Automatic generation of RISC-V cores through speculative loop pipelining

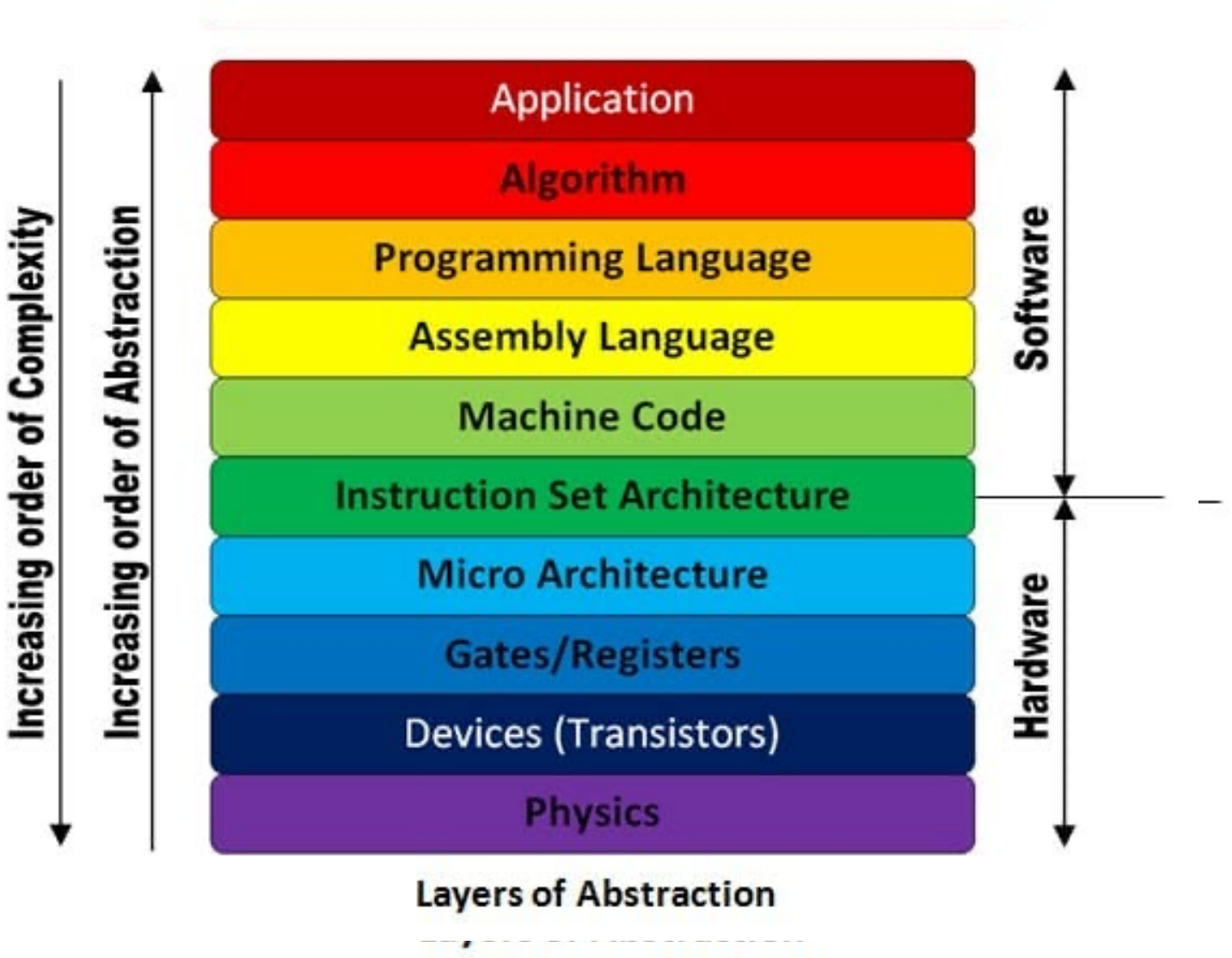
Colloque GDR-SoC

Steven Derrien

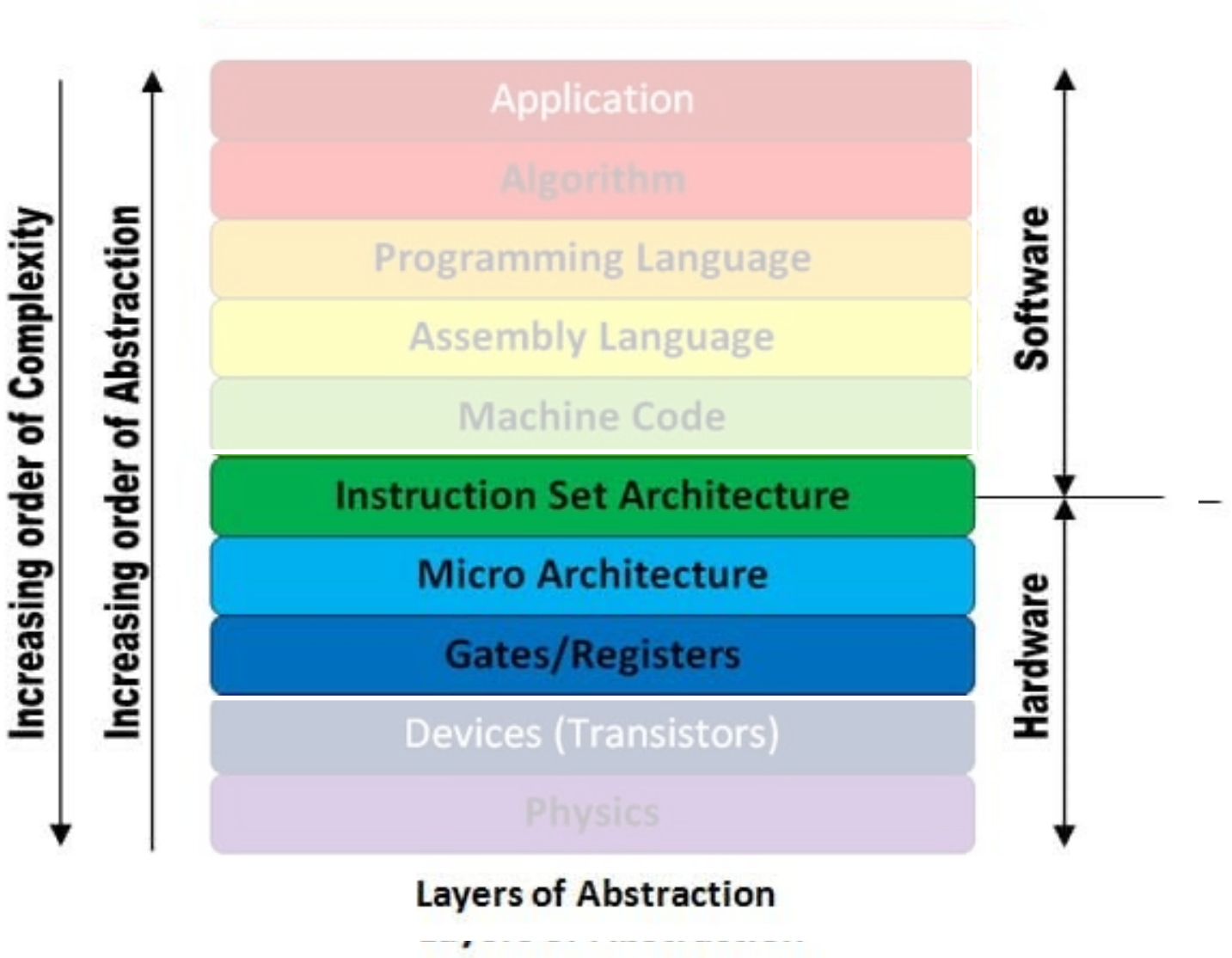
ARCAD Team, UBO/LabSTICC

(with contributions of former colleagues from IRISA/INRIA : Simon Rokicki, Jean-Michel Gorius[1], Dylan Leothaud, Thibaut Marty, Tomofumi Yuki)

Abstraction layers in computer systems



Abstraction layers in computer systems



Instruction Set Architectures

Paper specification

Executable specification

imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL

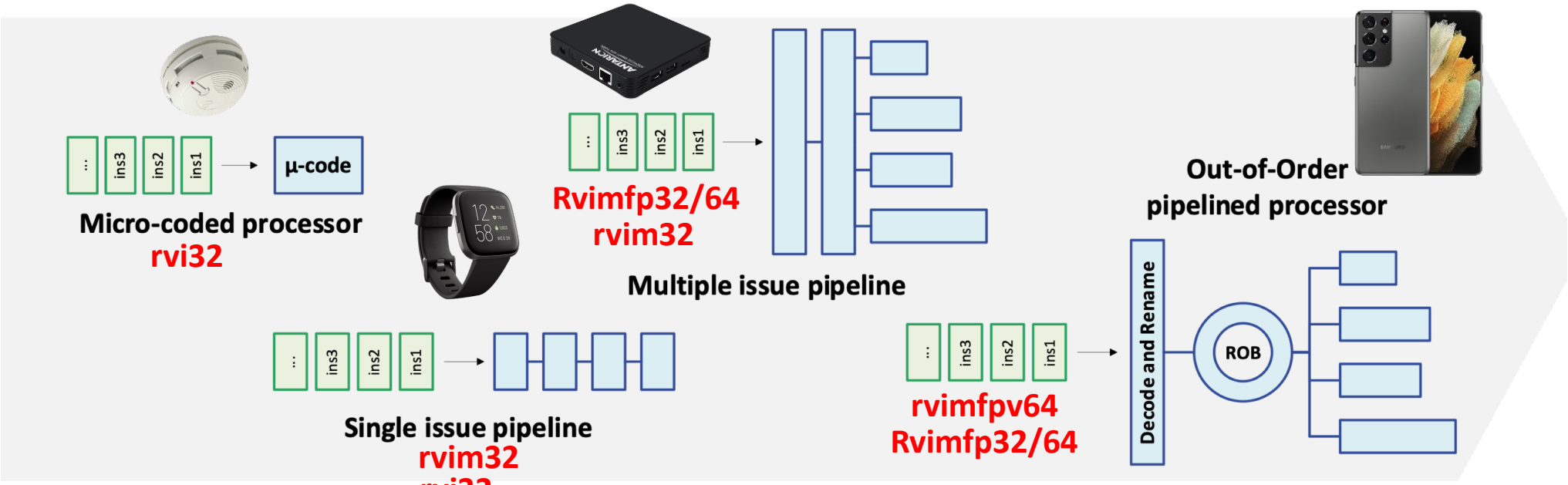
Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	`Regs[x1]←Regs[x2]+Regs[x3]`
`addi x1,x2,3`	Add immediate unsigned	`Regs[x1]←Regs[x2]+3`

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

Instruction Set Simulator (ISS)

Micro-architectures

- µarchitecture = *how* the machine executes instructions

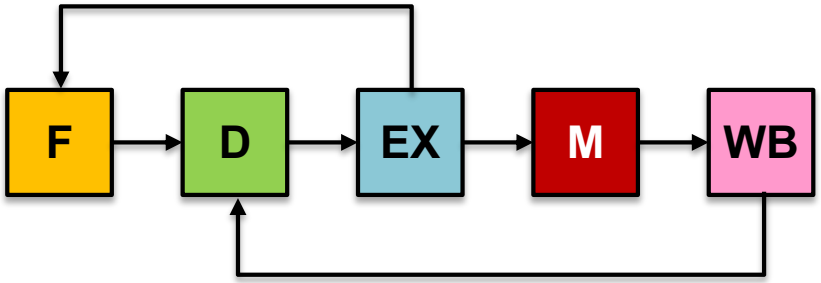


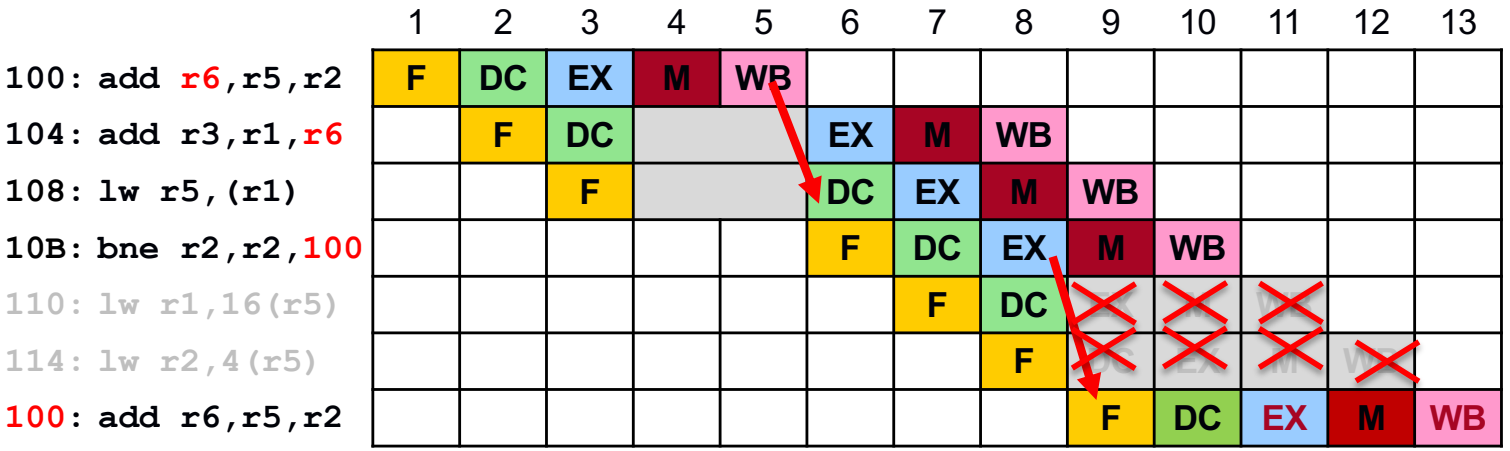
Takeaway

Need for highly customized micro-architecture (there is no *one size fits all*)

What is *processor* pipelining ?

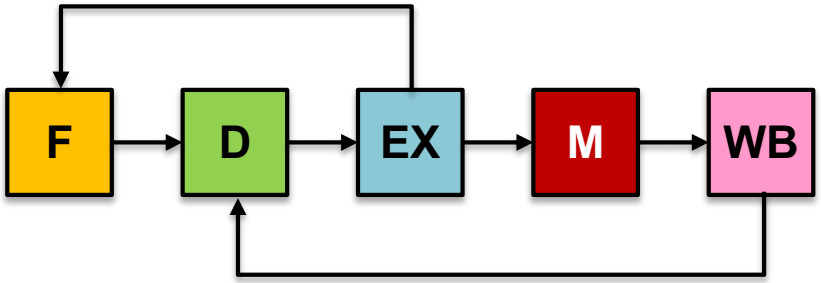
- **Overlapping the execution of consecutive instruction**
  - An instruction execution is decomposed into stages

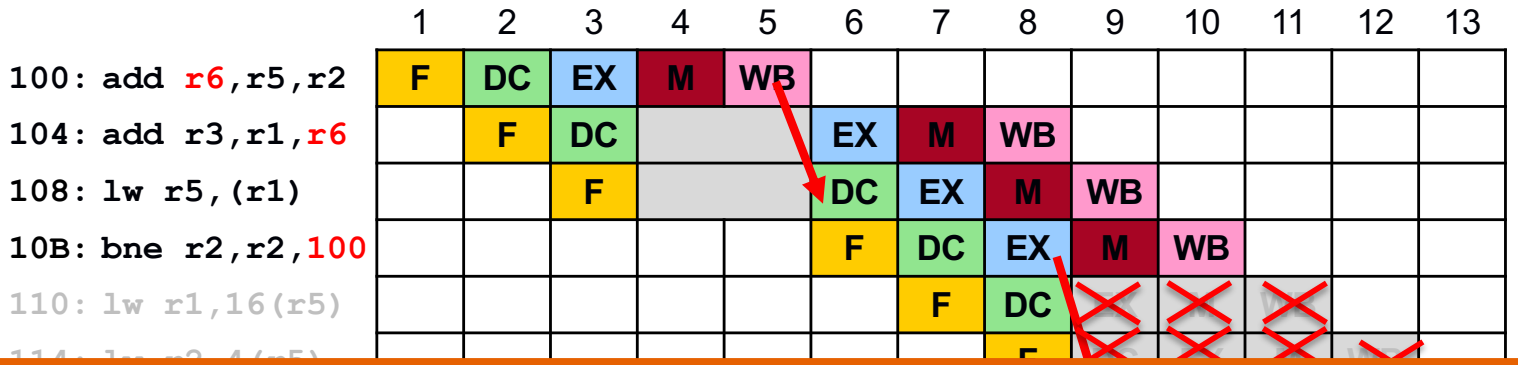




What is *processor* pipelining ?

