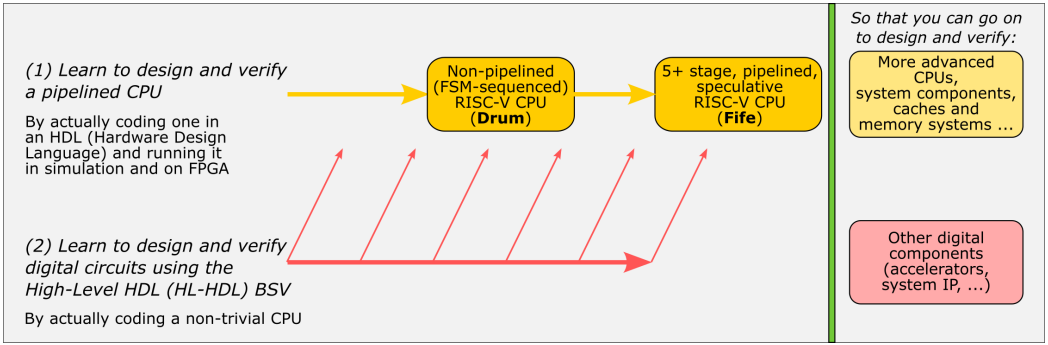
Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L1: Introduction

Goals for this book/course



These goals are complementary:

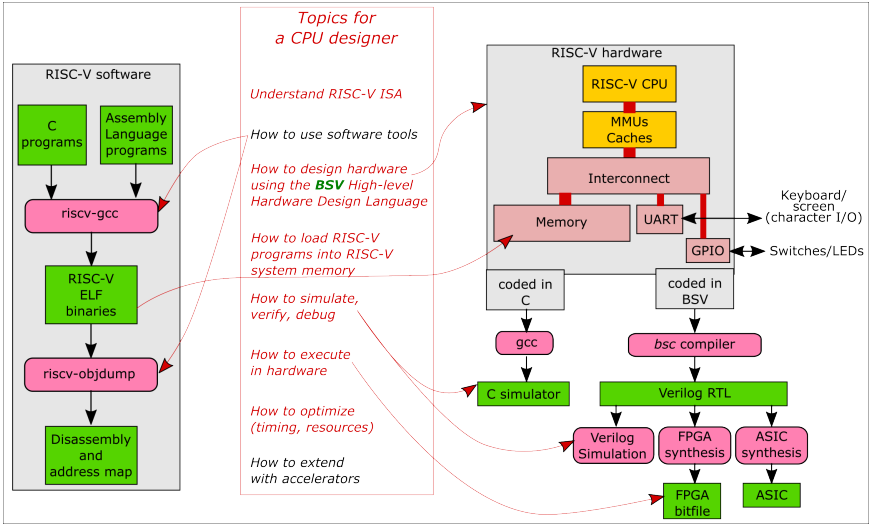
- One cannot really learn CPU design without actually doing it, for which we need an HDL.
- One cannot really learn an HDL without actually doing it, for which we need interesting examples.

BSV

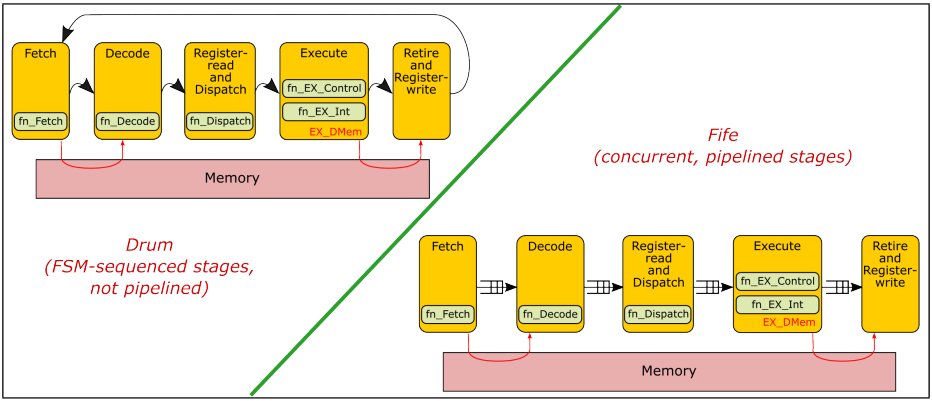
Learn BSV to design the CPUs

- BSV is a high-level hardware design language (HL-HDL)
- Modern (features on par with modern software programming languages)
- More than 20 years usage in industry and academia for major hardware designs

Topics we will cover (in red text)



Two CPU implementations (microarchitectures): Drum and Fife



The two implementations share much code capturing RISC-V functionality

Why two microarchitectures?

