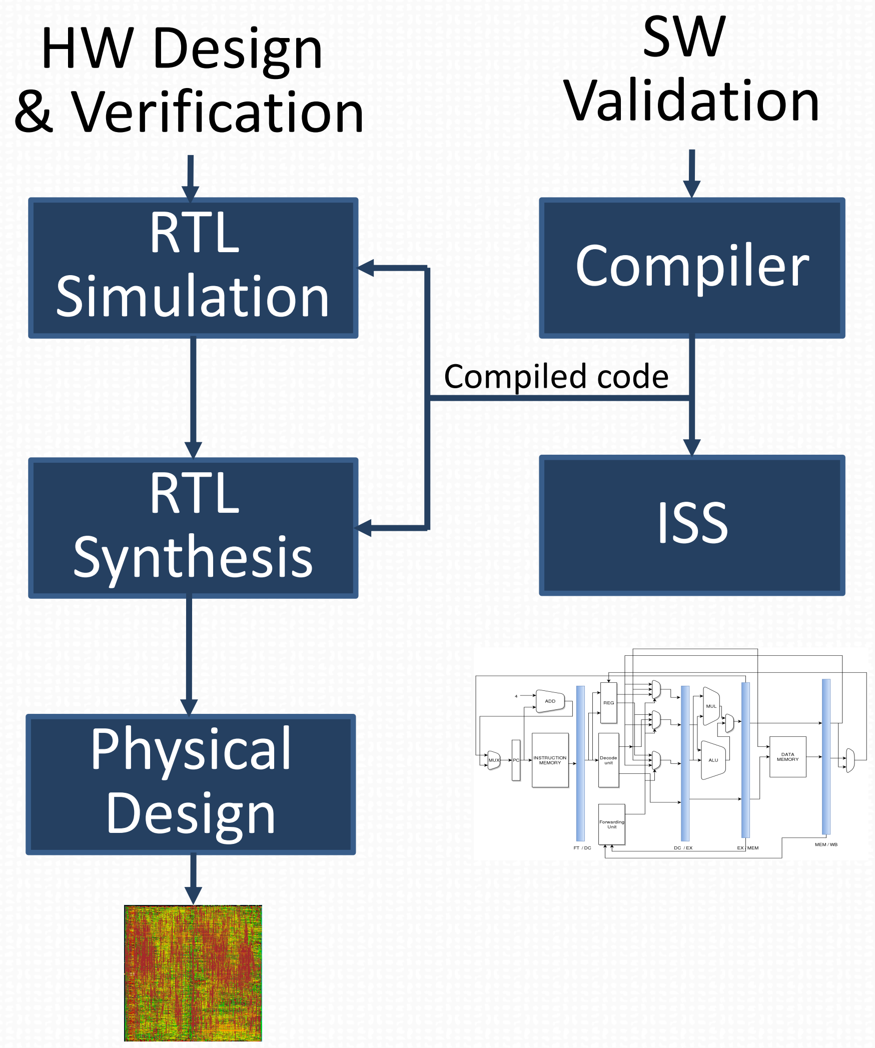
What You Simulate is What You Synthesize: Designing a Processor Core from C++ Specifications

Simon Rokicki, Joseph Paturel, Davide Pala, Olivier Sentieys

Univ. Rennes, Inria, IRISA

What You Simulate is What You Synthesize

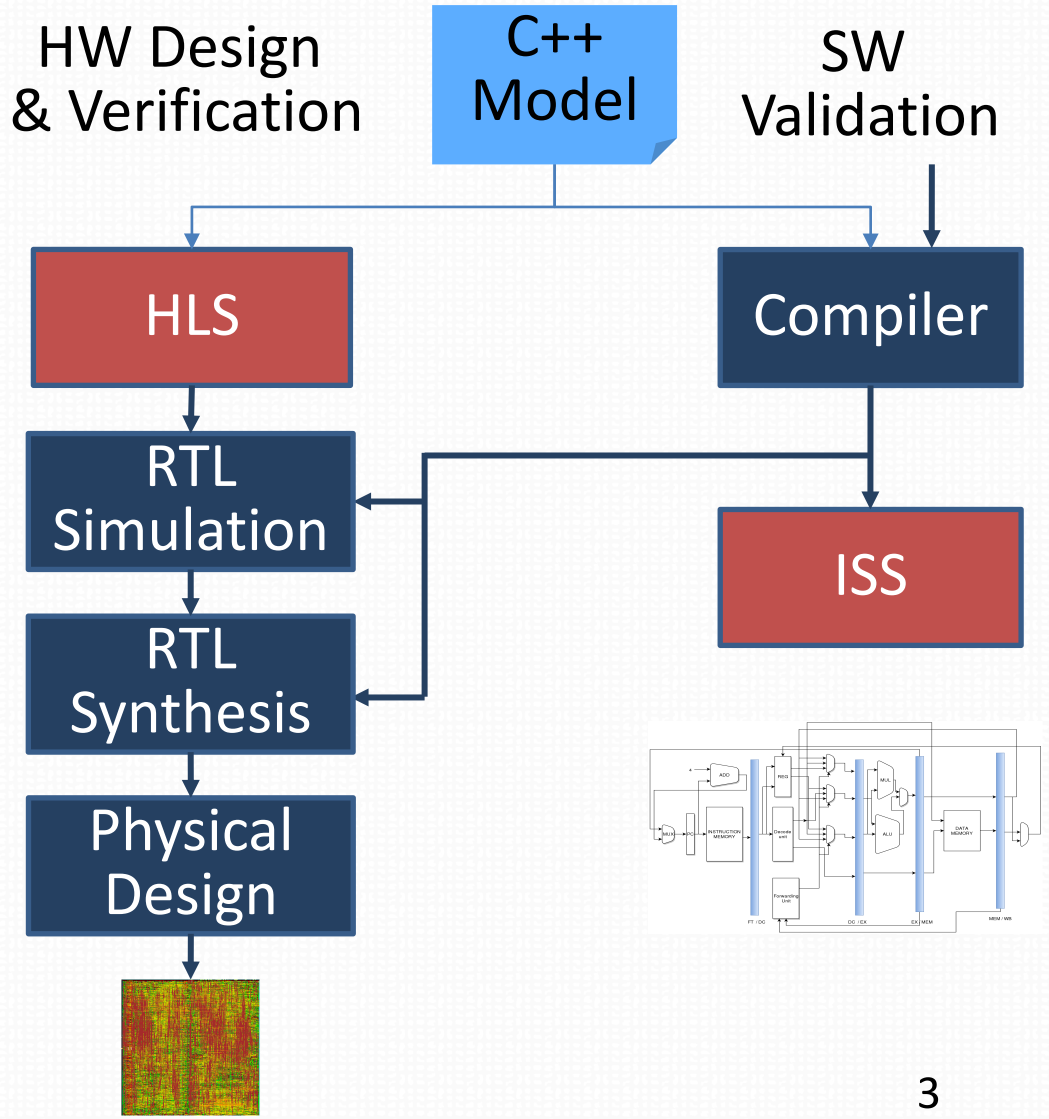
- Traditional Processor Design Flow
  - Maintain two coherent models:
    - RTL and simulation (ISS) models



What You Simulate is What You Synthesize

- Traditional Processor Design Flow
  - Maintain two coherent models:
    - RTL and simulation (ISS) models

- Proposed Flow
  - Design the processor as well as its software validation flow from a single high-level model



Is HLS suitable for designing processors?