- **Overlapping the execution of consecutive instruction**
  - An instruction execution is decomposed into stages



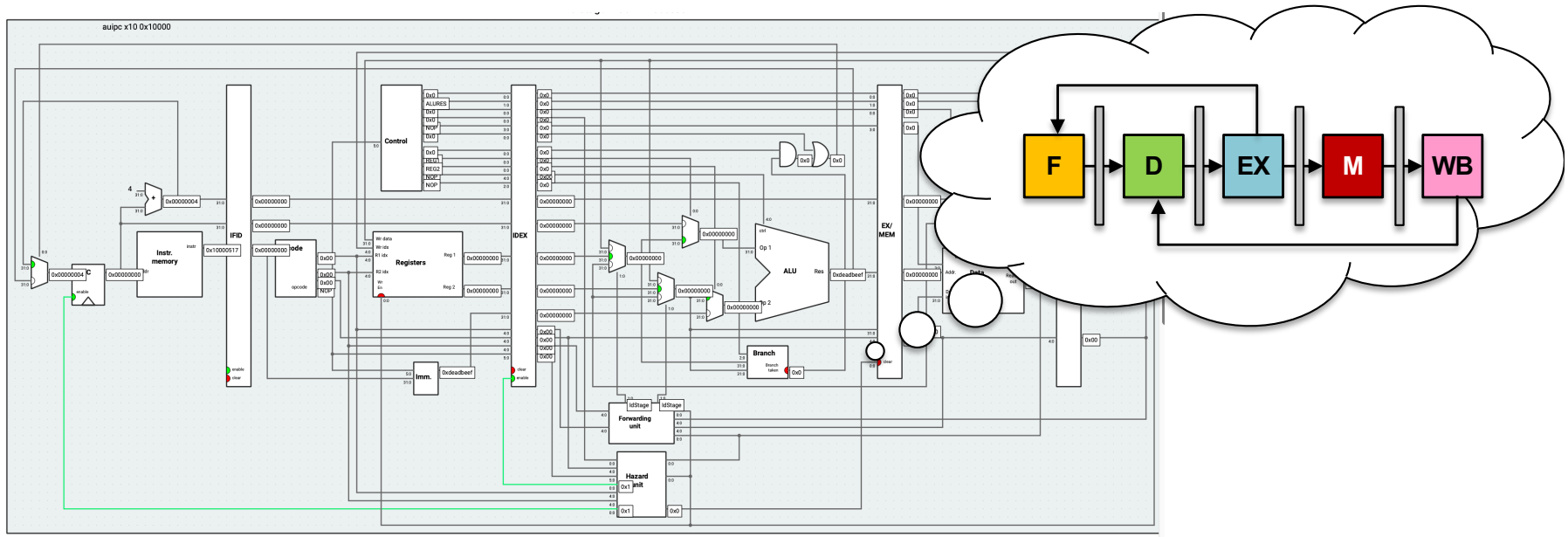


Takeaway

Hazard management is what make processor pipelining difficult

Implementing the processor as a circuit

- **Logic gates + registers implementing the chosen march**
  - Described using Hardware Description Languages



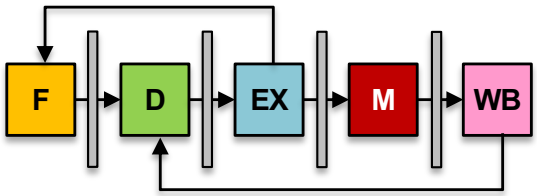
- **Must be optimized for energy, area and clock speed**
  - and must still be faithful to the reference marchitecture !

Existing CPU Design Flows

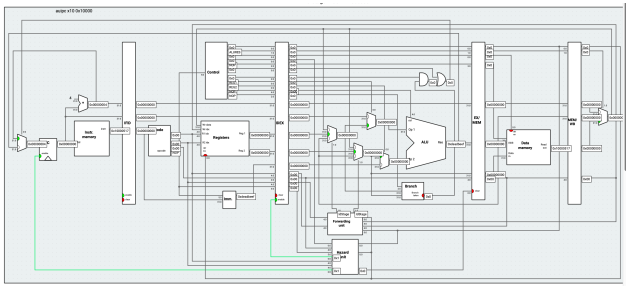
Instruction set

Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	Regs[x1]←Regs[x2]+Regs[x3]
`addi x1,x2,3`	Add immediate unsigned	Regs[x1]←Regs[x2]+3
`lui x1,42`	Load upper immediate	Regs[x1]←0^{32}##42##0^{12}
`sll x1,x2,5`	Shift left logical	Regs[x1]←Regs[x2]<<5
`slt x1,x2,x3`	Set less than	if (Regs[x2]<Regs[x3]) Regs[x1]←1 else Regs[x1]←0

Micro-architecture



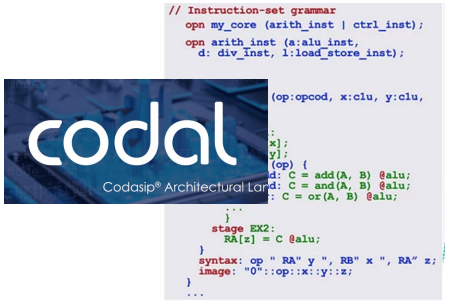
Circuit



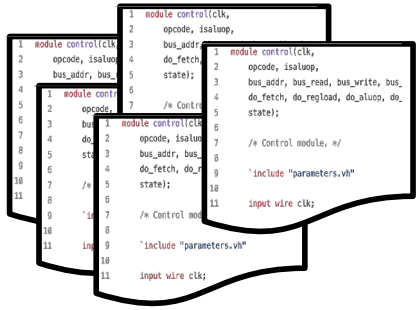
DSL specification

```
while (1) {
    opcode,ra,rb,rc,imm = mem[pc];
    pc++;
    switch (opcode) {
        case LOAD: reg[rc] = mem[reg[ra] + imm]; break;
        case ADD: reg[rc] = reg[ra] + reg[rb]; break;
        case JUMP: pc = reg[ra]; break;
        case BNZ: pc = rb ? reg[ra] : pc; break;
            ...
    }
}
```

DSL specification



Verilog, Chisel



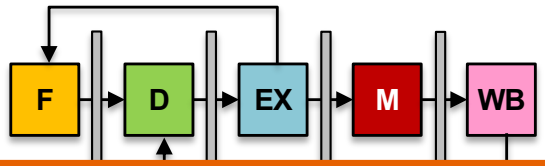
DSL = Domain Specific Languages

Existing CPU Design Flows

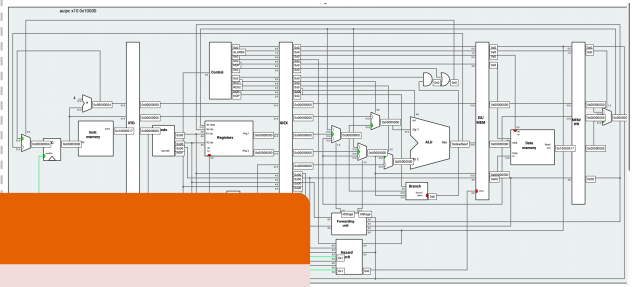
Instruction set

Example instrucmtion	Instruction name	Meaning
add x1,x2,x3	Add	Regs[x1]←Regs[x2]+Regs[x3]
addi x1,x2,3	Add immediate unsigned	Regs[x1]←Regs[x2]+3
lui x1,42	Load upper immediate	Regs[x1]←0^{32}##42##0^{12}
sll x1,x2,5	Shift left logical	Regs[x1]←Regs[x2]<<5
slt x1,x2,x3	Set less than	if (Regs[x1]

Micro-architecture



Circuit



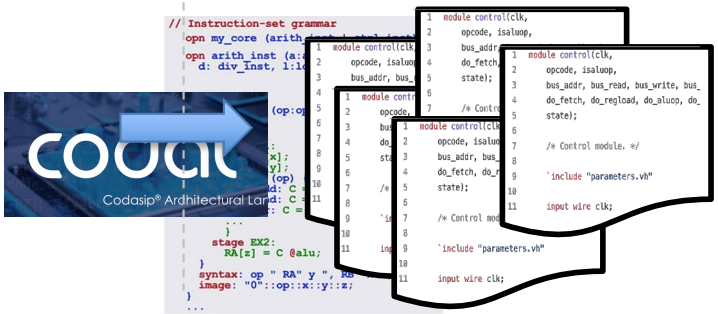
Takeaway

In practice, micro architectural DSLs are RTL in disguise

DSL specification

```
while (1) {
    opcode,ra,rb,rc,imm = mem[pc];
    pc++;
    switch (opcode) {
        case LOAD: reg[rc] = mem[reg[ra] + imm]; break;
        case ADD: reg[rc] = reg[ra] + reg[rb]; break;
        case JUMP: pc = reg[ra]; break;
        case BNZ: pc = rb ? reg[ra] : pc; break;
            ...
    }
}
```

DSL specificati… Verilog, Chisel



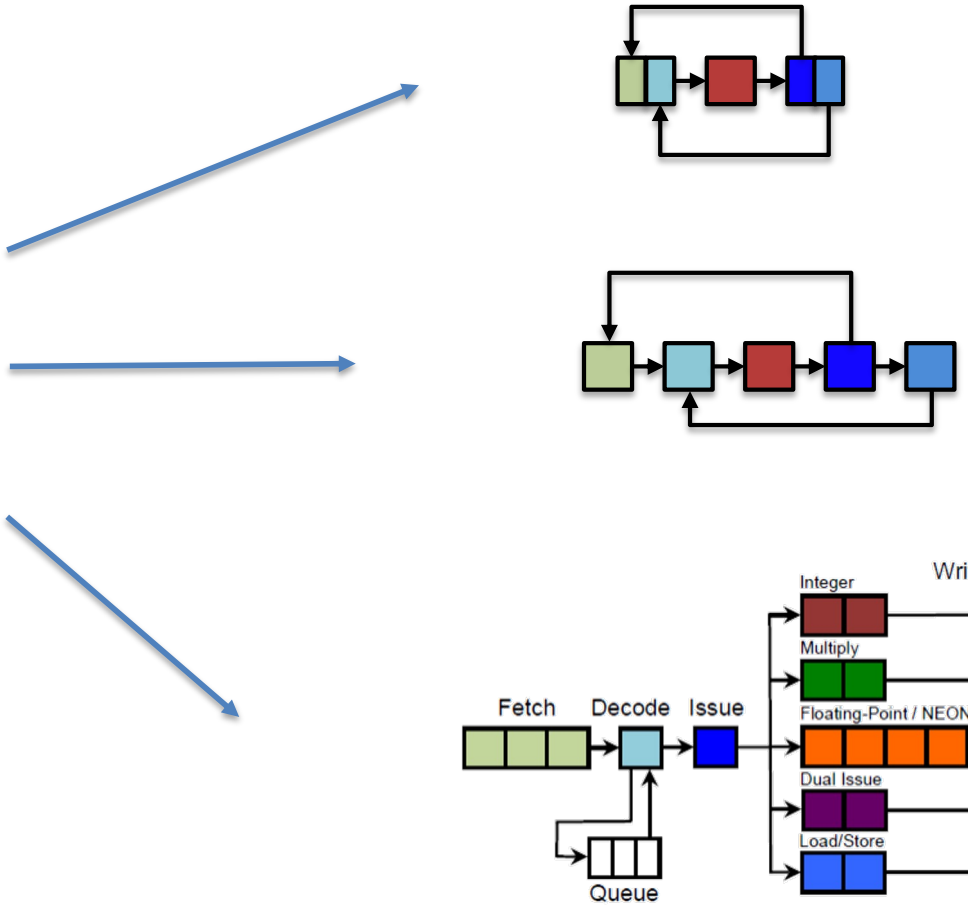
DSL = Domain Specific Languages

Punchline : let's try to *compile* CPUs

- **Generate CPU circuits straight out from the ISA simulator!**

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

Instruction Set Simulator (ISS)

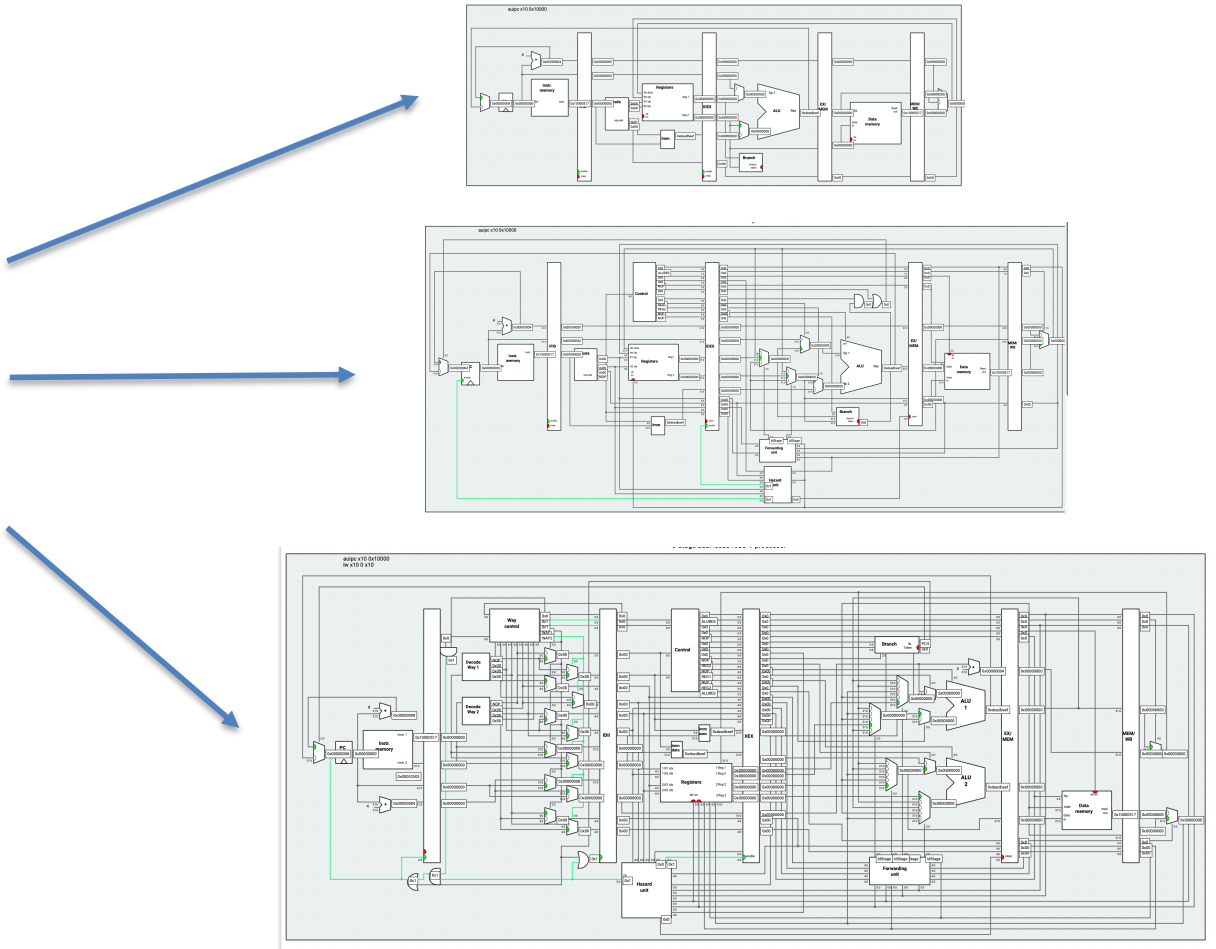


Punchline : let's try to *compile* CPUs

- **Generate CPU circuits straight out from the ISA simulator!**

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

Instruction Set Simulator (ISS)

