Alternative HDLs & SoC Generators

98-154/18-224/18-624: Intro to Open-Source Chip Design

Alternative HDLs

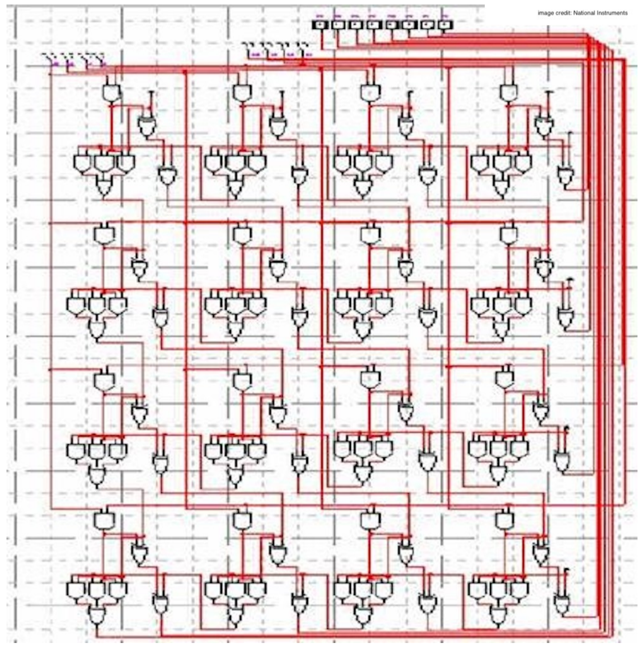
- HDL = Hardware Description Language
- Common ones are (System)Verilog and VHDL, used by most of the industry
- They are very widely-used but tend to have weird quirks and limitations
  - i.e. Verilog `wire` vs `reg` types don't correspond to wires and registers
- Recently, there has been a push for greater abstractions and cleaner languages

Taxonomy of HDLs

- Structural Design (Schematics, Verilog/VHDL)
- Behavioral / RTL Design (Verilog/VHDL)
- Abstracted RTL Design (Chisel, Amaranth, Hardcaml)
- Dataflow-Level Design (PipelineC, DFiantHDL)
- High-Level Synthesis (Google XLS, Vitis HLS)
- Code-Generators (MATLAB HDL Coder, Vitis AI)

Structural Design

- Gate-level design using a schematic or gate-primitives in Verilog/VHDL
- Rarely used other than for designing extremely-small, frequently-reused modules



Structural Design: Pros and Cons

- Pro: Complete control over how design will be implemented
- Pro: Useful for introspection into low-level design
- Con: Tedious to design anything complex, even with hierarchy
- Con: Difficult to read and understand code
  - Overuse of hierarchy can make this even worse

Behavioral/RTL Design

```
module branch_controller (
  input wire [31:0] i_rs1, i_rs2,
  input wire [2:0] btype,
  output logic taken);

always_comb begin
  case (btype[2:0])
    BR_EQUAL: begin
      taken = (i_rs1 == i_rs2);
    end
    BR_LT_SIGNED: begin
      taken = ($signed(i_rs1) < $signed(i_rs2));
    end
    BR_LT_UNSIGNED: begin
      taken = (i_rs1 < i_rs2);
    end
...
```

Behavioral/RTL Design: Pros and Cons

- Pro: High degree of flexibility but still easy to write
- Pro: Code is (mostly) understandable and readable
- Con: Easy to write code that will synthesize poorly
- Con: Lack of strong compile-time abstractions
    - Hacky workarounds can make code much harder to read

Abstracted RTL Design

- Similar to RTL design but with higher degree of abstraction
- Based on a software language but NOT a software-to-hardware compiler
    - Languages like Chisel3 (Scala), Migen/Amaranth (Python), Hardcaml (OCaml)
- Provides features like custom types, better compile-time generation, etc.

Abstracted RTL Design

- The software is executed and generates the hardware using a Domain Specific Language
- "Elaboration" = the process of executing the code and generating hardware
- Certain operators and datatypes are generated into hardware when "executed"
  - i.e. `Reg` and `Wire` types in Chisel, the `=` operator assigns at compile-time, `:=` is connection generated in hardware
- Language built-in datatypes and operators are evaluated at elaboration-time (more examples soon)