- *Incremental learning:* we can master all the *functional* aspects of RISC-V with Drum, and focus entirely on pipelining aspects with Fife.

Why two microarchitectures?

- *Incremental learning:* we can master all the *functional* aspects of RISC-V with Drum, and focus entirely on pipelining aspects with Fife.
- *Functional debugging:* Bugs may be due to:
  (A) wrong implementation of RISC-V functionality (*e.g.,* address calculation)
  (B) wrong implementation of pipelining (*e.g.,* register read/write hazard)

  Drum allows us to debug all aspects of (A) in a simpler system (no complications due to pipelining).

  All this carries over directly into Fife, so when debugging Fife, we can focus on (B).

  Drum can be viewed as a *synthesizable reference model* for Fife or other RISC-V CPUs.

Why two microarchitectures?

- *Incremental learning:* we can master all the *functional* aspects of RISC-V with Drum, and focus entirely on pipelining aspects with Fife.
- *Functional debugging:* Bugs may be due to:
  (A) wrong implementation of RISC-V functionality (*e.g.,* address calculation)
  (B) wrong implementation of pipelining (*e.g.,* register read/write hazard)

  Drum allows us to debug all aspects of (A) in a simpler system (no complications due to pipelining).

  All this carries over directly into Fife, so when debugging Fife, we can focus on (B).

  Drum can be viewed as a *synthesizable reference model* for Fife or other RISC-V CPUs.
- *Both implementations are useful in different applications:*
  - Drum uses smaller circuits (microcontrollers, embedded, IoT, ...)
  - Fife is faster, larger, for more demanding applications

Book and full source code; ready to build and execute

This course is based on:

- a free PDF textbook
- full source code for Drum and Fife
  - Free and open-source
  - Drum and Fife CPU sources are written in BSV. Parts of the testbench are written in C.

Book and full source code; ready to build and execute

This course is based on:

- a free PDF textbook
- full source code for Drum and Fife
  - Free and open-source
  - Drum and Fife CPU sources are written in BSV. Parts of the testbench are written in C.

Use free and open-source tools to build Drum and Fife into simulation executables, and have them execute RISC-V binaries (ELF files, output of *gcc*).

- BSV code needs the *bsc* compiler.
- Two ways to simulate:
  - Bluesim (standalone; just need *bsc*)
  - Standard Verilog simulation
    We will demostrate using Verilator, but you can use any Verilog simulator.

Book and full source code; ready to build and execute

This course is based on:

- a free PDF textbook
- full source code for Drum and Fife
  - Free and open-source
  - Drum and Fife CPU sources are written in BSV. Parts of the testbench are written in C.

Use free and open-source tools to build Drum and Fife into simulation executables, and have them execute RISC-V binaries (ELF files, output of *gcc*).

- BSV code needs the *bsc* compiler.
- Two ways to simulate:
  - Bluesim (standalone; just need *bsc*)
  - Standard Verilog simulation
    We will demostrate using Verilator, but you can use any Verilog simulator.

We will also demonstrate synthesis of Drum and Fife, and running on an FPGA.
(we will use an Amazon AWS "F1 instance", but you can use any FPGA).

Teaching methodology

Learn RISC-V CPU design and BSV together:

- We will interleave RISC-V and BSV topics, to learn them together.
- We will learn a little BSV, then apply that to some part of the RISC-V CPU design.
- We will repeat this process until we have the full CPU designs.
- Every example will involve actual BSV code for Drum and/or Fife.