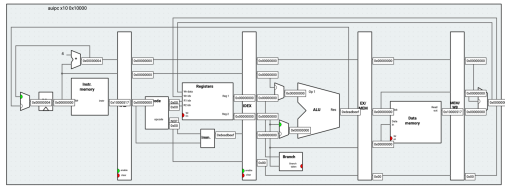


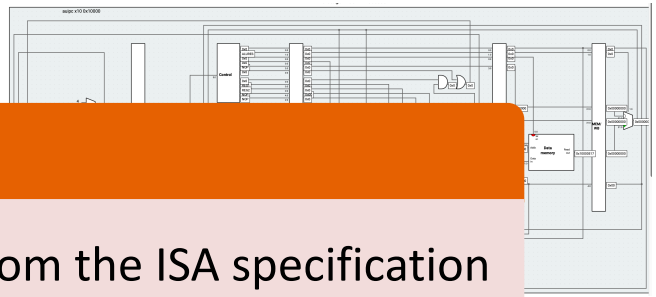
Punchline : let's try to *compile* CPUs

- **Generate CPU circuits straight out from the ISA simulator!**

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI:
   break;
  case LDW: X
   break;
  case STW: M
   break;
  case BEQ:
   nxtPC += X
   break;
  // ...
 }
 PC = nxtPC;
}
```

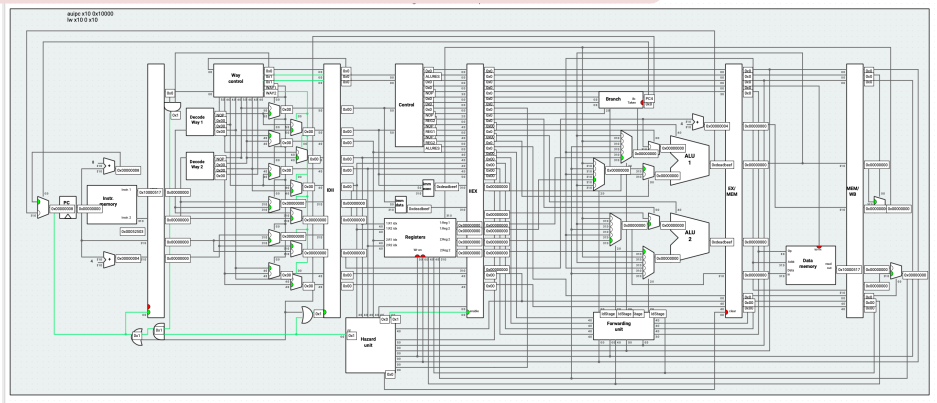
Instruction Set Simulator (ISS)





Key challenge

We need to infer a microarchitecture from the ISA specification



Wait, can’t I already do that ?

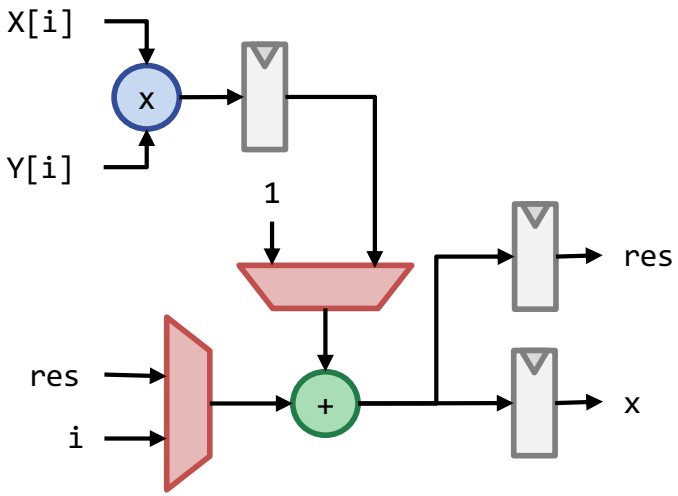
- Yes, High Level Synthesis tools generate circuits from C/C++

Synthesizing circuits from an algorithmic specification

Efficient design synthesis for regular access patterns and computations

```
int X[N], Y[N];
int tmp, res = 0;
#pragma HLS mult=1, adder=1
for(int i = 0; i < N; ++i) {
    tmp = X[i] * Y[i];
    res += tmp;
}
```

HLS

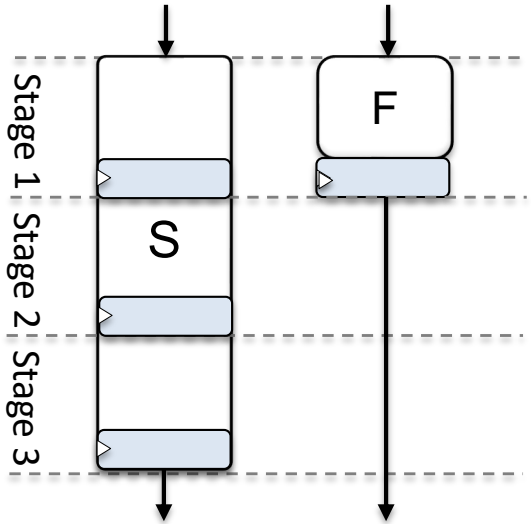


Automatic pipelining in HLS

- **Overlaps operations from distinct iterations**

```
for(i=4;i<…;i++) {
  X[i]= F(Z[i]);
  Y[i]= S(Z[i]);
}
```

loop kernel



Datapath

X[i]= F(Z[i]);

F	F_2	F_3	F_4	F_5	F_6

S_1	S_2	S_3	S_4	S_5	S_6
	S_1	S_2	S_3	S_4	S_5
		S_1	S_2	S_3	S_4

Y[i]= S(Z[i]);

Execution trace

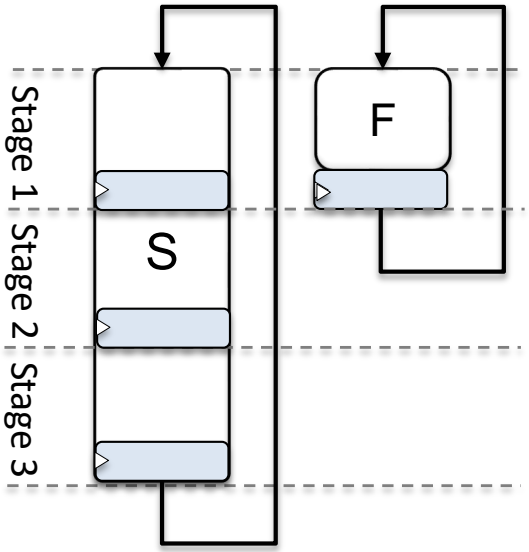
- ***May*** **result in high clock speed, low CPI**

Pipelining and reuse distance

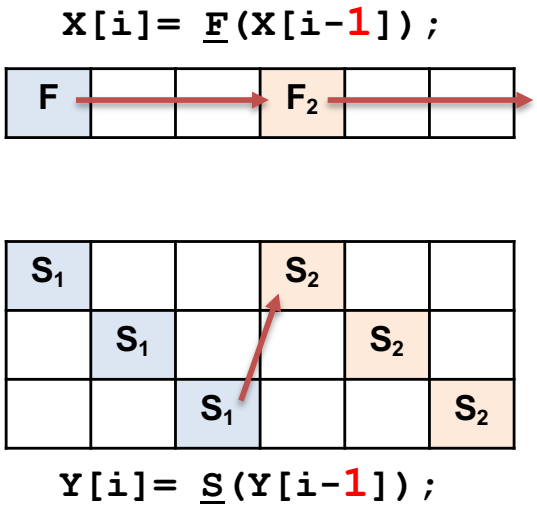
- **Overlapping distinct iterations is not always possible**

```
for(i=4;i<…;i++) {
  X[i]= F(X[i-1]);
  Y[i]= S(Y[i-1]);
}
```

loop kernel



Datapath



Execution trace

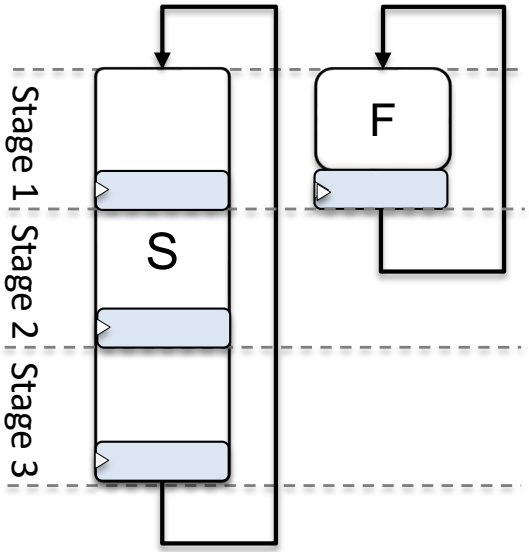
- **Constrained by loop carried dependencies**

Pipelining and reuse distance

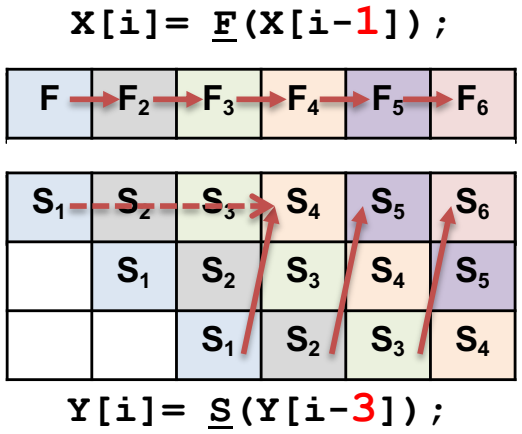
- What if the reuse distance is >1 ?

```
for(i=4;i<…;i++) {
  X[i]= F(X[i-1]);
  Y[i]= S(Y[i-3]);
}
```

loop kernel



Datapath



Execution trace

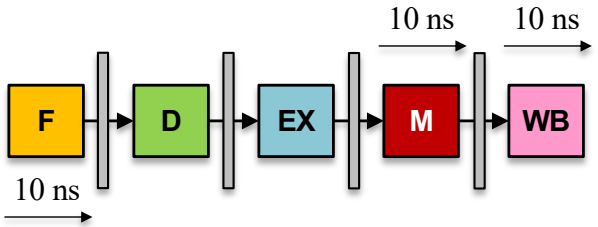
Reuse distance ⟹ pipeline depth ⟹ clock speed T_{clk}

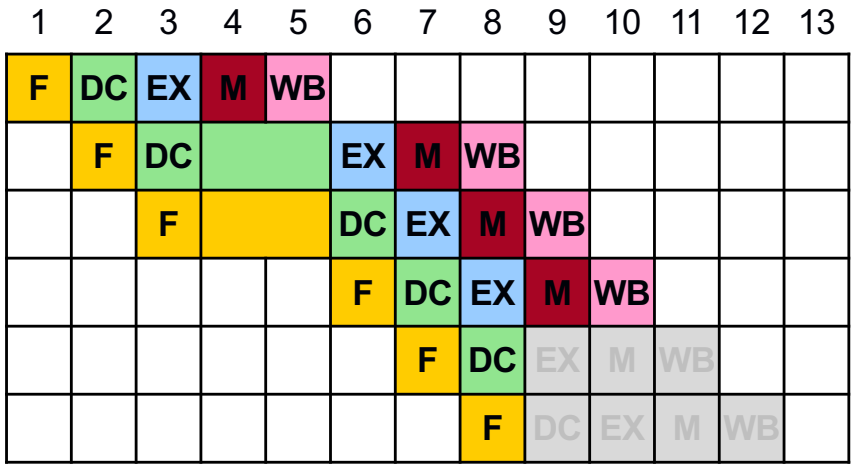
Synthesizing an ISS using HLS

- **Taking advantage of pipelining capabilities of HLS tools**
  - Based on “software pipelining”, a well known compiler optimization

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

T_{clk}=10ns



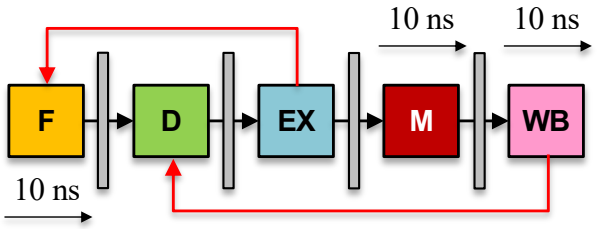


Synthesizing an ISS using HLS

- **Taking advantage of pipelining capabilities of HLS tools**
  - Based on “software pipelining”, a well known compiler optimization

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

T_{clk}=10ns



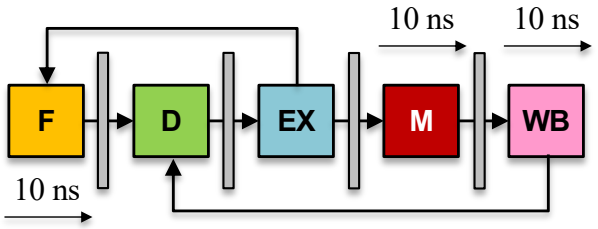
1	2	3	4	5	6	7	8	9	10	11	12	13
F	DC	EX	M	WB								
					F	DC	EX	M	WB			
										F	DC	EX

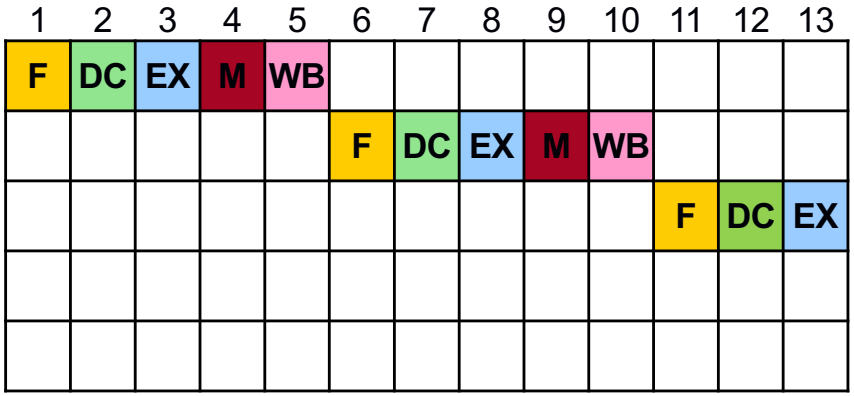
Synthesizing an ISS using HLS

- **Taking advantage of pipelining capabilities of HLS tools**
  - Based on “software pipelining”, a well known compiler optimization

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

T_{clk}=10ns





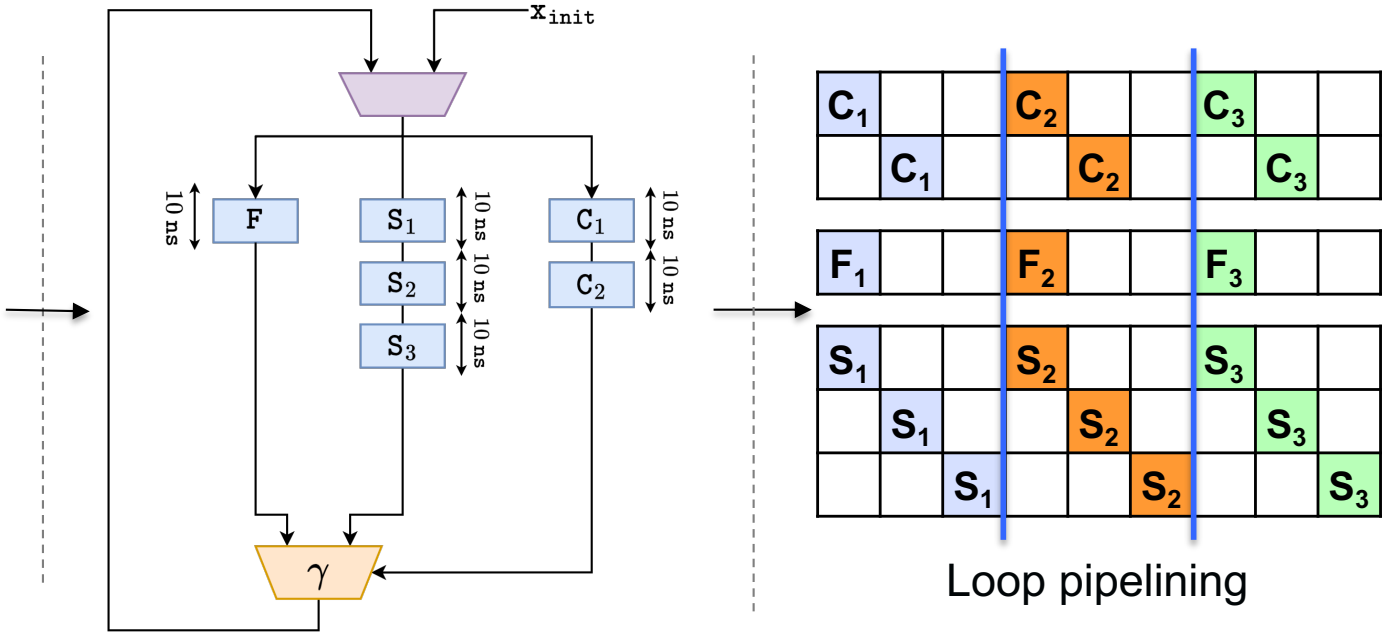
Takeaway

Existing static HLS scheduling techniques are unfit for CPU design

Speculative Loop Pipelining (SLP)

- **Extends classic loop pipelining with speculative execution**
  - Speculatively overlap the execution of distinct iterations