- The answer is yes but….

Advantages and Challenges

Advantages

- Improves readability, productivity, maintainability, and flexibility of the design
- Object-Oriented processor model can easily be modified, expanded and verified

Challenges

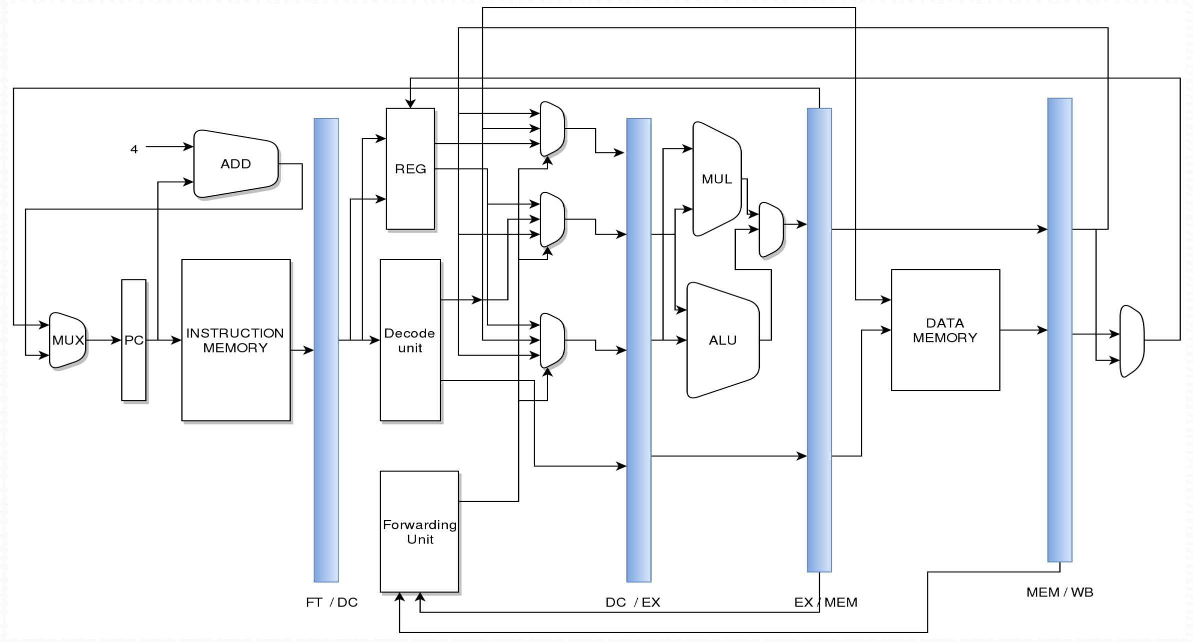
- How to specify core/un-core components, cache memory hierarchy, synchronization, etc.?
- How to specify parallel computing pipelines using HLS?

Comet

Is HLS suitable for processor design?

- This work describes Comet
  - 32-bit RISC-V instruction set
  - In-order 5-stage pipeline micro-architecture

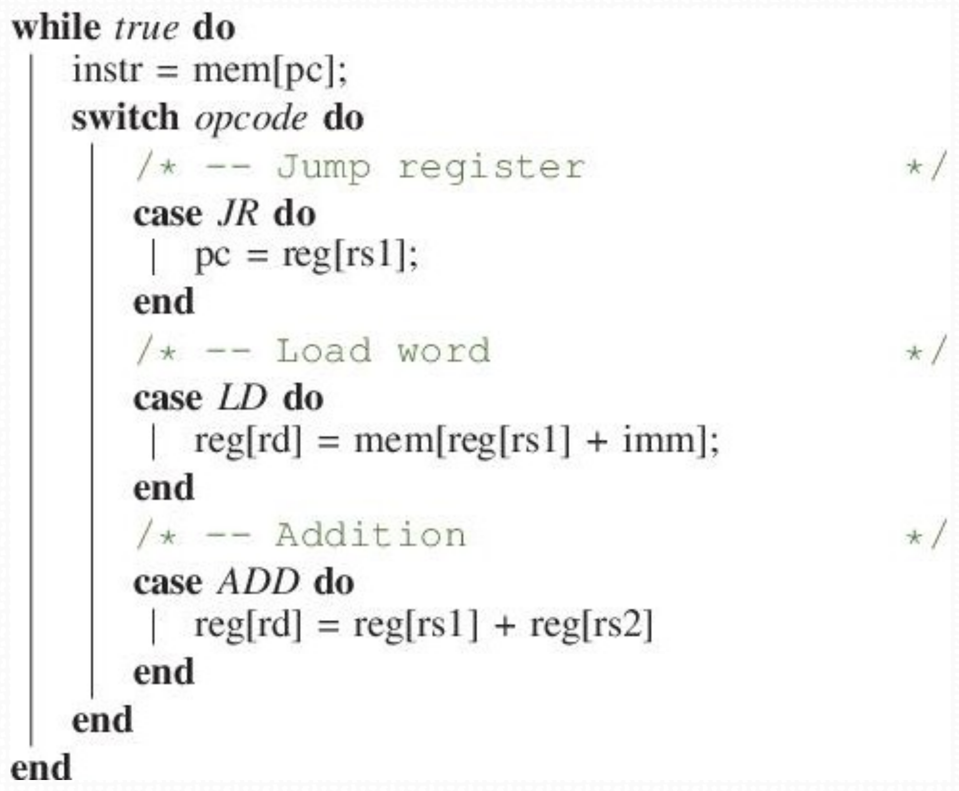
- Designed from a single C++ specification using High-Level Synthesis (HLS)



https://gitlab.inria.fr/srokicki/Comet

Synthesizing from an Instruction Set Simulator

- Main loop is pipelined
- Inter-iteration dependencies:
  - register file read/write dependencies
  - determining PC value
- Cycles per Instruction ≈ 3

