://github.com/gmarkall/oshcamp-2019-workshop/
  - https://oshcampri5cyworkshop.slack.com
- Introduction to Verilog with Verilator:
  - https://gmarkall.wordpress.com/teaching/
- Chiphack (Intro to Verilog with FPGA):
  - http://chiphack.org/

Enjoy OSHCamp!

www.embecosm.com