```
do {
  if(C(x)) {
    // slow
    x = S(x);
  } else {
    // fast
    x = F(x);
  }
} while (…)
```

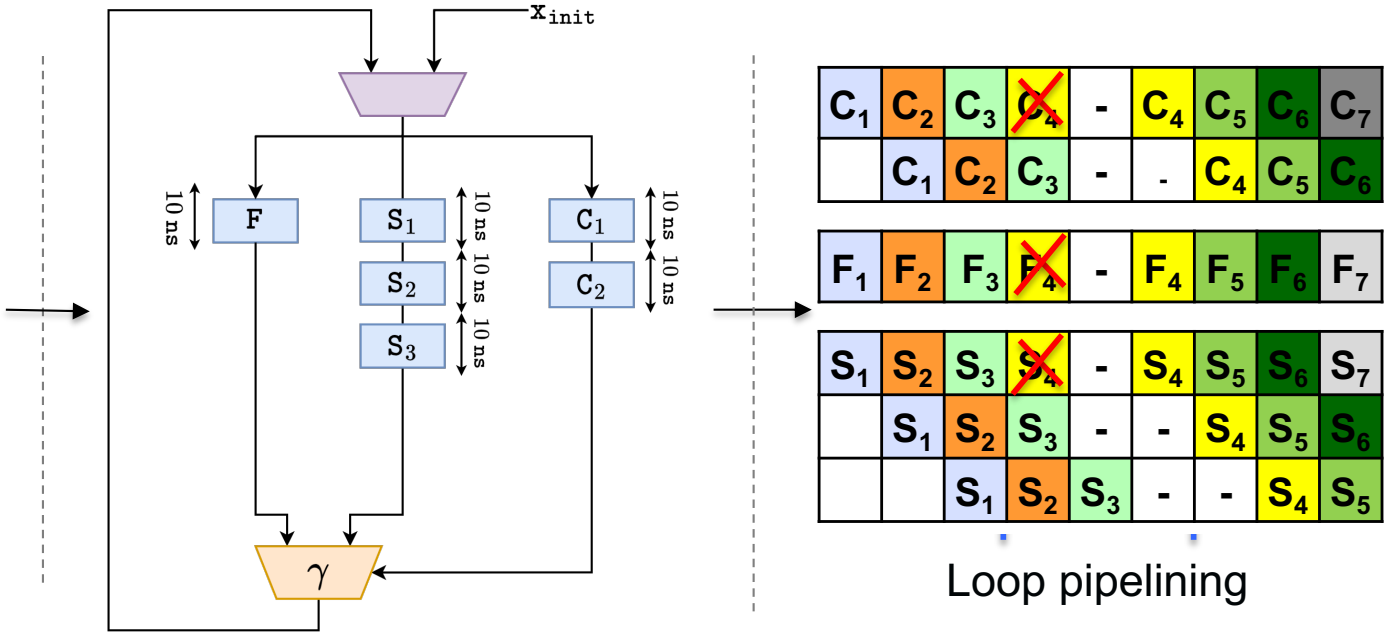


- **Requires a complete rehaul of HLS scheduling techniques**

Speculative Loop Pipelining (SLP)

- **Extends classic loop pipelining with speculative execution**
  - Speculatively overlap the execution of distinct iterations

```
do {
  if(C(x)) {
    // slow
    x = S(x);
  } else {
    // fast
    x = F(x);
  }
} while (…)
```



- **Requires a complete rehaul of HLS scheduling techniques**

Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. “Toward Speculative Loop Pipelining for High-Level Synthesis”. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39.11 (Nov. 2020), pp. 4229–4239.

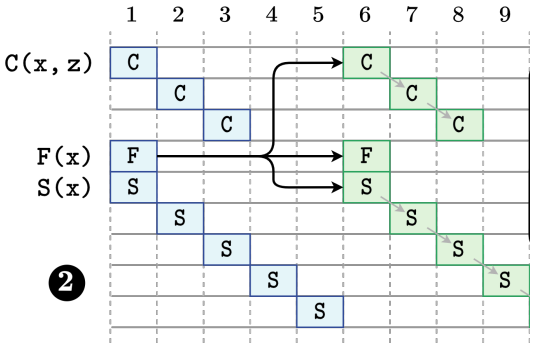
SpecHLS : a tool for SLP in Vitis HLS

Input code

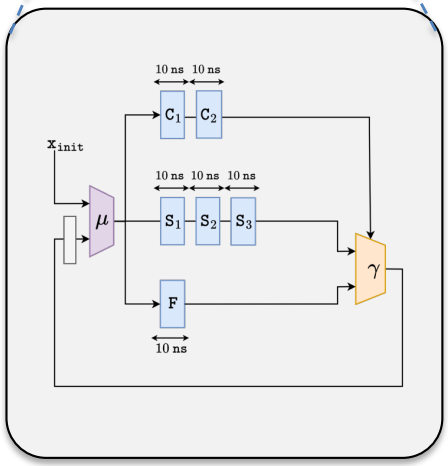
```
do {
  if(C(x, z))
    x = S(x);
  else
    x = F(x);
} while(!x);
```

❶

XILINX VITIS



SpecHLS

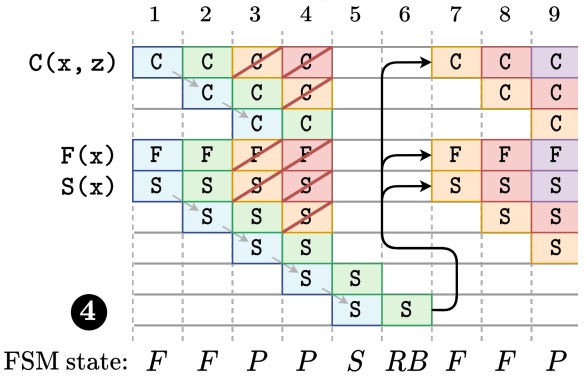


Program IR

```
#pragma hls distance mis_x=5
#pragma hls distance ctrl=3
do {
#pragma hls pipeline II=1
  ctrl[t] = C(s_x[t-1], z[t-1]);
  mis_x[t] = S(s_x[t-1]);
  cs = nextstate(cs, ctrl[t-3]);
  s_x[t] = cs.selSlow ?
             mis_x[t-5] :
             F(s_x[t-1]);
  if(cs.rollback) {
    s_x[t] = s_x[t-5];
    z[t] = z[t-5];
  }
  if(cs.commit)
    x = s_x[t-4];
  t += 1;
} while(!(x && cs.commit));
```

❸

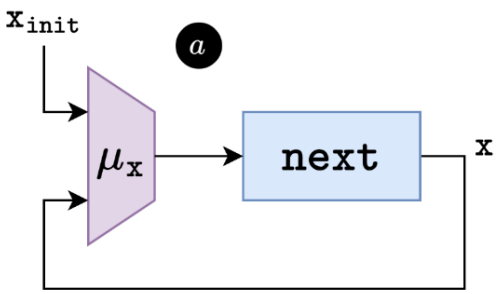
XILINX VITIS



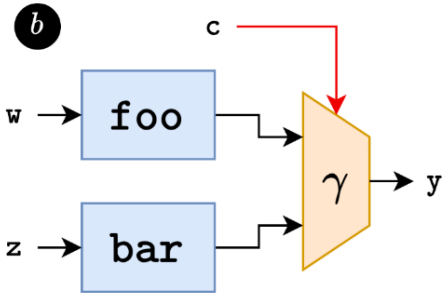
Program representation in Spec-HLS

- **The way the compiler sees the program to optimize**
  - Ours is a variant of Gated Static Single Assignment (GSSA)

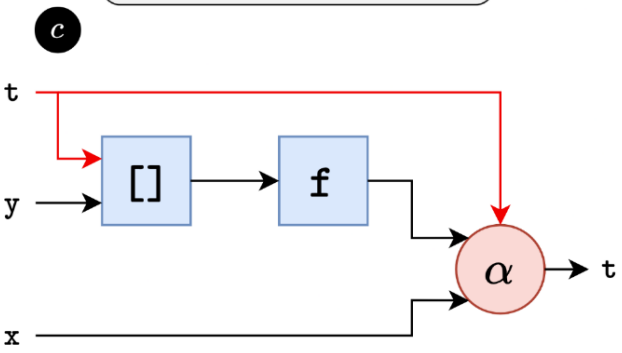
```
do {
  x = next(x);
} while (1);
```




```
if (c)
  y = foo(w);
else
  y = bar(z);
```




```
if(we)
  t[x] = f(t[y]);
```



[1]Peng Tu and David Padua, "Gated SSA-based demand-driven symbolic analysis for parallelizing compilers," in *Proceedings of the 9th international conference on Supercomputing (ICS '95)*, 1995

Takeaway

In GSSA, γ nodes capture speculation opportunities

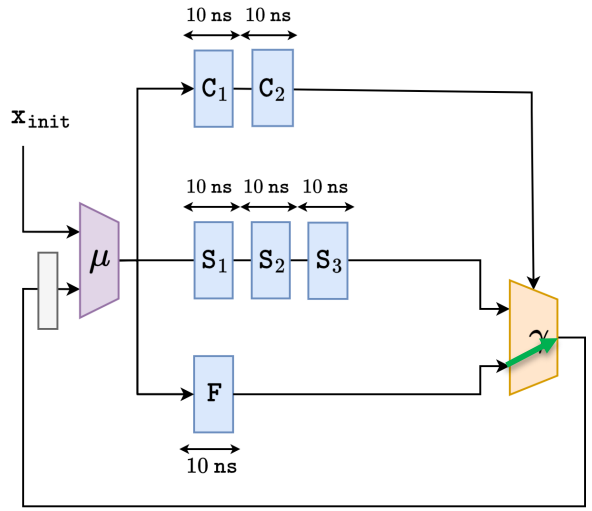
Speculative Loop Pipelining (SLP)

- **Building the program IR + identify speculation points**

Input code

```
do {
  if(C(x, z))
    x = S(x);
  else
    x = F(x);
} while(!x);
```

❶



Speculation decision based on γ profiling

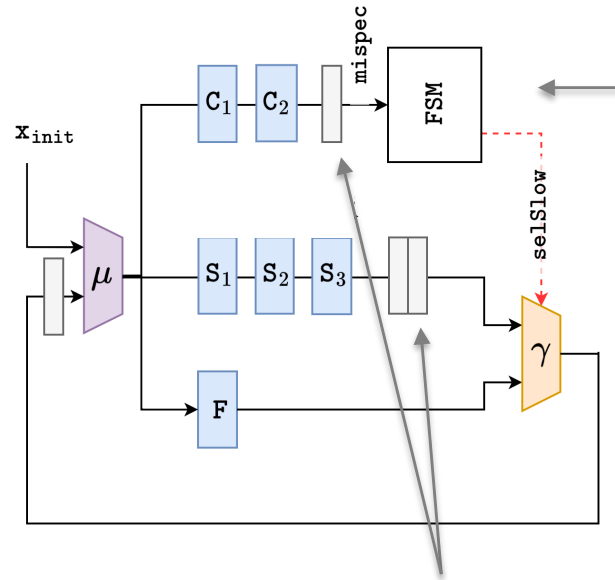
Steven Derrien, Thibaut Marty, Simon Rokicki, Tomofumi Yuki. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2020

Speculative Loop Pipelining (SLP)

- Adding delays + FSM to enable speculative pipelining

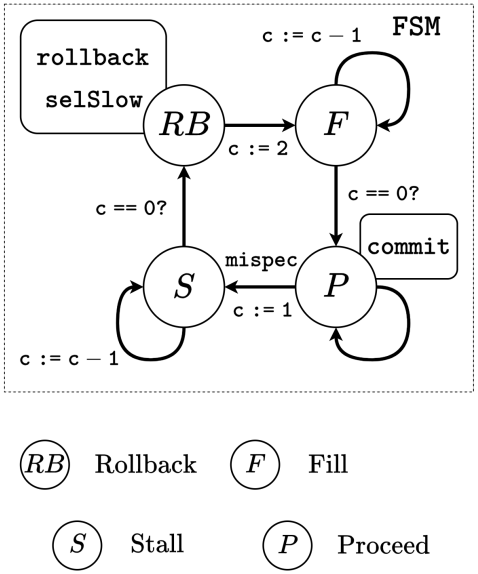
Input code