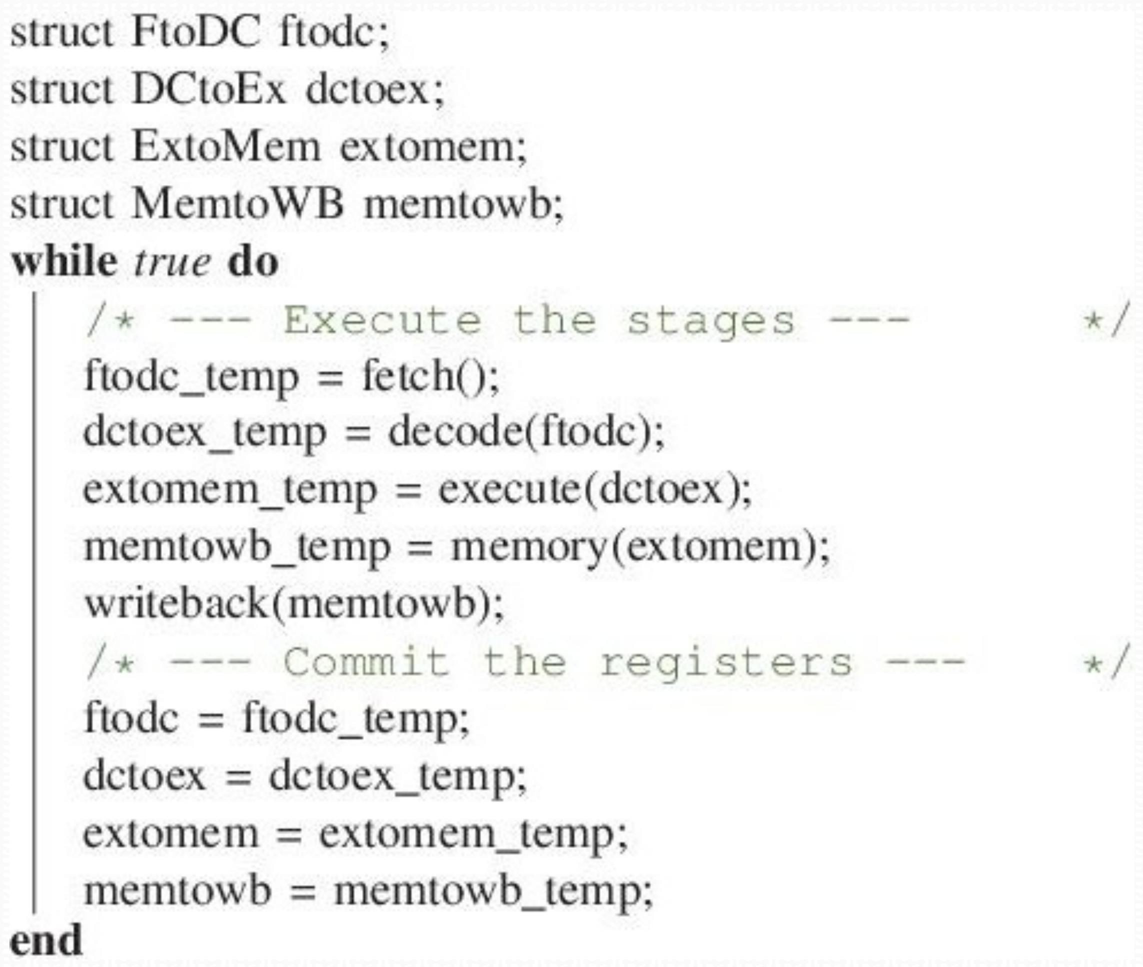


```
while true do
    instr = mem[pc];
    switch opcode do
        /* -- Jump register                    */
        case JR do
            pc = reg[rs1];
        end
        /* -- Load word                        */
        case LD do
            reg[rd] = mem[reg[rs1] + imm];
        end
        /* -- Addition                         */
        case ADD do
            reg[rd] = reg[rs1] + reg[rs2]
        end
    end
end
```

Need to explicit the pipeline and the stall/forwarding logic!

Explicitly Pipelined Simulator (1/2)

- Pipelined stages are explicit
- Pipeline registers are variables
- One iteration is the execution of each stage
- Main loop is pipelined (II=1)

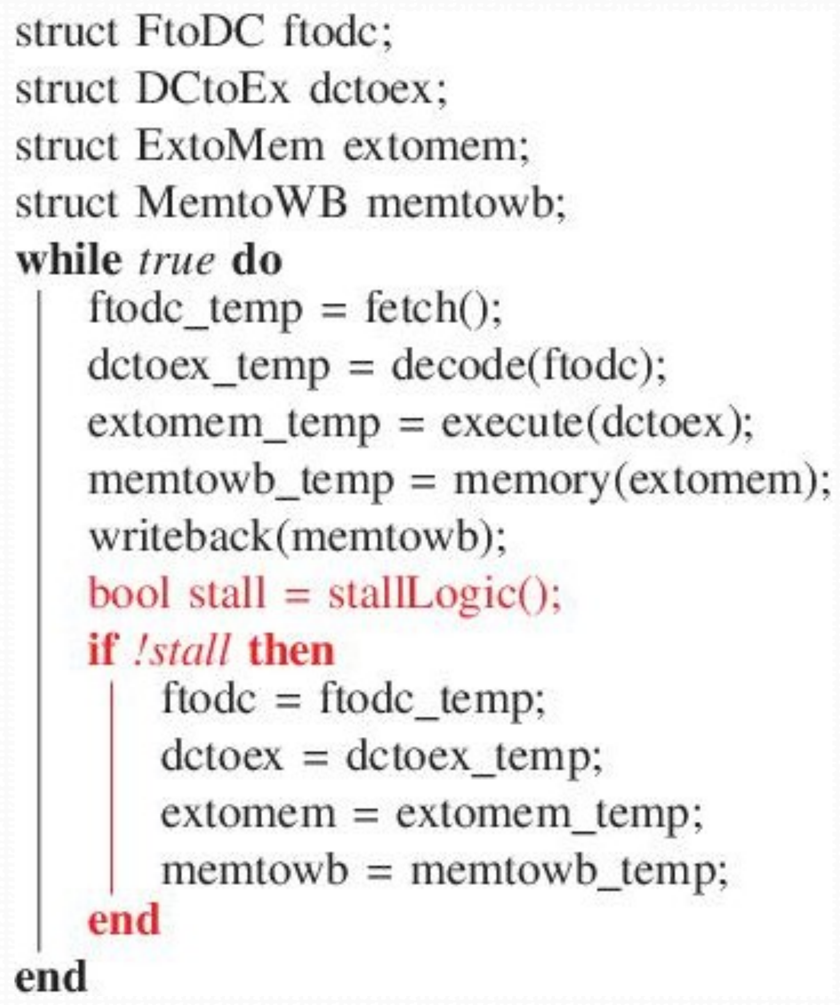


```
struct FtoDC ftodc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;
while true do
    /* --- Execute the stages ---          */
    ftodc_temp = fetch();
    dctoex_temp = decode(ftodc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    /* --- Commit the registers ---        */
    ftodc = ftodc_temp;
    dctoex = dctoex_temp;
    extomem = extomem_temp;
    memtowb = memtowb_temp;
end
```