Learn Drum first, then Fife:

- We will learn all of Drum first, and have Drum execute RISC-V programs.
- Then we will learn pipelining principles, and how we code them for Fife.
- Fife executes the same RISC-V programs as Drum.

Debugging and optimization:

- Along the way we will develop testbenches for components and demonstrate debugging the BSV code.
- After Fife is running, we will discuss deeper optimizations for Drum and Fife.

Teaching methodology: an unconventional approach

Most courses on hardware design spend a lot of time on circuit diagrams and schematics, boolean logic gates (AND, OR, NOT, XOR, ...), combinational circuits, logic minimization, flip flops, registers, binary number representations (twos complement), FSMs (Finite State Machines), and so on.

However, most modern digital hardware design is done at a higher level of abstraction. Designs are described using languages like Verilog, SystemVerilog and VHDL, and we rely on *compilers* to produce gates, wires, registers, *etc.*. Accordingly, we will focus on language-based hardware design, and not so much on the detailed underlying circuits.

With BSV, because of its rule-based execution model, we can take it even further; we will not even have to talk about clocks and clock timing until late in the course.

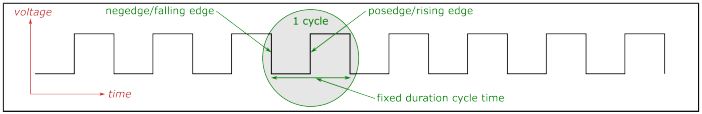
Analogy

Most modern programs are written in high-level languages, which are compiled into machine code using compilers. It is not necessary to understand machine code in order to write programs (unless you are trying to optimize at the bleeding edge of high performance).

General operation of digital circuits

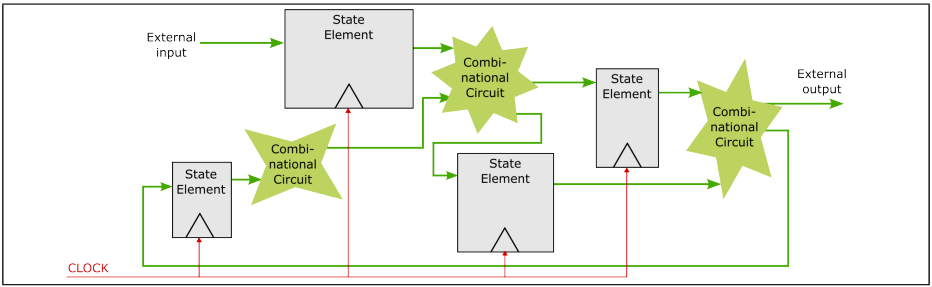
We now provide just enough intuition about digital circuits to allow us to proceed with BSV design.

Digital hardware is driven by a *clock*, an electrical signal that oscillates between a low voltage and a high voltage, with sharp transitions between the two, at regular intervals (it is therefore sometimes called a "square" wave).



Any value that must be visible from one clock to the next must be stored in a *state element* (a component that can "remember" values, *i.e.,* has *state*, like a register, FIFOF, buffer, memory, ...).

General structure and operation of digital circuits



Digital circuits are composed of *state elements* and *combinational circuits* connecting outputs of state elements to inputs of state elements. State elements are *storage* elements. Combinational circuits are *acyclic* networks of *logic gates* (AND, OR, NOT, ...). Any digital circuit repeatedly performs the following:

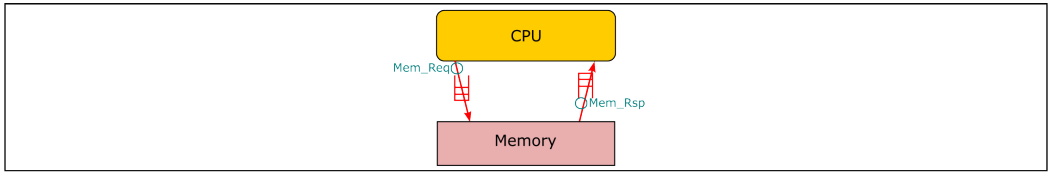
- After each "posedge" (positive edge of a clock), the combinational circuits compute new inputs for all state elements, based on the current values in state elements
- At the next posedge, these new input values are stored into the state elements.