```
do {
  if(C(x, z))
    x = S(x);
  else
    x = F(x);
} while(!x);
```



Virtual pipeline stage registers

FSM



Steven Derrien, Thibaut Marty, Simon Rokicki, Tomofumi Yuki. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2020

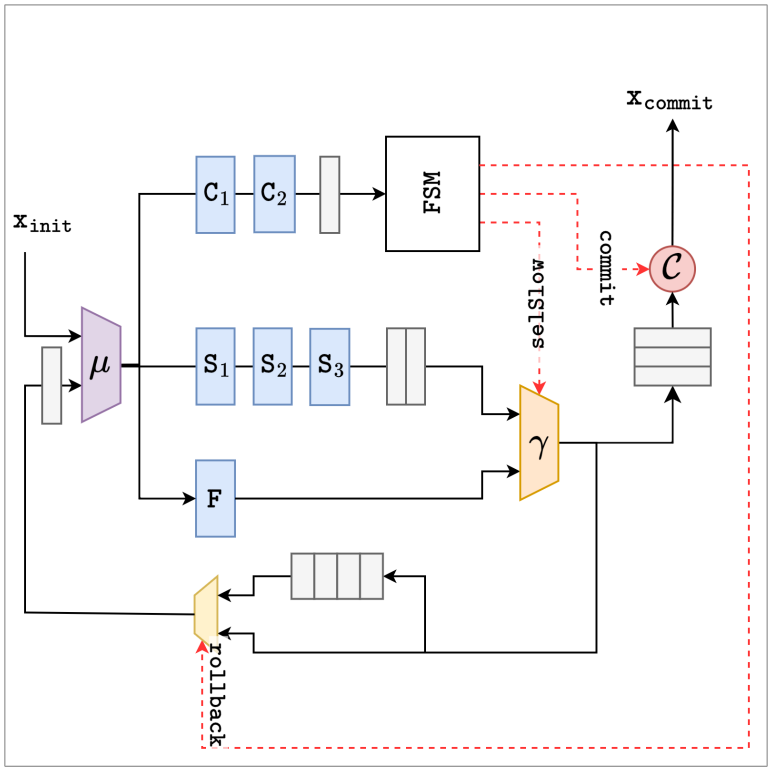
Speculative Loop Pipelining (SLP)

- Insert commit + rollback logic to fully handle hazards

Input code

```
do {
  if(C(x, z))
    x = S(x);
  else
    x = F(x);
} while(!x);
```

❶



Steven Derrien, Thibaut Marty, Simon Rokicki, Tomofumi Yuki. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2020

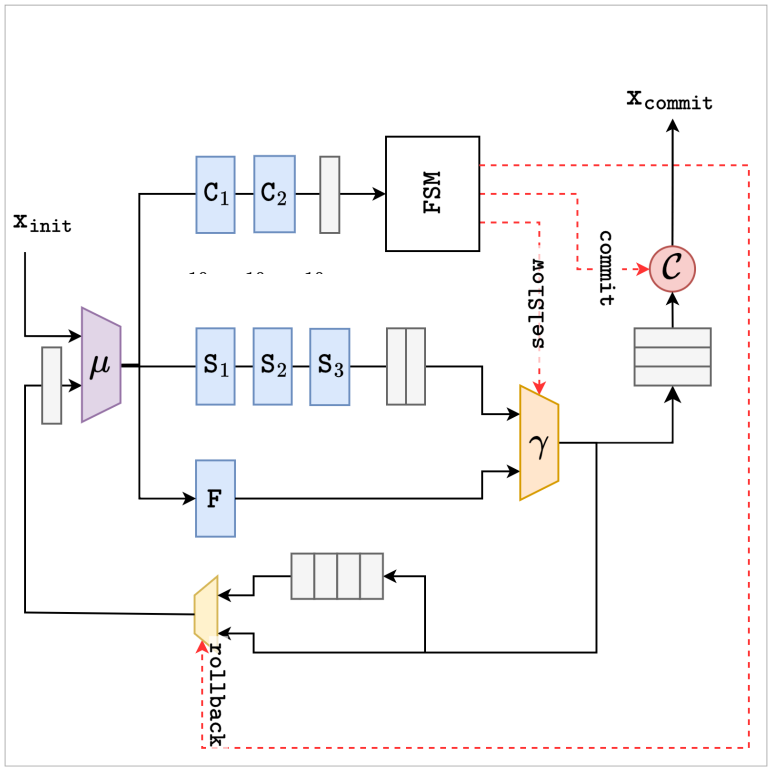
Speculative Loop Pipelining (SLP)

- Regenerate C code and let HLS balance the pipeline stage

Input code

```
do {
  if(C(x, z))
    x = S(x);
  else
    x = F(x);
} while(!x);
```

❶



Steven Derrien, Thibaut Marty, Simon Rokicki, Tomofumi Yuki. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2020

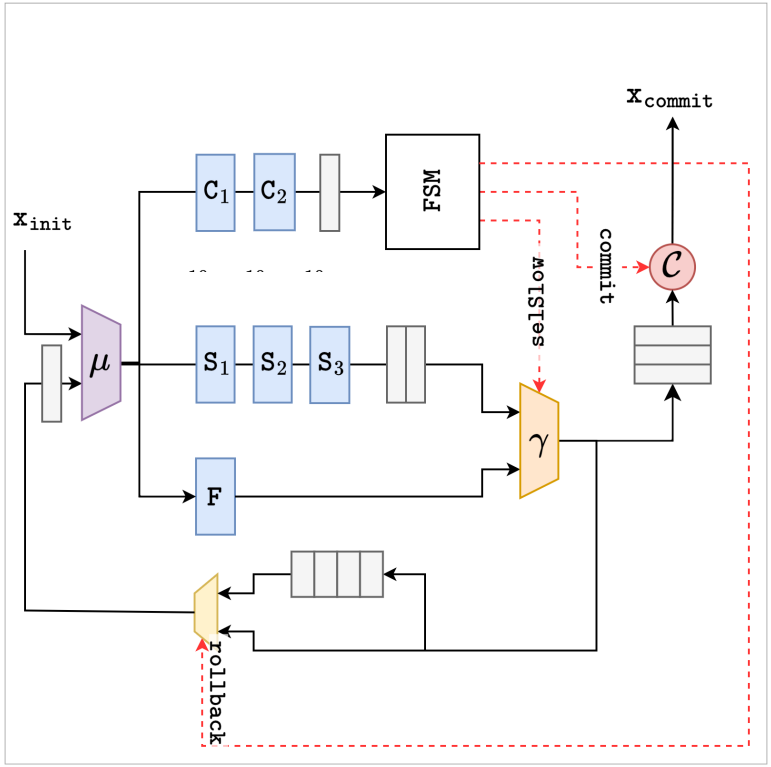
Speculative Loop Pipelining (SLP)

- Regenerate (cycle accurate) C code

Input code

```
do {
  if(C(x, z))
    x = S(x);
  else
    x = F(x);
} while(!x);
```

❶




```
#pragma hls distance mis_x=5
#pragma hls distance ctrl=3
do {
#pragma hls pipeline II=1
  ctrl[t] = C(s_x[t-1], z[t-1]);
  mis_x[t] = S(s_x[t-1]);
  cs = nextstate(cs, ctrl[t-3]);
  s_x[t] = cs.selSlow ?
             mis_x[t-5] :
             F(s_x[t-1]);
  if(cs.rollback) {
    s_x[t] = s_x[t-5];
    z[t] = z[t-5];
  }
  if(cs.commit)
    x = s_x[t-4];
  t += 1;
} while(!(x && cs.commit));
```

❸

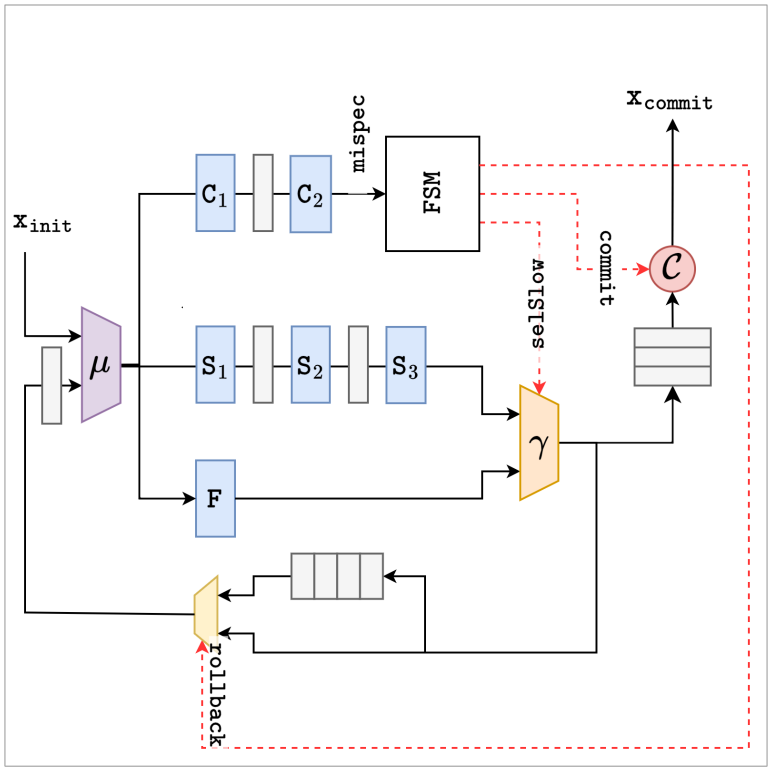
Steven Derrien, Thibaut Marty, Simon Rokicki, Tomofumi Yuki. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2020

Speculative Loop Pipelining (SLP)

- Let HLS balance pipeline stages

Input code

```
do {
  if(C(x, z))
    x = S(x);
  else
    x = F(x);
} while(!x);
```
❶



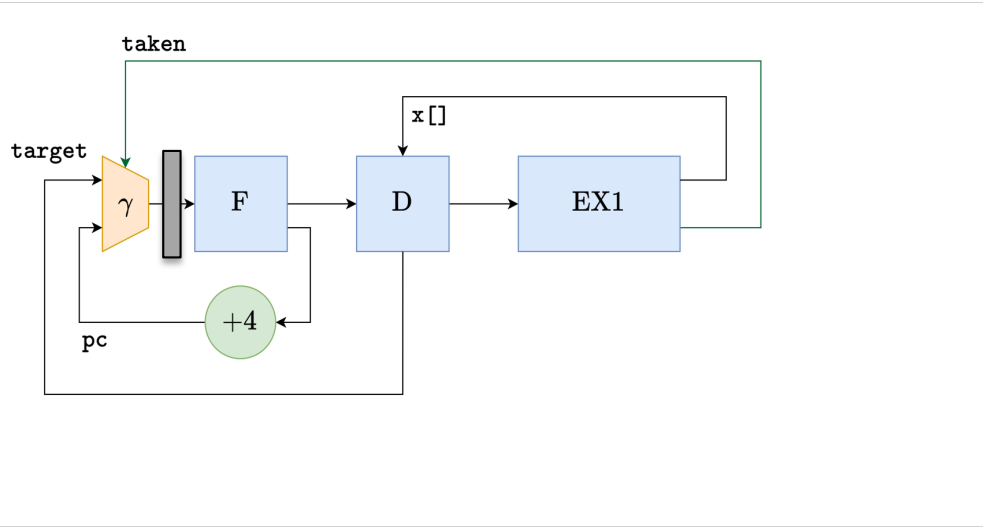

```
#pragma hls distance mis_x=5
#pragma hls distance ctrl=3
do {
#pragma hls pipeline II=1
  ctrl[t] = C(s_x[t-1], z[t-1]);
  mis_x[t] = S(s_x[t-1]);
  cs = nextstate(cs, ctrl[t-3]);
  s_x[t] = cs.selSlow ?
             mis_x[t-5] :
             F(s_x[t-1]);
  if(cs.rollback) {
    s_x[t] = s_x[t-5];
    z[t] = z[t-5];
  }
  if(cs.commit)
    x = s_x[t-4];
  t += 1;
} while(!(x && cs.commit));
```
❸

Steven Derrien, Thibaut Marty, Simon Rokicki, Tomofumi Yuki. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2020

Synthesizing RISC-V cores from ISSs

- Instruction Set Simulator = behavioral description

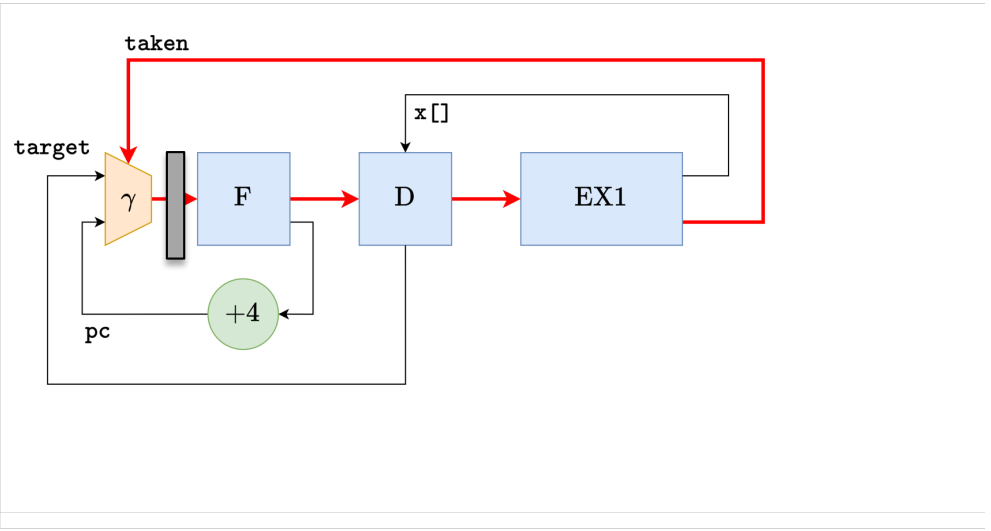
```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```



Synthesizing RISC-V cores from ISSs

- Instruction Set Simulator = behavioral description

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

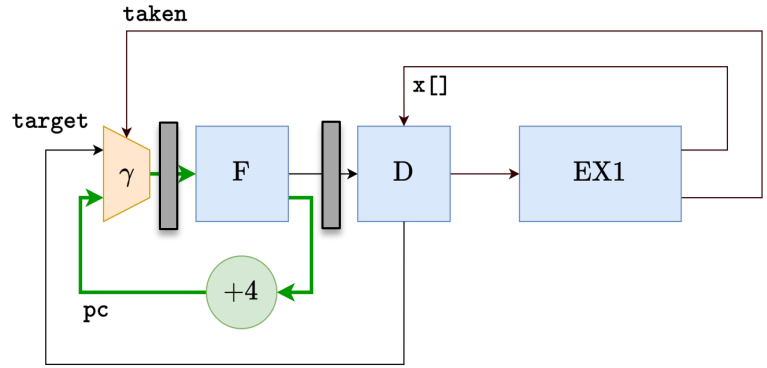


Synthesizing RISC-V cores from ISSs

- **Instruction Set Simulator = behavioral description**

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

We can speculate that current instruction ***is not*** a branch



We then have a 1 cycle penalty for every executed branch

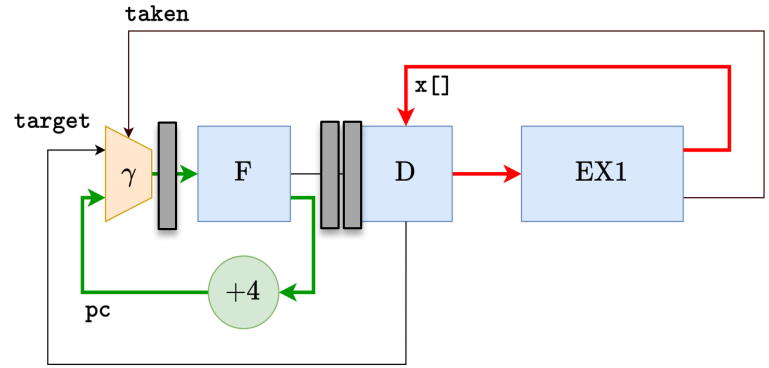
This is a 2 stage pipelined CPU

Synthesizing RISC-V cores from ISSs

- **Instruction Set Simulator = behavioral description**

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

Can we obtain a 3 stage pipeline by increasing the penalty ?