Explicitly Pipelined Simulator (1/2)

- Pipelined stages are explicit
- Pipeline registers are variables
- One iteration is the execution of each stage
- Main loop is pipelined (II=1)

- Explicit stall mechanism

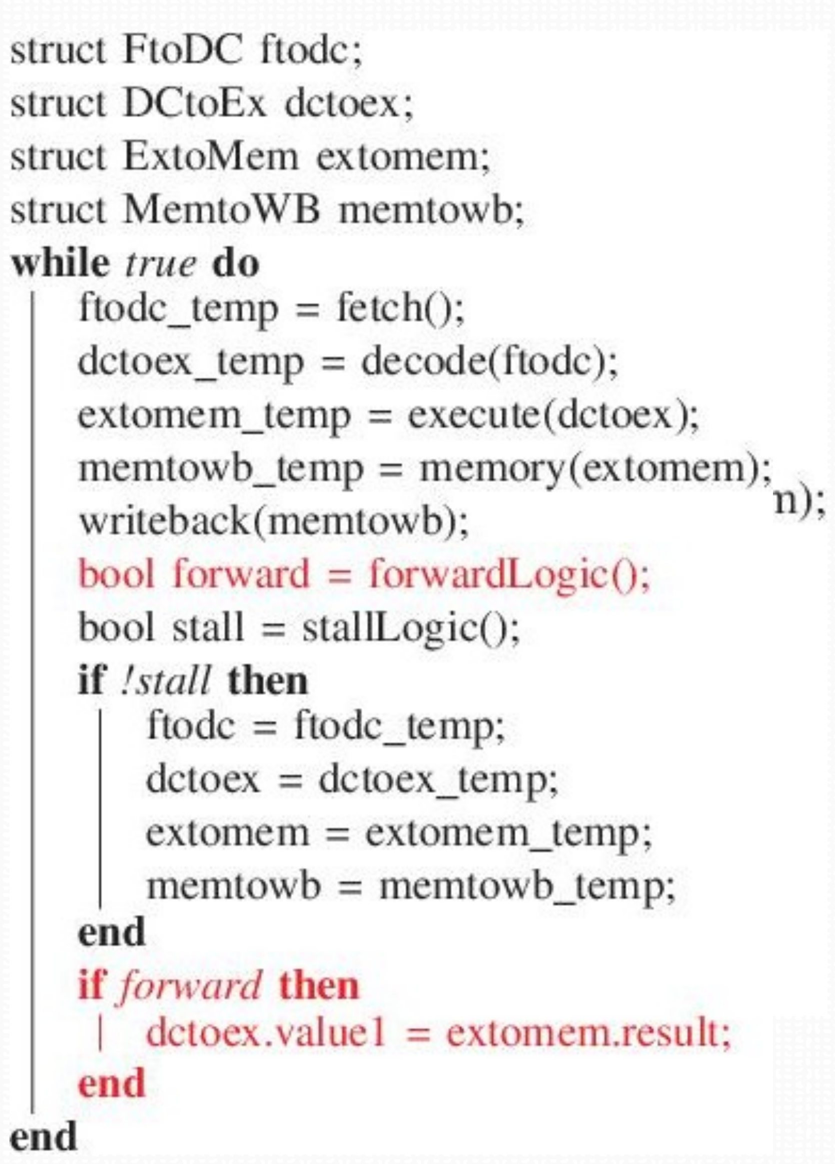


```
struct FtoDC ftodc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;
while true do
    ftodc_temp = fetch();
    dctoex_temp = decode(ftodc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    bool stall = stallLogic();
    if !stall then
        ftodc = ftodc_temp;
        dctoex = dctoex_temp;
        extomem = extomem_temp;
        memtowb = memtowb_temp;
    end
end
```

Explicitly Pipelined Simulator (1/2)

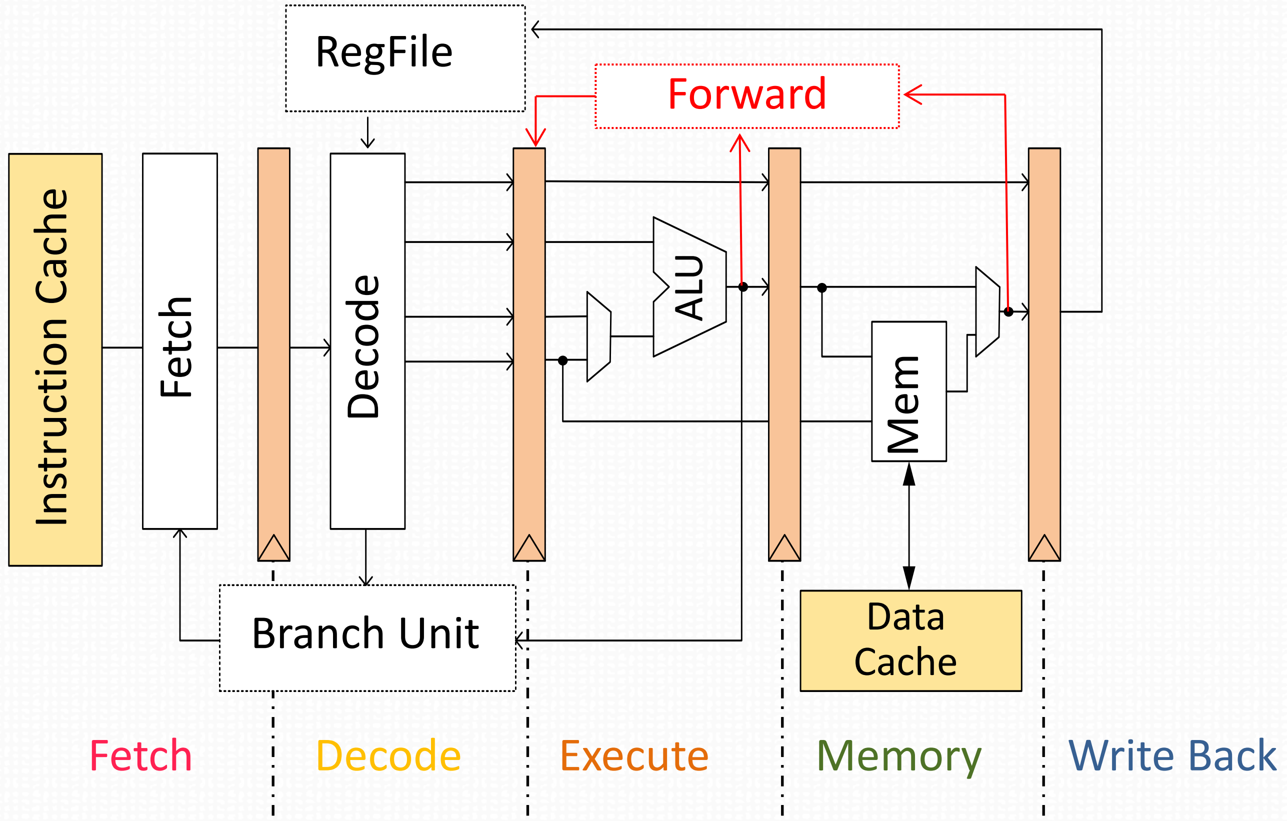
- Pipelined stages are explicit
- Pipeline registers are variables
- One iteration is the execution of each stage
- Main loop is pipelined (II=1)