General operation of digital circuits; notes and caveats

- Nowadays, clock cycle times/periods are typically measured in nanoseconds.
  For example, a 100 MHz clock has a 10ns cycle time.
- Physics and circuit silicon technology dictates how long an electrical signal takes to propagate from the output of a state element and through a combinational circuit before it arrives at the input of a state element. The clock period needs to be greater than this.
  In the *digital abstraction* we assume this condition is met, and we "idealize" wire and combinational circuit delays as taking zero time.

- Advanced digital ciruits may have multiple clocks driving different parts of the circuit.
- Advanced digital ciruits may vary the clock speed dynamically, to match power consumption to current performance demands (higher speeds $\Rightarrow$ more power consumption).
- Some digital ciruits use the negative edge ("negedge") of the clock insted of the posedge.
- Advanced digital ciruits may use both the posedge and the negedge of the clock.

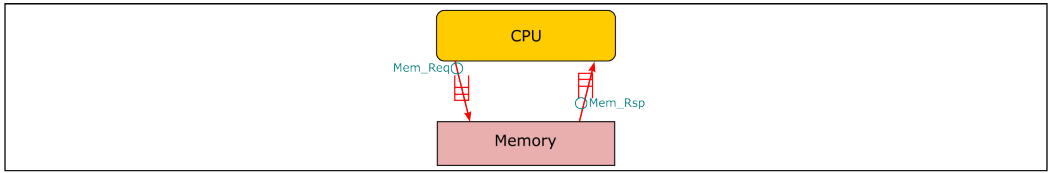
Computer Memories



A computer memory is a packaged device providing *random access* to a sequence of locations:

- To read memory, we present a *read request* at its inputs, with an *address*; the memory provides a *read response* at its outputs, containing the addressed data.

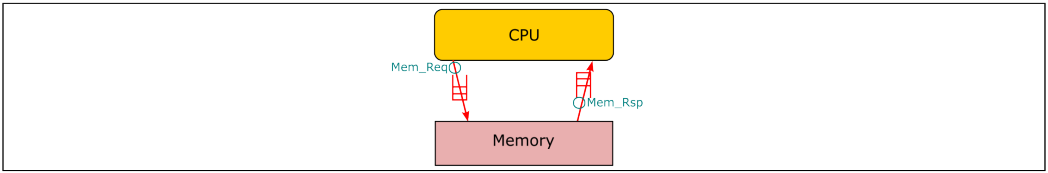
Computer Memories



A computer memory is a packaged device providing *random access* to a sequence of locations:

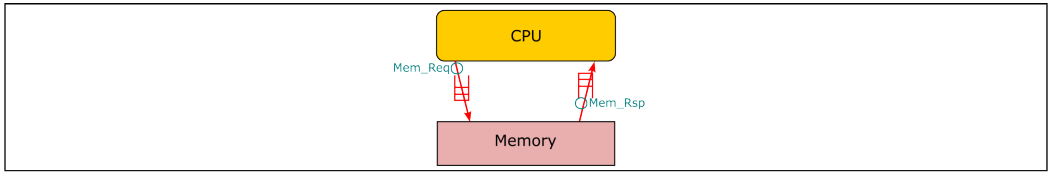
- To read memory, we present a *read request* at its inputs, with an *address*; the memory provides a *read response* at its outputs, containing the addressed data.
- To write memory, we present a *write request* at its inputs, with an address and data; the memory provides a *write response* at its outputs, containing an "ok" status.

Kinds of Computer Memory



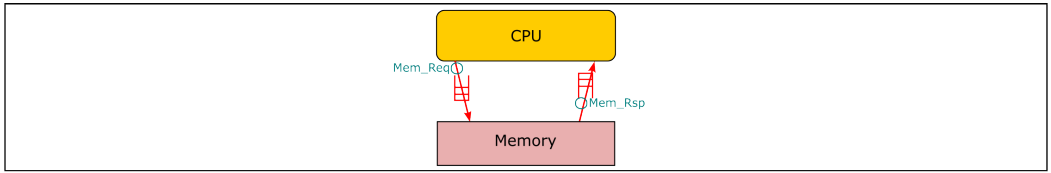
- SRAMs (Static Random Access Memories)
  - Typically from a few kilobytes to a few megabytes
  - Faster than DRAM
  - More power-consumption than DRAM
  - Often used for "caches" close to the CPU.
- DRAMs (Dynamic Random Access Memories)
  - Typically from a few megabytes to gigabytes
  - Slower than SRAM
  - Less power-consumption than SRAM
  - Often used for caches and "main memory" further away from the CPU.