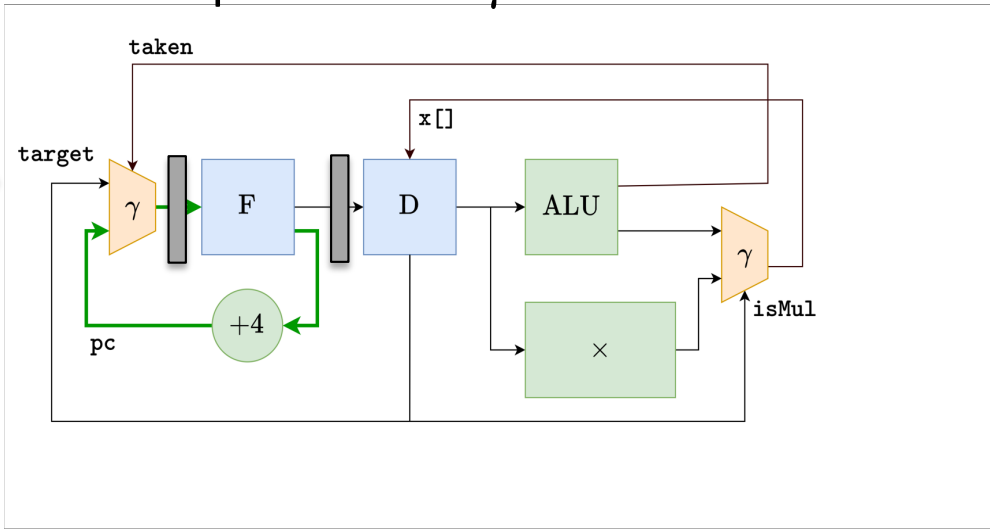
No, because of the dependency on register file **x[]**

Synthesizing RISC-V cores from ISSs

- **Instruction Set Simulator = behavioral description**

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

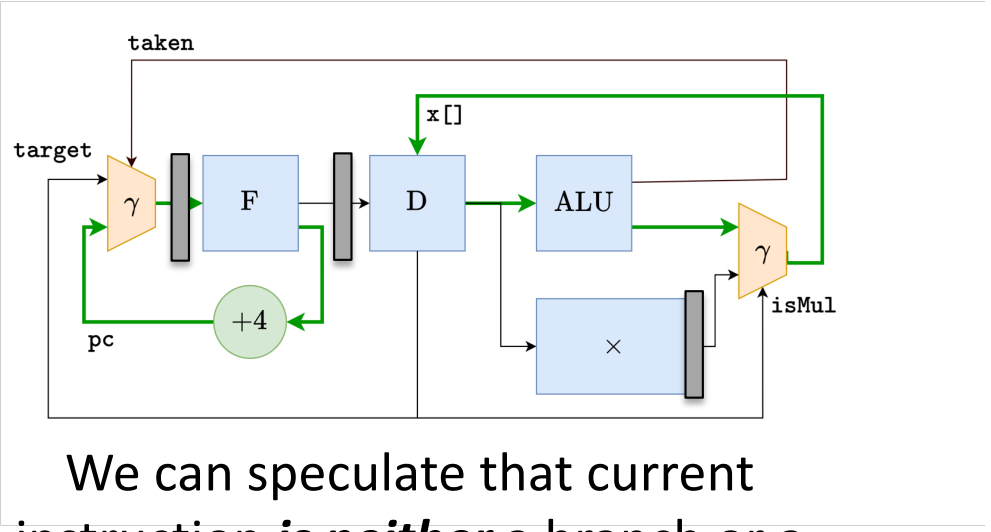
By opening the EX1 stage, we expose a new γ node …



Synthesizing RISC-V cores from ISSs

- **Instruction Set Simulator = behavioral description**

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

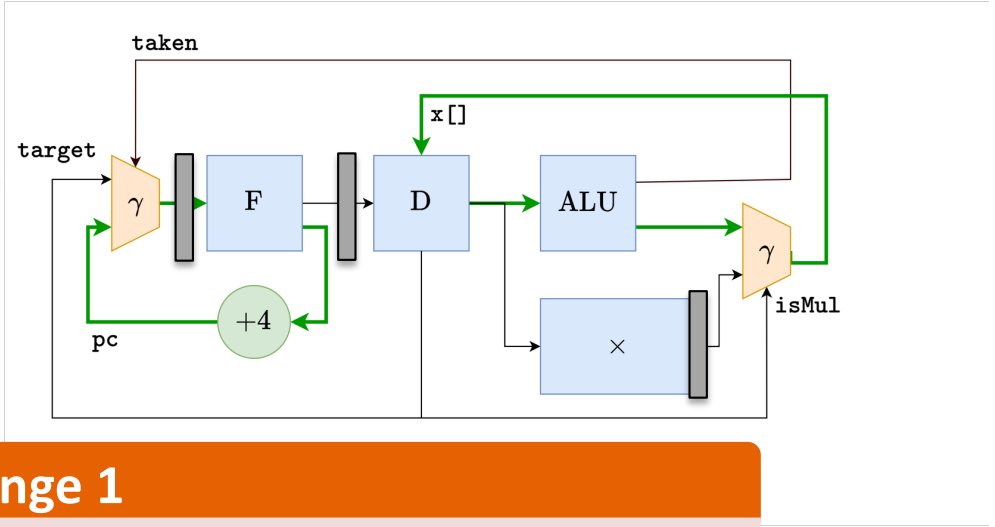


We can speculate that current instruction ***is neither*** a branch or a multiplication.

Synthesizing RISC-V cores from ISSs

- **Instruction Set Simulator = behavioral description**

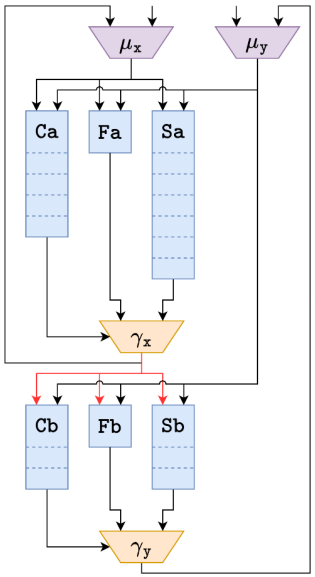
```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC =
}
```

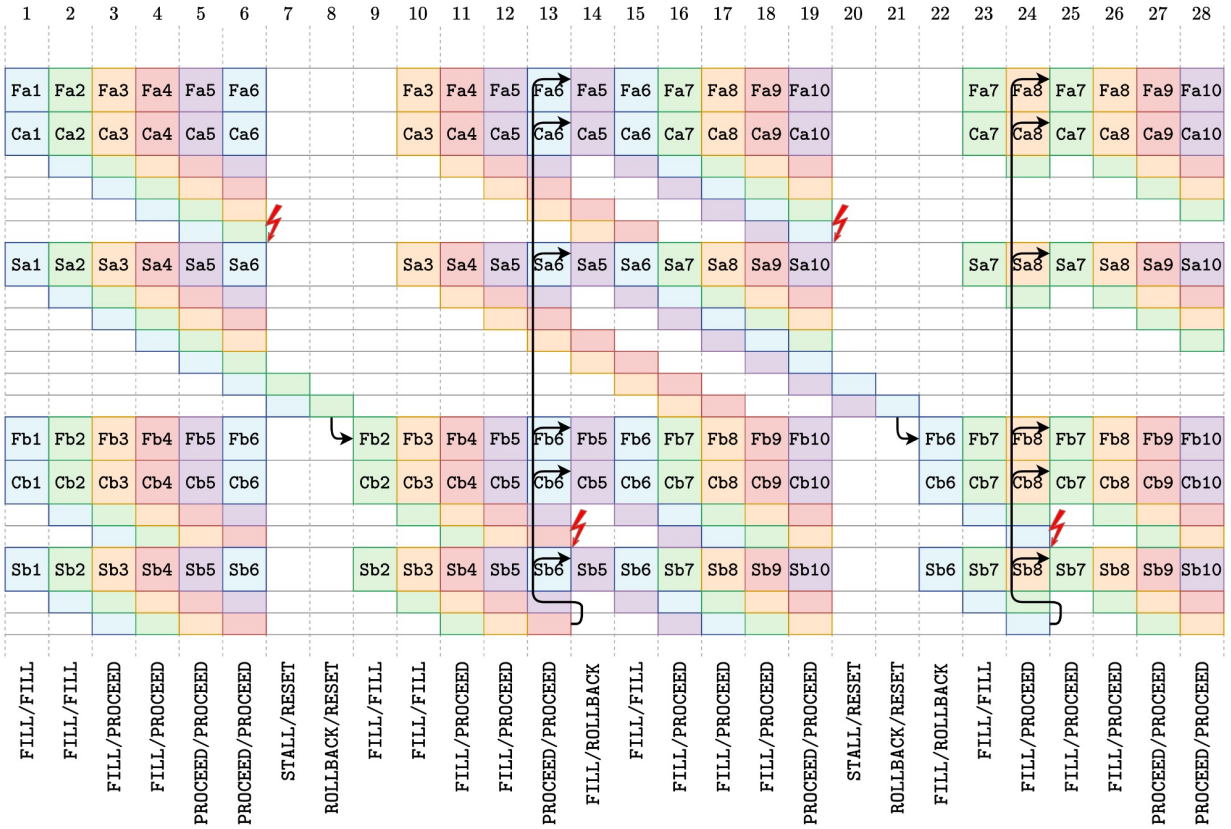


Challenge 1

But now, we have to speculate over multiple γ nodes …

Challenge 1 : multiple speculations





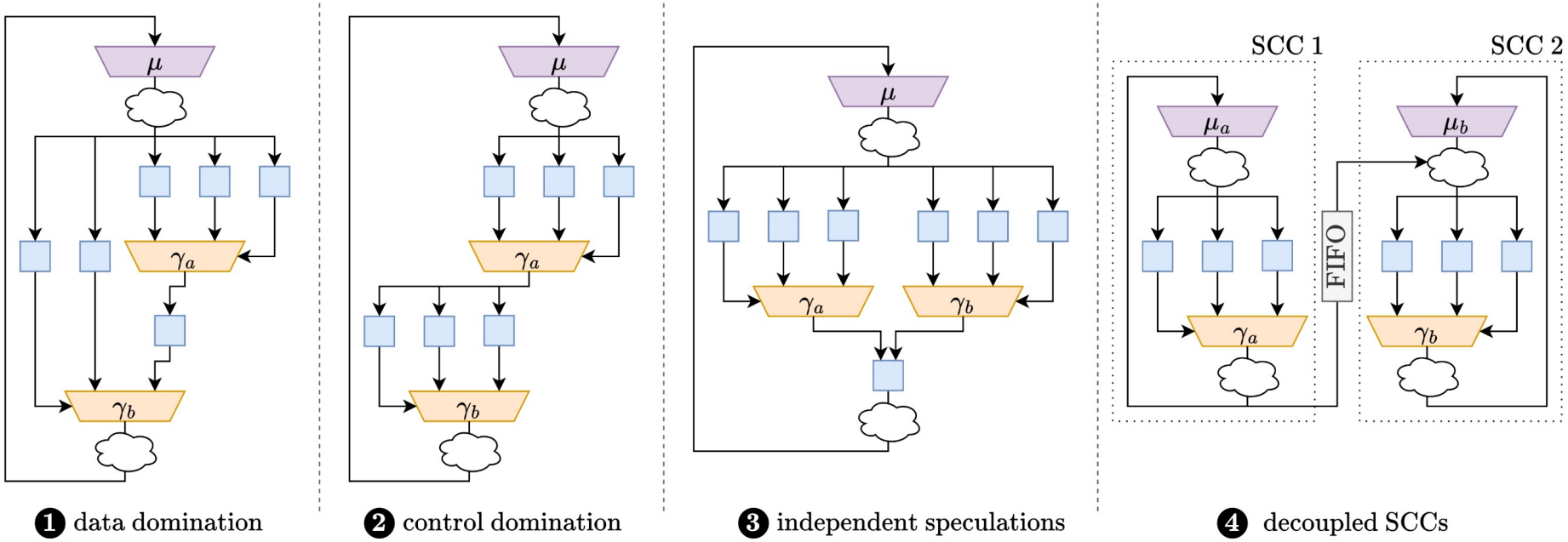
Interacting speculations

Speculative values of x influence the control decision on y

Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. “SpecHLS: Speculative Accelerator Design Using High-Level Synthesis”. In: IEEE Micro 42.5 (Sept. 2022), pp. 99–107. doi: 10.1109/MM.2022.3188136.

Identifying *speculation patterns*

We identify four distinct *speculation patterns*

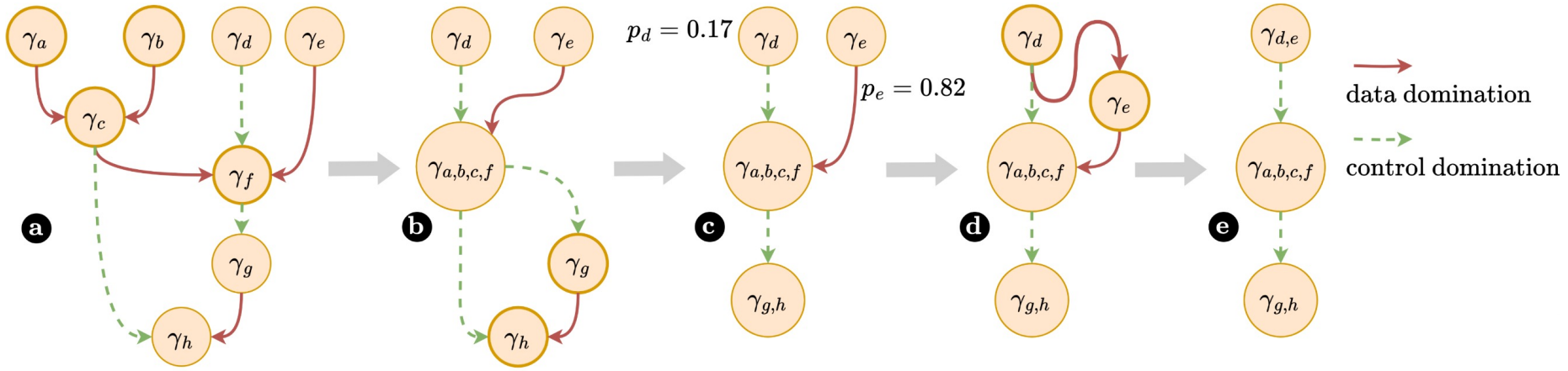


Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. “SpecHLS: Speculative Accelerator Design Using High-Level Synthesis”. In: IEEE Micro 42.5 (Sept. 2022), pp. 99–107. doi: 10.1109/MM.2022.3188136.

Iterative Pattern Resolution

Pattern resolution

- Iterative algorithm to synthesize the speculation control FSM
- Based on a set of **rewriting rules**, coupled to **profiling**

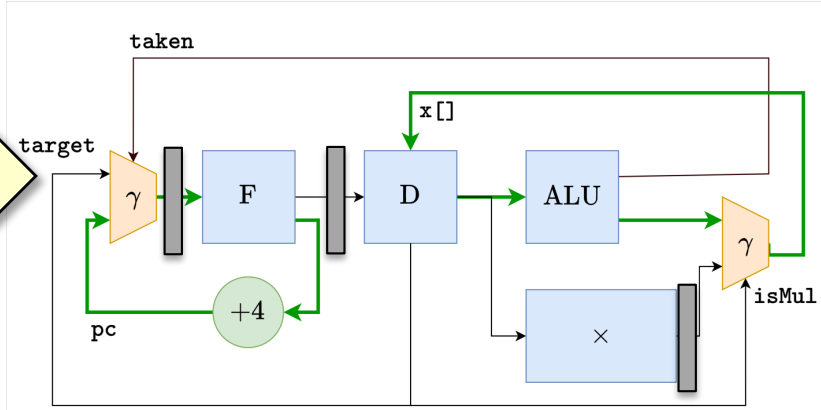


Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. "SpecHLS: Speculative Accelerator Design Using High-Level Synthesis". In: IEEE Micro 42.5 (Sept. 2022), pp. 99–107. doi: 10.1109/MM.2022.3188136.