- Explicit stall mechanism
- Forwarding



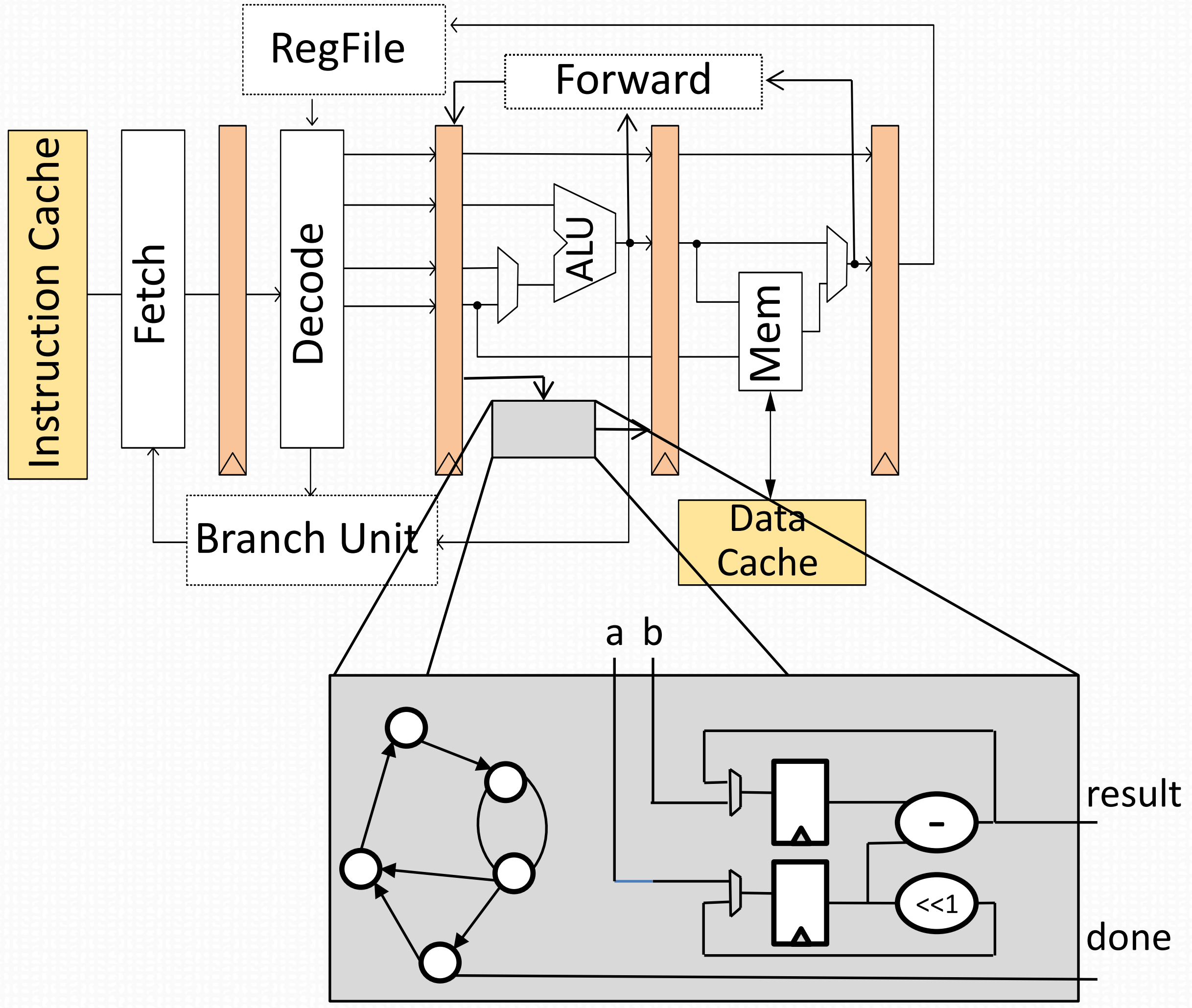
```
struct FtoDC ftodc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;
while true do
    ftodc_temp = fetch();
    dctoex_temp = decode(ftodc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);  n);
    writeback(memtowb);
    bool forward = forwardLogic();
    bool stall = stallLogic();
    if !stall then
        ftodc = ftodc_temp;
        dctoex = dctoex_temp;
        extomem = extomem_temp;
        memtowb = memtowb_temp;
    end
    if forward then
        dctoex.value1 = extomem.result;
    end
end
```

Explicitly Pipelined Simulator (2/2)



```
struct FtoDC ftodc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;
while true do
    ftodc_temp = fetch();
    dctoex_temp = decode(ftodc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    bool forward = forwardLogic();
    bool stall = stallLogic();
    if !stall then
        ftodc = ftodc_temp;
        dctoex = dctoex_temp;
        extomem = extomem_temp;
        memtowb = memtowb_temp;
    end
    if forward then
        dctoex.value1 = extomem.result;
    end
end
```