Split-phase Memory Access



Typically (for any memory larger than a few kilobytes), the response is available *one or more clocks after* the inputs were presented. We also say that memory accesses are split-phase—requiring temporally separated request-production and response-consumption.

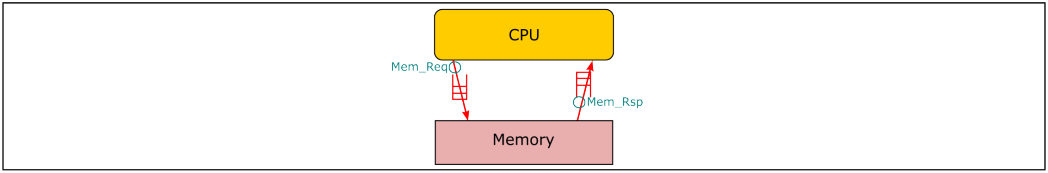
Split-phase Memory Access



Typically (for any memory larger than a few kilobytes), the response is available *one or more clocks after* the inputs were presented. We also say that memory accesses are split-phase—requiring temporally separated request-production and response-consumption.

The delay from when the CPU issues a request until when it can collect the response is called *memory latency* (units: clock cycles, nanoseconds).

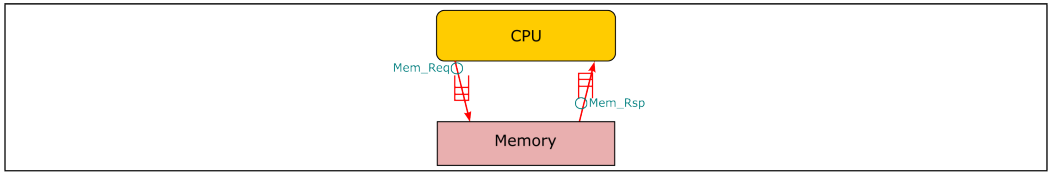
Pipelined Memory Access



The CPU ⇒ memory ⇒ CPU path is often pipelined and behaves like a queue:

- The CPU can pump in requests towards memory without waiting for responses.
- Concurrently, another part of the CPU can pump out responses from memory.

Pipelined Memory Access

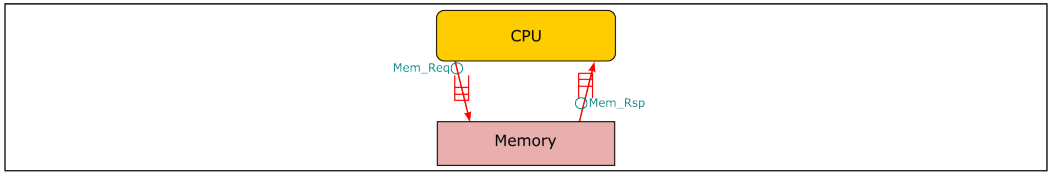


The CPU ⇒ memory ⇒ CPU path is often pipelined and behaves like a queue:

- The CPU can pump in requests towards memory without waiting for responses.
- Concurrently, another part of the CPU can pump out responses from memory.

The rate at which the CPU can pump requests is called *memory bandwidth* (units: transactions/second, requests/second, bytes/second).

Pipelined Memory Access



The CPU ⇒ memory ⇒ CPU path is often pipelined and behaves like a queue:

- The CPU can pump in requests towards memory without waiting for responses.
- Concurrently, another part of the CPU can pump out responses from memory.

The rate at which the CPU can pump requests is called *memory bandwidth* (units: transactions/second, requests/second, bytes/second).

Latency and bandwidth can vary (may depend on size of memory request, cache misses, virtual memory page faults, cache coherence traffic, ...).

End