Synthesizing RISC-V cores from ISSs

- Instruction Set Simulator = behavioral description

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```



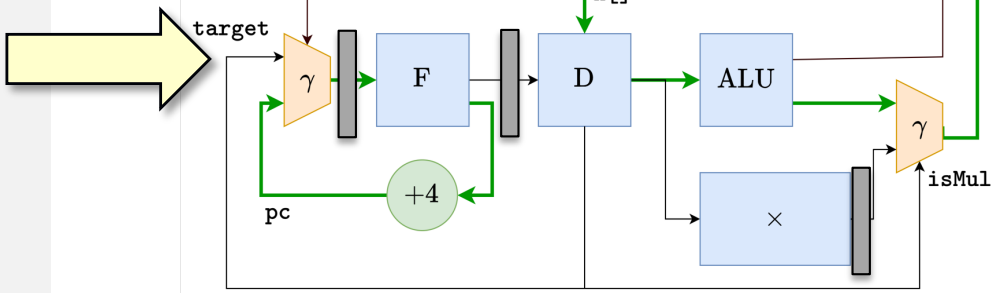

```
add x1,x2,x3
mul x6,x8,x9
```

This is too pessimistic, we can do better !

Synthesizing RISC-V cores from ISSs

- **Instruction Set Simulator = behavioral description**

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```



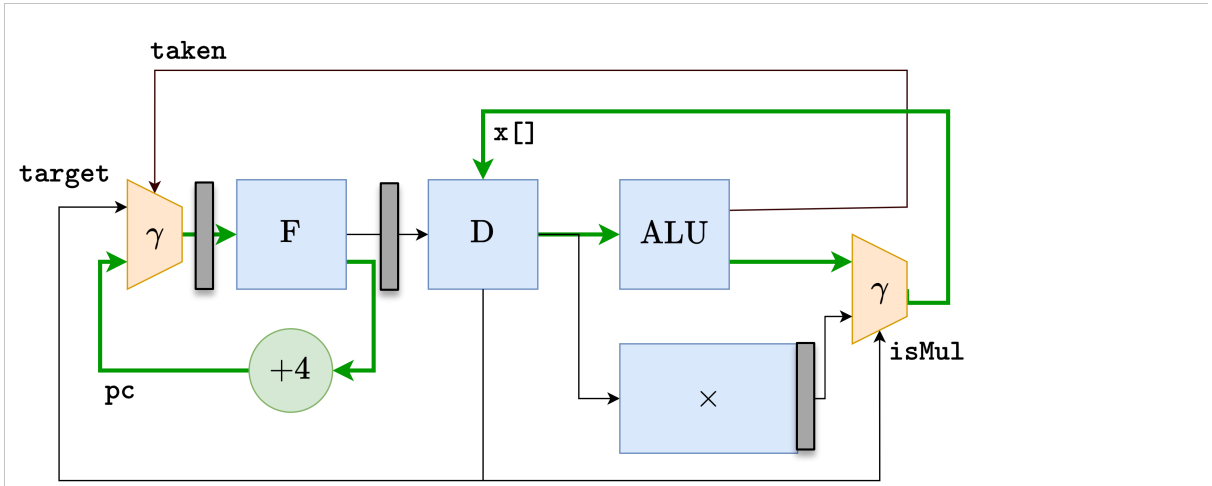
Challenge 2

How can we speculate on RAW dependencies ?

etter !

Challenge 2 : memory speculation

- **In our program IR Arrays are modeled as values**
  - We can check at run-time which version of the array value is needed

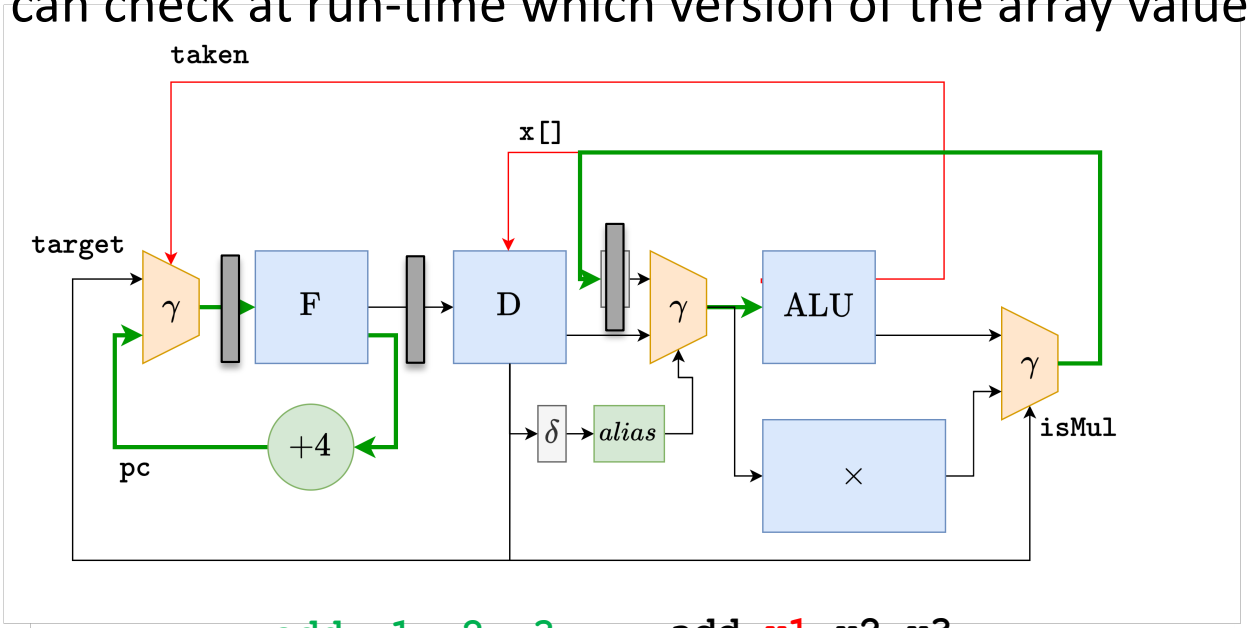


We can now speculate that there is no RAW dependency on x[] …

Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. “A Unified Memory Dependency Framework for Speculative High-Level Synthesis”. International Conference on Compiler Construction. CC 2024. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2024, pp. 13–25.

Challenge 2 : memory speculation

- **In our program IR Arrays are modeled as values**
  - We can check at run-time which version of the array value is needed



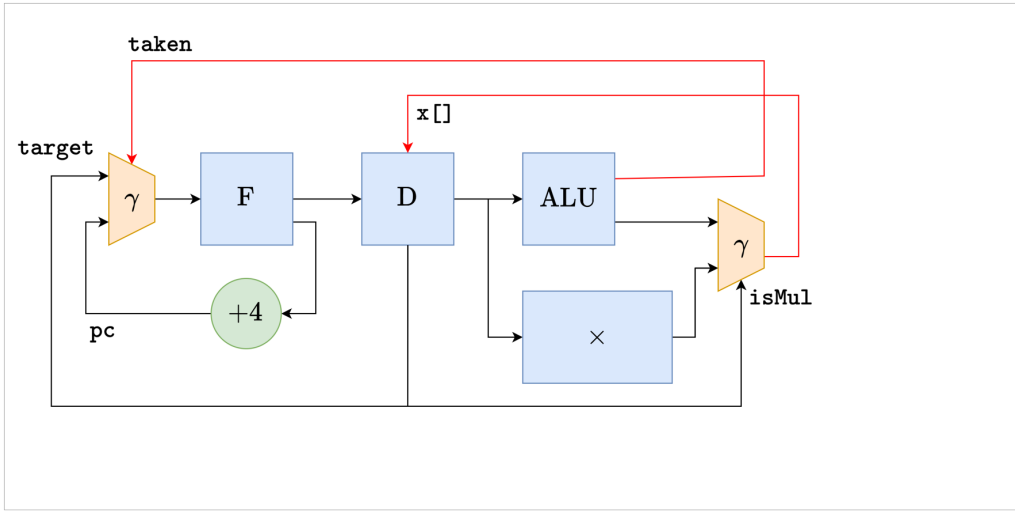
We can now speculate that there is no RAW dependency on x[] …

Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. “A Unified Memory Dependency Framework for Speculative High-Level Synthesis”. International Conference on Compiler Construction. CC 2024. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2024, pp. 13–25.

Synthesizing RISC-V cores from ISSs

- Instruction Set Simulator = behavioral description

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```



Synthesizing RISC-V cores from ISSs

- Instruction Set Simulator = behavioral description

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```

16 γ nodes =
~10^9 solutions to explore

Challenge 3

How do we figure out which γ nodes to speculate on ?

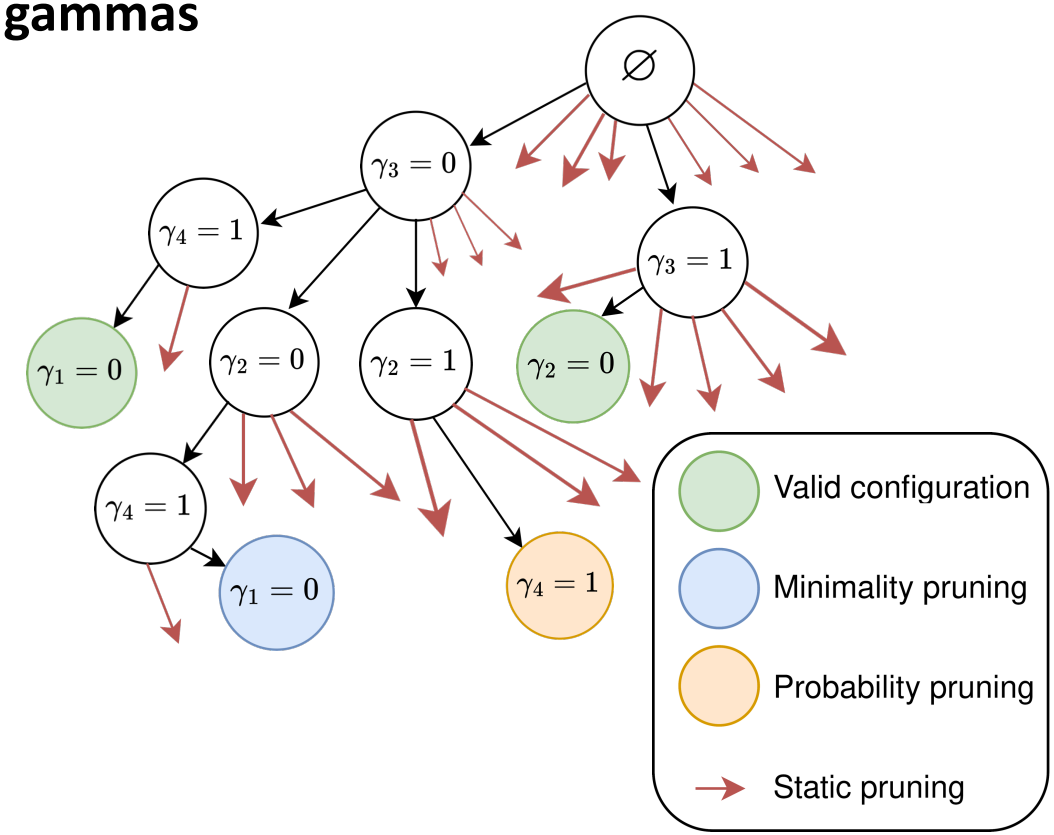
Challenge 3: which gamma to speculate ?

- **Solution : a branch and bound algorithm**
  - **Use profiling information for gammas**

Branch-and-bound algorithm find every valid configurations

Configuration sorted by increasing speculation profitability (gain + probability)

We can efficiently prune the exploration tree

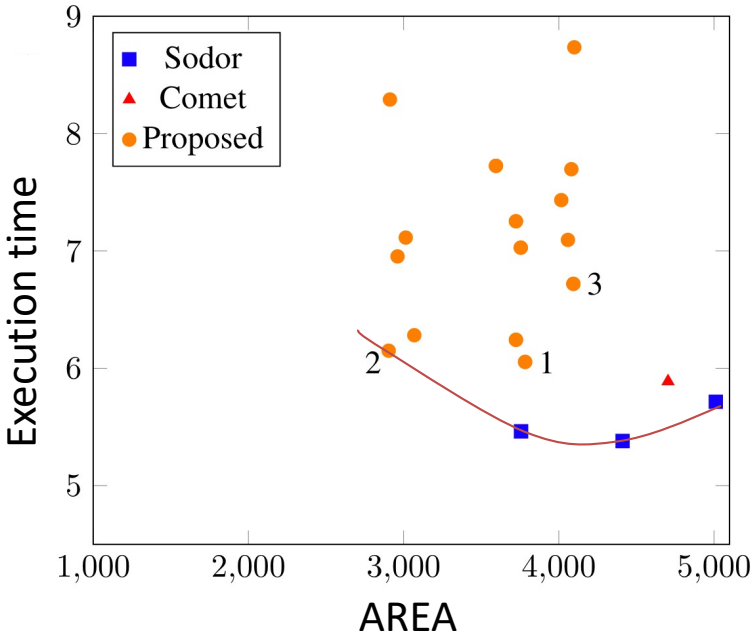


Leothaud, Dylan, et al. "Efficient Design Space Exploration for Dynamic & Speculative High-Level Synthesis." *Field-Programmable Logic and Applications (FPL)*. IEEE, 2024.

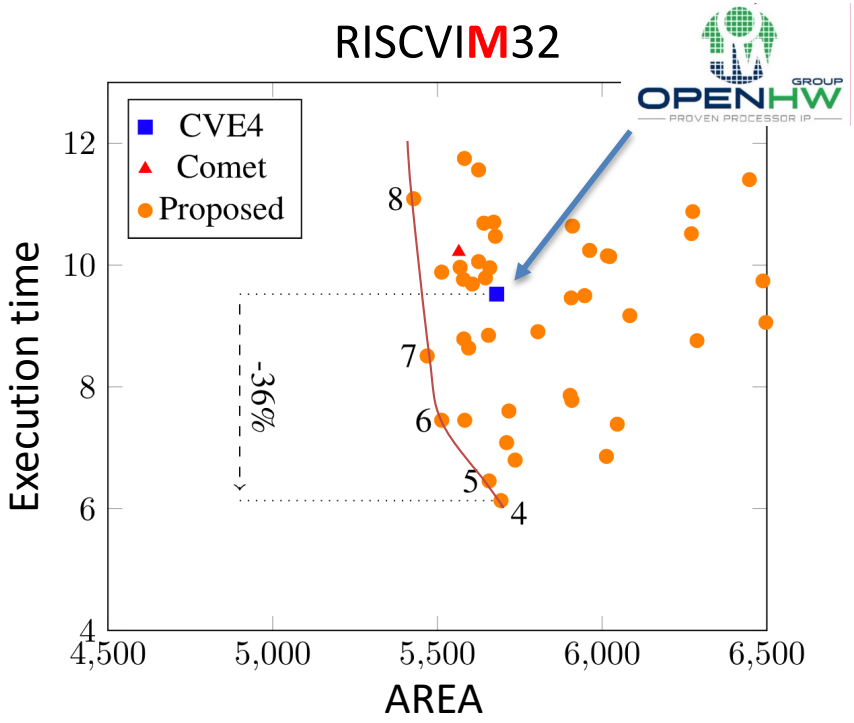
Early experimental results

- **Evaluation on two simple RISC-V ISAs (RVI32/RVIM32)**
  - Searching for Pareto Optimal solutions (performance/area trade-off)
  - Compared against manual designs (CVE4, Sodor)