Dealing with Multi-Cycle Operators

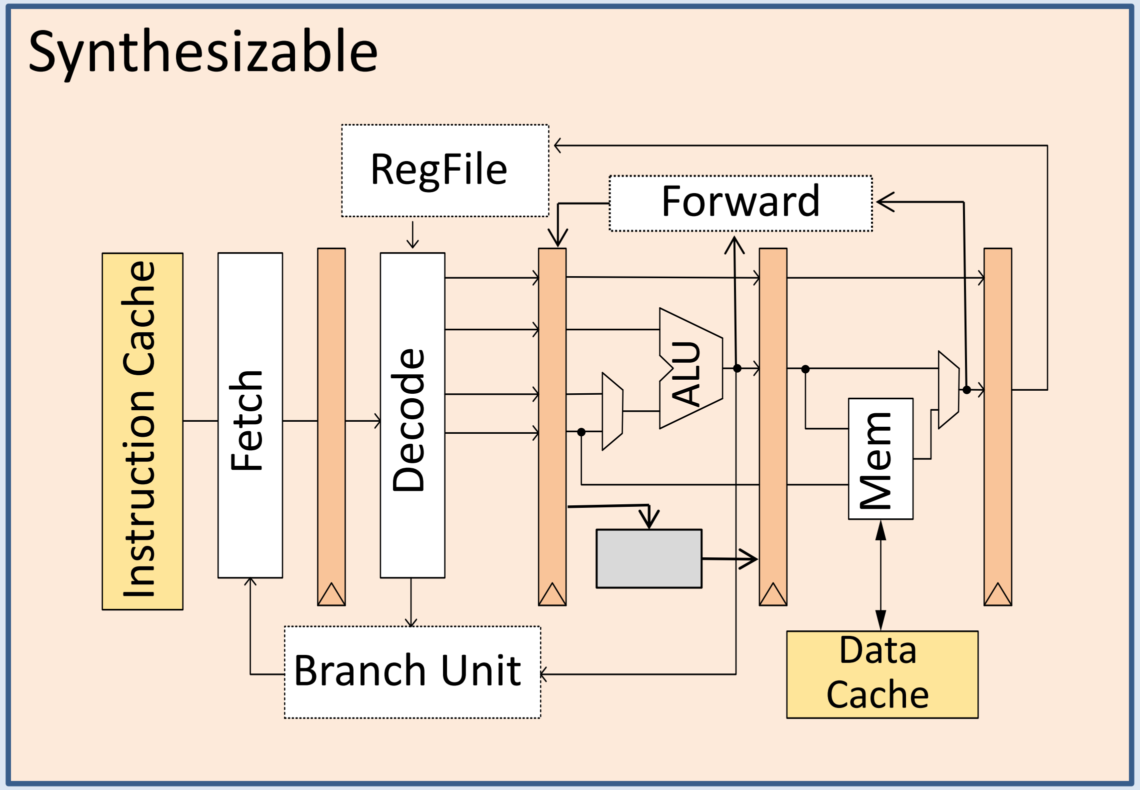


- Multi-cycle operators combine state machine & execution logic
- State machine **encoded** in the C code using a switch/case
- Used for:
  - Division
  - Caches
  - FPU

Comet Simulation Environment

Not Synthesizable

- Instrumentation
- Syscall emulation
- Elf reader
- Not-yet-synthesized extensions



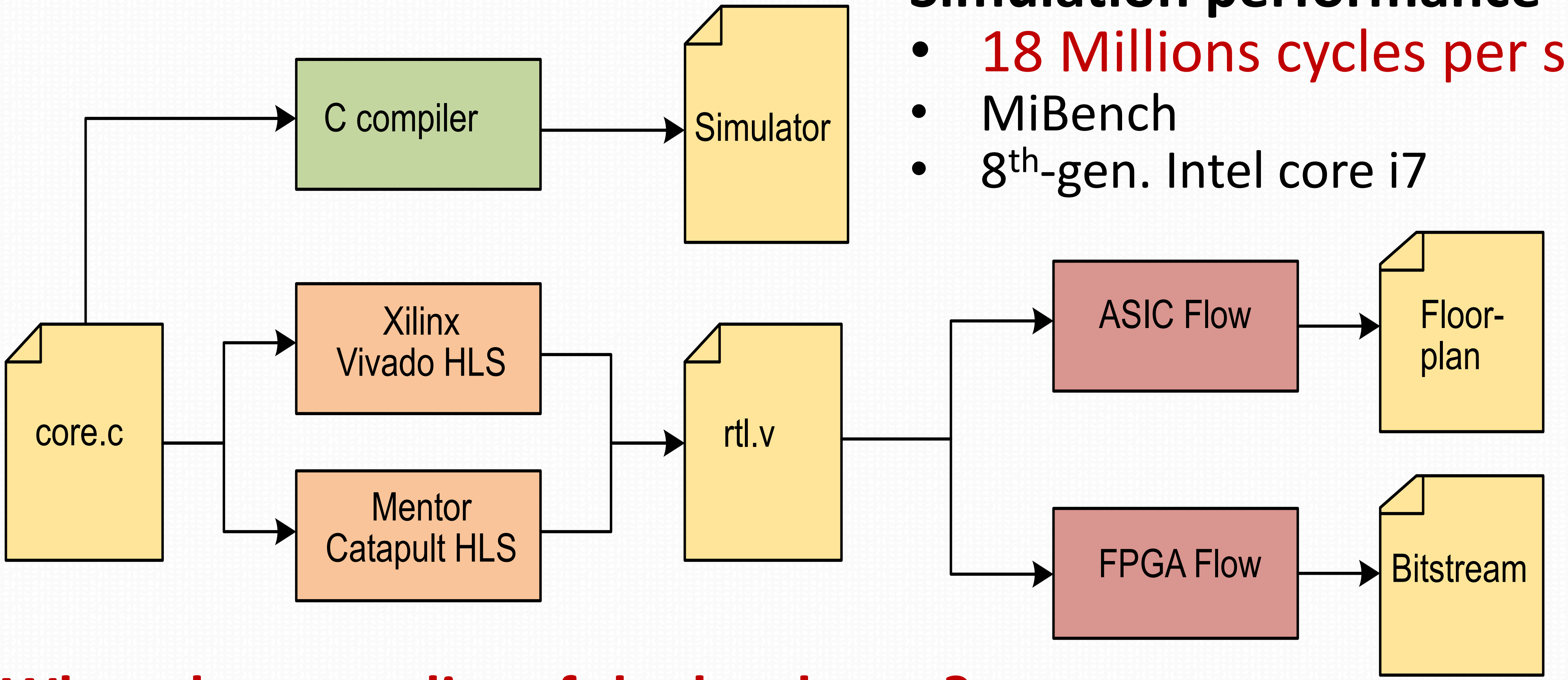
- A C++ simulator handles all features that cannot be synthesized
- Two ways of handling syscalls
  - Syscall emulation
  - Through an OS

(Ongoing work) Handling interrupts

- Support for CSR registers
- Ecall instruction fire an interrupt
- When external interrupt (if not masked)
  - Disable fetched instruction
  - Save current PC
  - Jump to ISR

- Not synthesized yet
- Simulator can boot RT os

Design and Validation Flow



Simulation performance

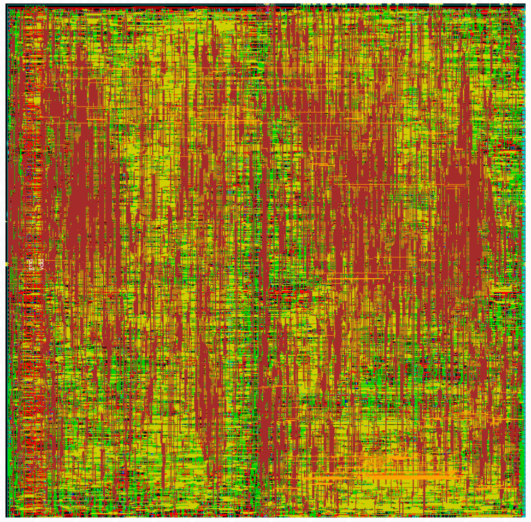
- 18 Millions cycles per sec.
- MiBench
- 8th-gen. Intel core i7

What about quality of the hardware?

Experimental Study - ASIC

- Comparison of Comet against similar implementations
- Target technology is STMicro 28nm FDSOI

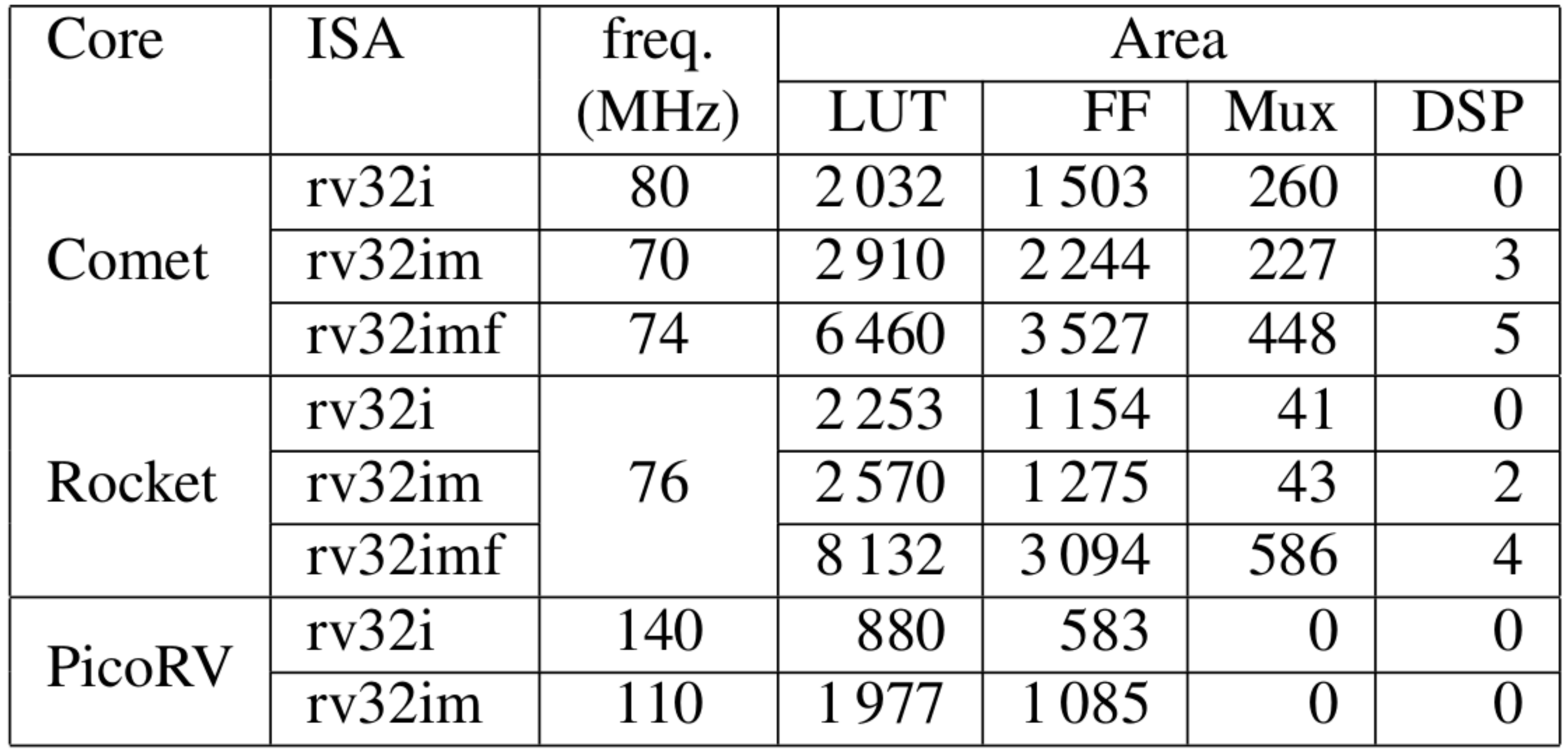
Core	ISA	freq. (MHz)	Area (μm^2)	Lang.
Comet [14]	rv32i	700	8 168	C++
	rv32im		11 099	
	rv32imf		26 760	
Rocket [12]	rv32i		11 114	Chisel
	rv32im		12 606	
	rv32imf		26 550	
PicoRV [15]	rv32i		7 747	Verilog
	rv32im		11 176	



Design	Area (μm^2)
FPU	8 147
Core w/o FPU	15 299
Core w/ FPU	26 760

Experimental Study - FPGA

- Comparison of Comet against similar implementations
- Target technology is Xilinx Artix 7 (Catapult + Vivado 2018.3)



Core	ISA	freq. (MHz)	Area			
			LUT	FF	Mux	DSP
Comet	rv32i	80	2 032	1 503	260	0
	rv32im	70	2 910	2 244	227	3
	rv32imf	74	6 460	3 527	448	5
Rocket	rv32i	76	2 253	1 154	41	0
	rv32im		2 570	1 275	43	2
	rv32imf		8 132	3 094	586	4
PicoRV	rv32i	140	880	583	0	0
	rv32im	110	1 977	1 085	0	0

Pros/Cons of Proposed Paradigm

- Pros
  - Debugging is done at C++ level
  - Fast simulation using the C++ simulator (~20.10^6 cycles per second)
  - Simulator is equivalent to RTL model
  - Modifying the core is simpler than at HDL level
  - Software development techniques (e.g., continuous integration)
- Cons
  - Some features are difficult to describe (e.g. multi-cycle operators)
  - Pipeline has to be explicit
  - HLS tools may have trouble synthesizing multi-core systems
  - Late modifications (e.g. metal fix)

Conclusion & Roadmap

- Efficient processor core design (HW µarch + SW simulator) from a single C++ code
- Current projects
  - VLIW Dynamic Binary Translation, Non-Volatile Processor, Fault-Tolerant Multicore
- Perspectives
  - Dedicated source-to-source transformations for HLS
  - Multi-core system with cache coherency (Q4 2019)
  - Many-core system with NOC (2020)

Questions

Thank you for your attention!

?