RISCVI32



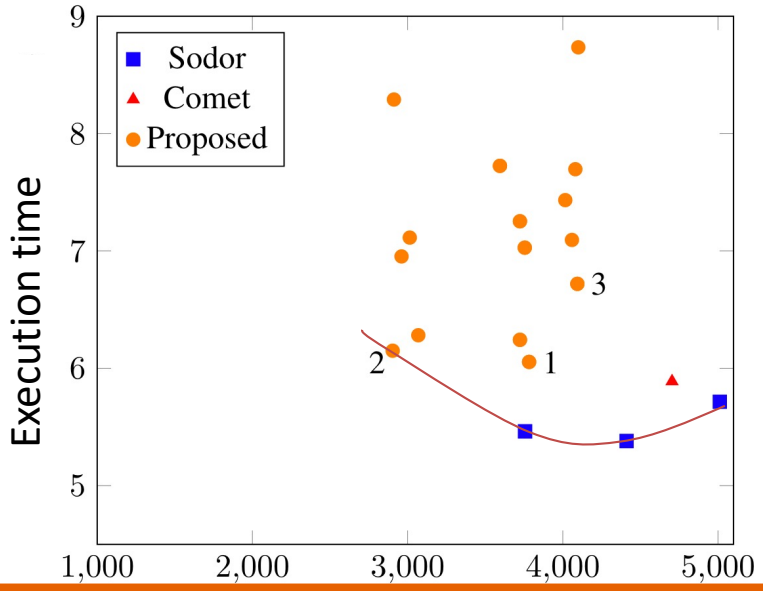
RISCVIM32



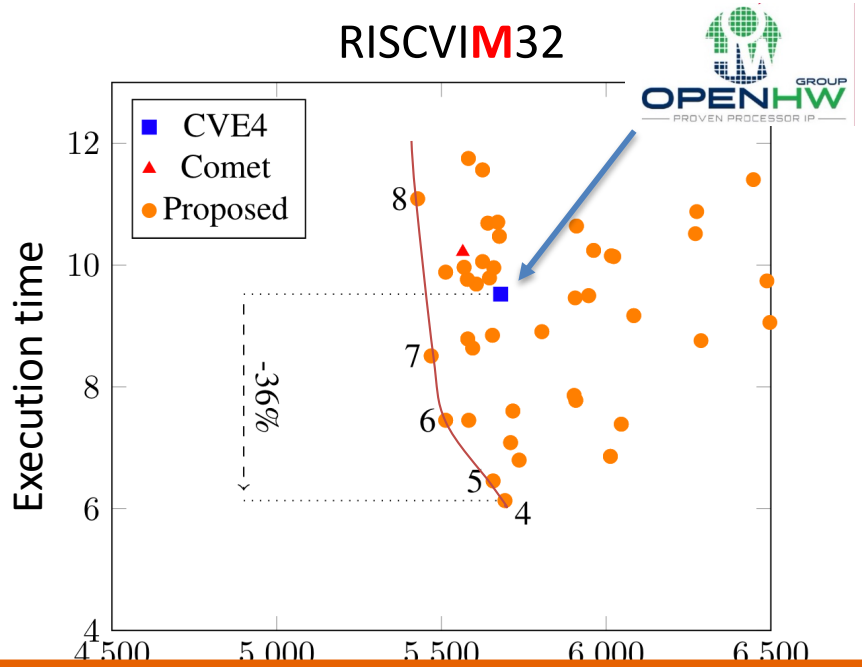
Early experimental results

- **Evaluation on two simple RISC-V ISAs (RVI32/RVIM32)**
  - Searching for Pareto Optimal solutions (performance/area trade-off)
  - Compared against manual designs (CVE4, Sodor)

RISCVI32



RISCVIM32



Takeaway

Ability to automatically derive competitive CPU designs

Future work

- **Pushing toward more complex microarchitcetures**
  - Dynamic branch prediction, cache, OoO
- **Formal verification of pipeline hazard logic**
  - Translation validation for SLP
- **Enforcing timing predictability for Real-time systems**
  - Automatic extraction of WCET pipeline models
  - Checking for timing-anomalies
- **Hardening against faults for security**
  - Partial duplication, error detectors/correctors
  - Automatic insertion of CFI/DFI mecanisms

Current & Future work

One flow to rule them all,
one flow to check them,
One flow to cover them all,
and to the silicon bind them

anr

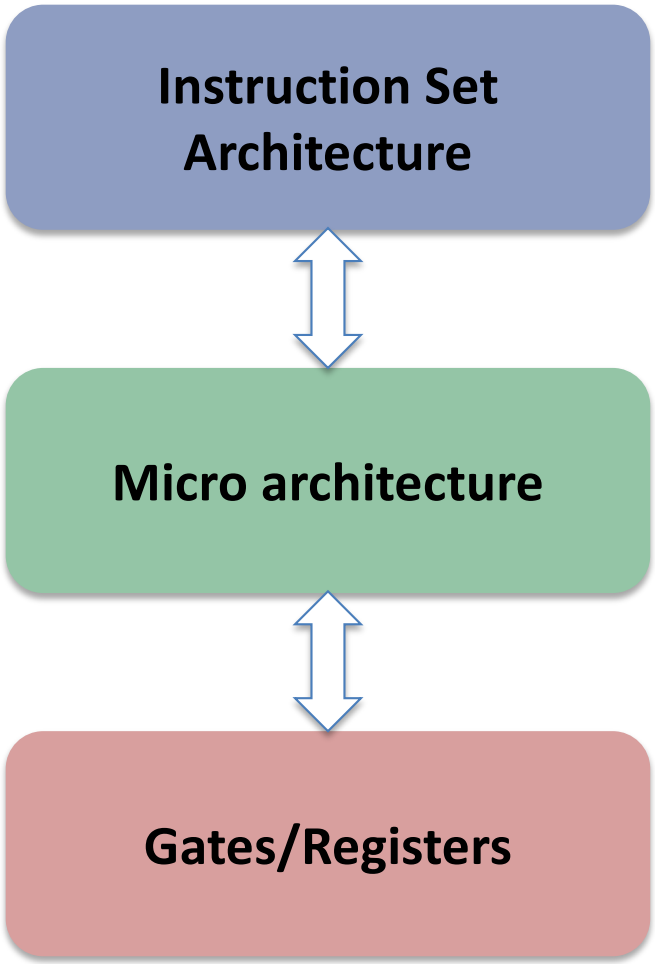
IRISA

cea

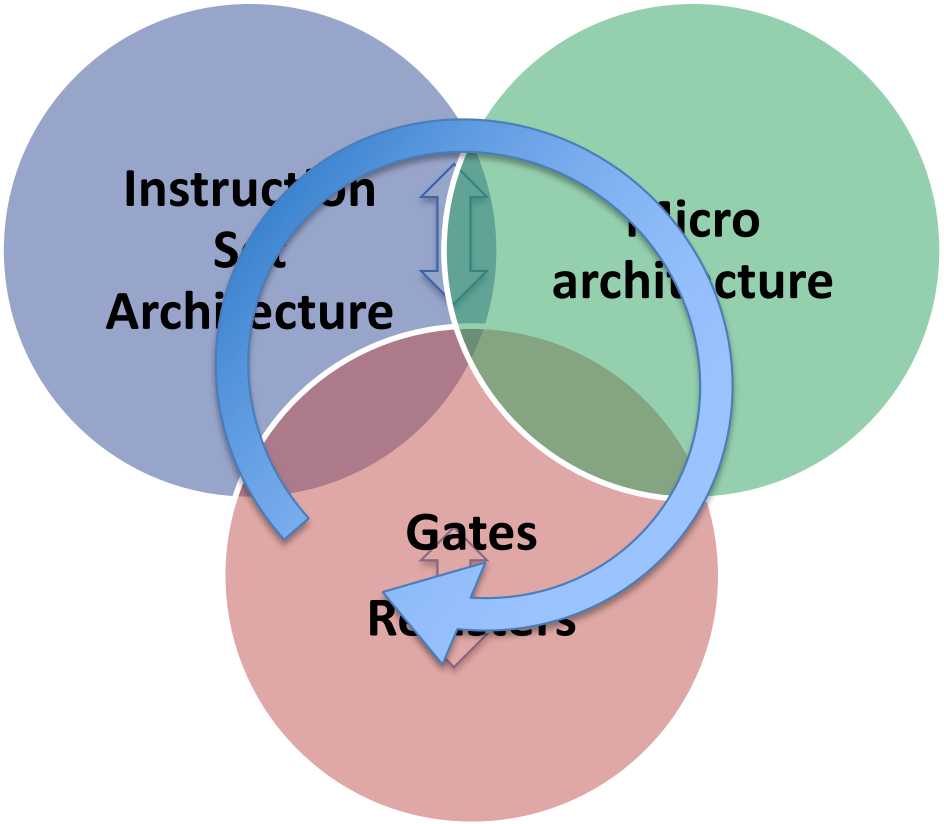
Lab-STICC

https://lotr.gitlabpages.inria.fr/website/

Take home message



Take home message

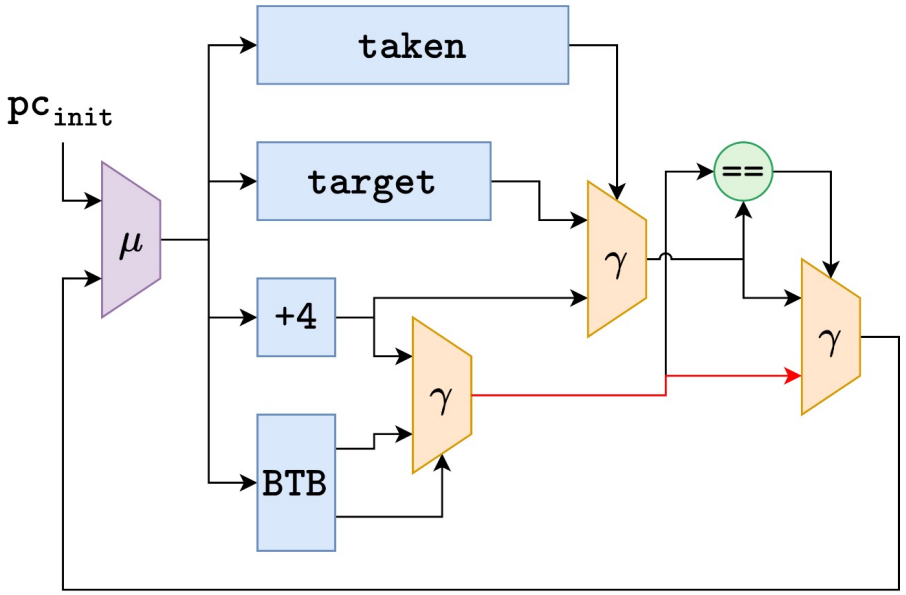


Modeling a Dynamic Branch Predictor

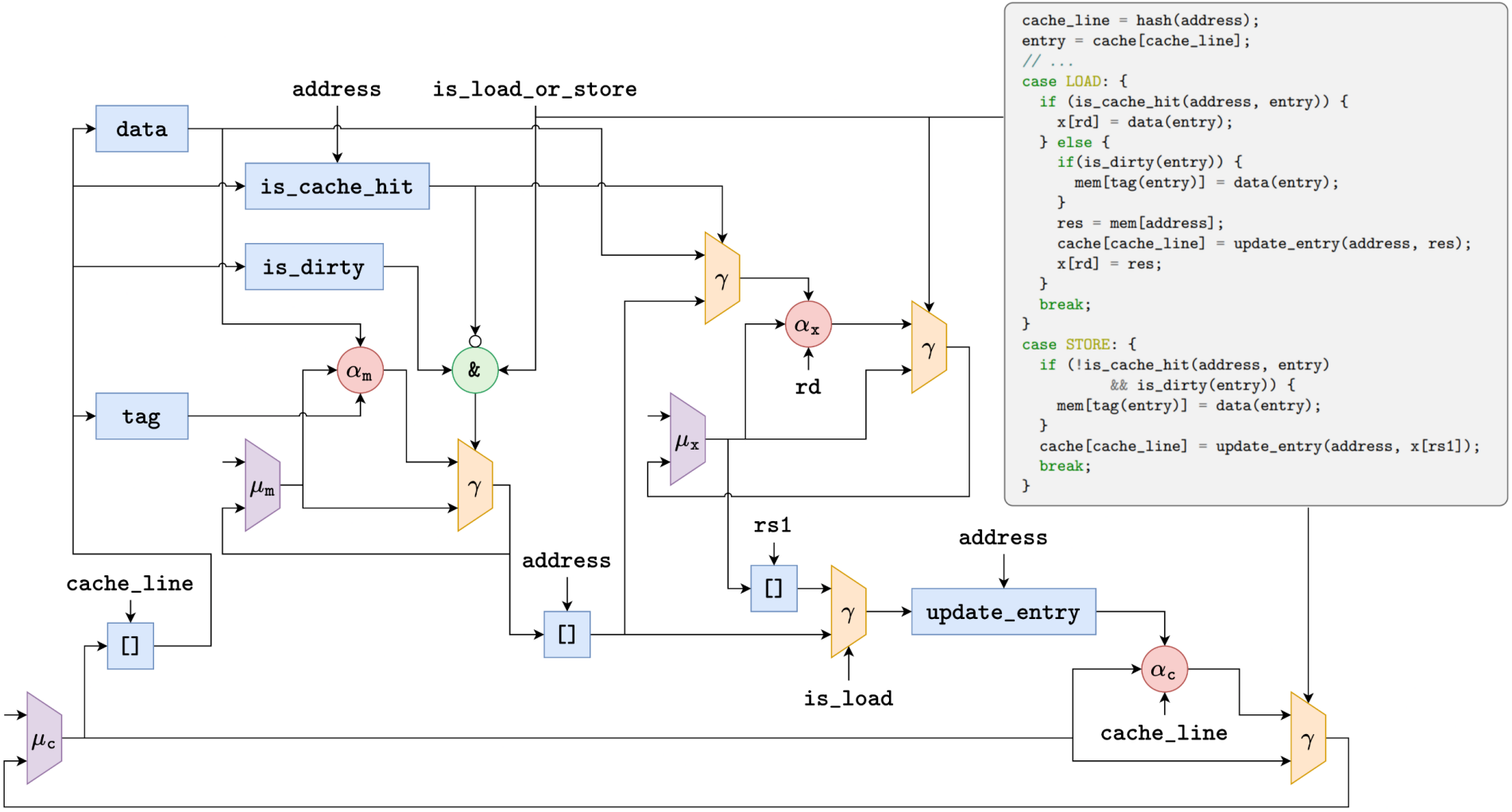
```
k = hash(pc);
pred_pc = pc + 4;
btb_hit = btb_tag[k] == pc;
// ...
if (btb_hit && btb_state[k] >= WEAK_TAKEN)
  pred_pc = btb_target[k];

switch (instr->opcode) { /* ... */ }

if (pred_pc == next_pc) {
  // Good prediction, fast path
  incr(btb_state[k]);
  pc = pred_pc;
} else {
  // Bad prediction, slow path
  if (btb_hit)
    decr(btb_state[k]);
  else if (/* The instruction was a branch */)
  ↪  {
    btb_state[k] = WEAK_TAKEN;
    btb_tag[k] = pc;
    btb_target[k] = /* Branch target */;
  }
  pc = next_pc;
}
```



Modeling a Data Cache




```
cache_line = hash(address);
entry = cache[cache_line];
// ...
case LOAD: {
  if (is_cache_hit(address, entry)) {
    x[rd] = data(entry);
  } else {
    if(is_dirty(entry)) {
      mem[tag(entry)] = data(entry);
    }
    res = mem[address];
    cache[cache_line] = update_entry(address, res);
    x[rd] = res;
  }
  break;
}
case STORE: {
  if (!is_cache_hit(address, entry)
          && is_dirty(entry)) {
    mem[tag(entry)] = data(entry);
  }
  cache[cache_line] = update_entry(address, x[rs1]);
  break;
}
```