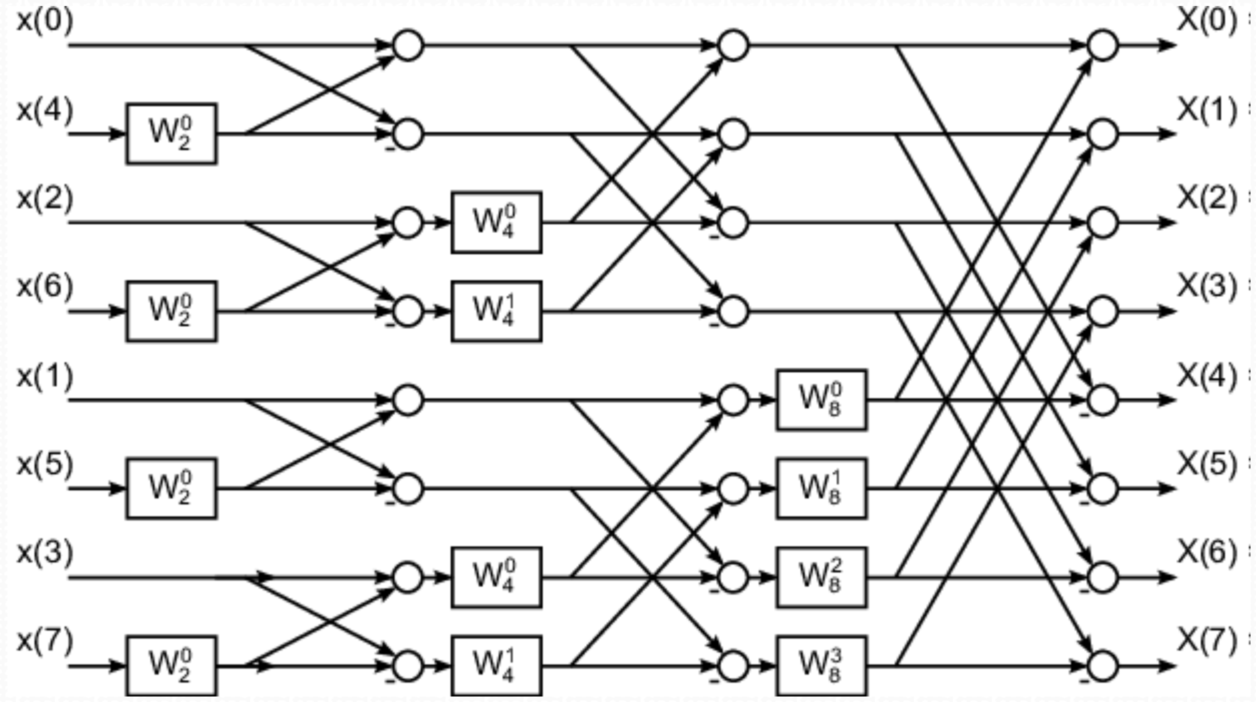
`https://gitlab.inria.fr/srokicki/Comet`

Tracing – Code Instrumentation

- Instrumenting Comet simulator is simple
- Overload function *everyCycle()* from the simulator
  - Example: counting stall caused by cache miss

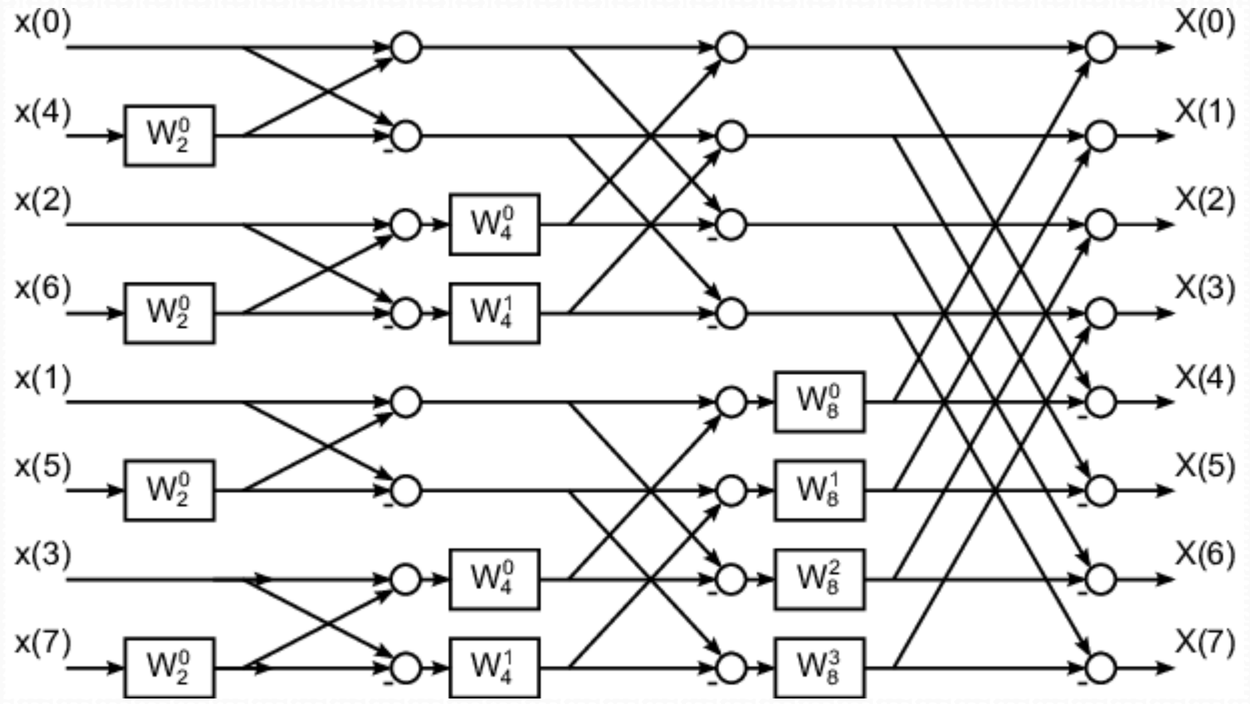
Adding Custom Instruction

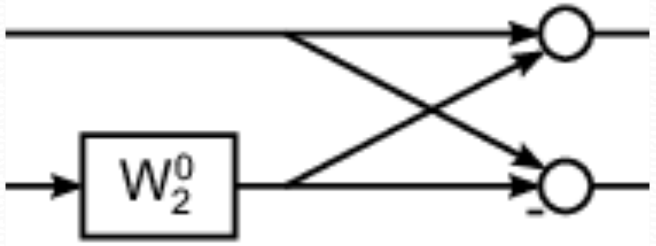
- Custom FFT accelerator as a custom instruction
- Overview of FFT algoritm



Adding Custom Instruction

- Custom FFT accelerator as a custom instruction
- Overview of FFT algoritm





Butterfly computation

Adding Custom Instruction

- What can custom instructions do?
  - Read two registers
  - Write one result
  - Can stall if needed
  - Can encode internal state machine

- Custom instruction limitations
  - Cannot access memory
  - Cannot use more than 2R/1W

Adding Custom Instruction

- Butterfly computation:
  - Reads two complex values
    - Two 16-bit fixed-point values in a 32-bit register
  - Writes two complex values
    - Main custom instruction computes the two values and output first value
    - Second custom instruction output the second value
  - Uses two twiddle factors
    - Index encoded in immediate field
    - Use of internal state machine to select the one to use

Adding Custom Instruction

- Results:
  - Around 50 lines of C code in the ALU
  - Execution of FFT is 14x faster
  - Core area increased by 31%   (baseline is rv32im)