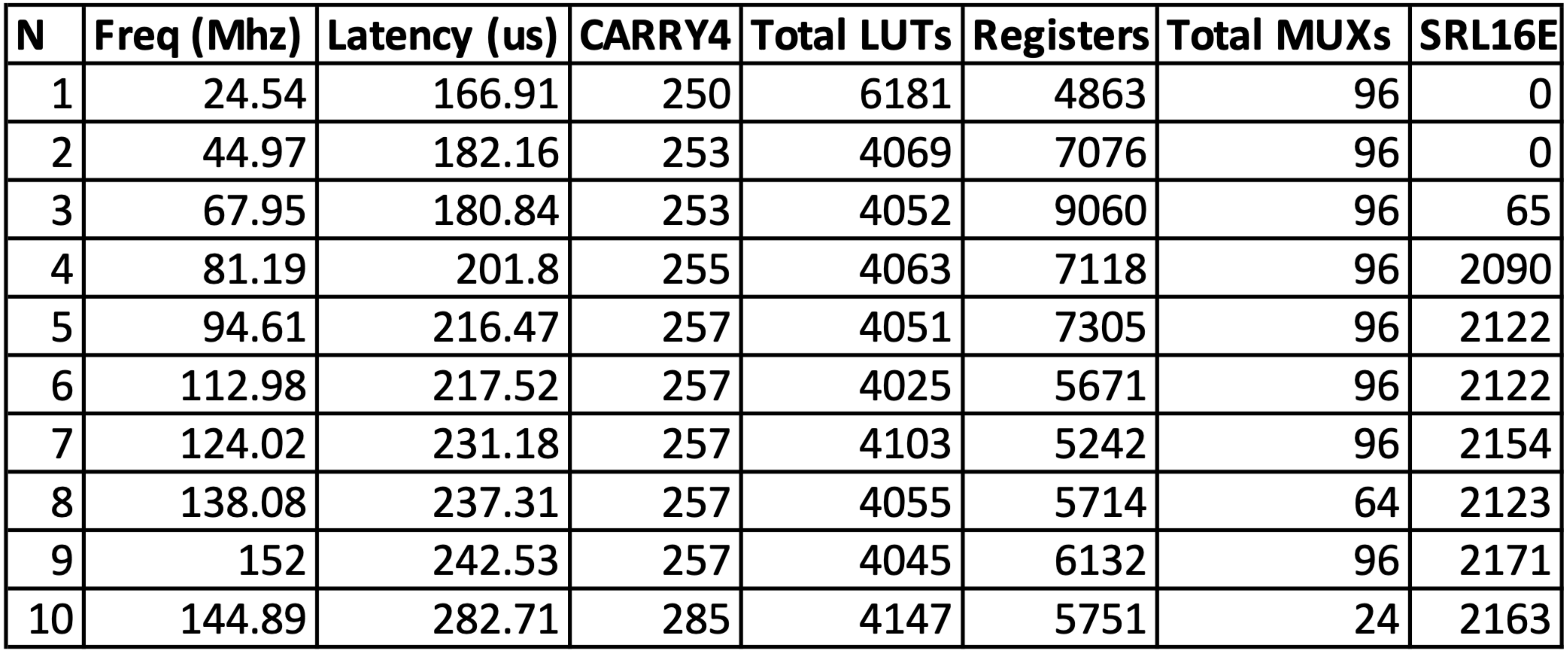
Abstracted RTL Design: Pros and Cons

- Pro: Sophisticated compile-time generative functionality
- Pro: Better syntax-sugar, faster to write complex designs
- Pro: Allows for things like SoC generation without external software tools
- Pro: Relatively low barrier-to-entry (can start by line-for-line translating existing RTL, and then add abstractions later)
- Con: Outputted Verilog not very clean, hard to debug
- Con: Not much industry adoption yet

Dataflow-Level Design

- Dataflow abstractions are agnostic to cycle and timing constraints, instead focused on movement of data and their dependencies.
- Anything implemented as a synthesizable pure function can be auto-pipelined to meet any required clock speed
- Great for datapath-heavy code, but inefficient for designs with complex state
- Languages like PipelineC and DFiantHDL

Dataflow-Level Design: PipelineC Example



N	Freq (Mhz)	Latency (us)	CARRY4	Total LUTs	Registers	Total MUXs	SRL16E
1	24.54	166.91	250	6181	4863	96	0
2	44.97	182.16	253	4069	7076	96	0
3	67.95	180.84	253	4052	9060	96	65
4	81.19	201.8	255	4063	7118	96	2090
5	94.61	216.47	257	4051	7305	96	2122
6	112.98	217.52	257	4025	5671	96	2122
7	124.02	231.18	257	4103	5242	96	2154
8	138.08	237.31	257	4055	5714	64	2123
9	152	242.53	257	4045	6132	96	2171
10	144.89	282.71	285	4147	5751	24	2163

Dataflow-Level Design: Pros and Cons

- Pro: Auto-pipelining allows running at very high clock speeds
- Pro: Harder to make mistakes related to misaligned delay paths
- Con: Inefficient and hacky for designs for state-heavy designs
- Con: Difficult to learn, unusual language syntax

High-Level Synthesis

- Holy grail: write software code and turn it into hardware
- No hardware knowledge needed, let the tools take your algorithm and do the hard work (Google XLS, Vitis HLS, etc.)
- **This will make hardware engineers irrelevant.**

High-Level Synthesis

- LIES! LIES! IT WAS ALL LIES!
- THERE IS NO HIGH-LEVEL SYNTHESIS

High-Level Synthesis

- Vendors loooooove to claim that software engineers can use their HLS tools to design hardware
- The reality is that it only works in very constrained cases.
- Great for DSP, ML, CV, and similar compute-acceleration
  - Loop unrolling, autopipelining, etc. are very easy wins
- Control-flow-heavy designs will require more effort to implement efficiently in HLS than just using RTLs
- Very, very, **very** easy to shoot yourself in the foot and generate inefficient designs by writing software-y code, need intuition about how code maps onto hardware

Code Generators

- Generate hardware by chaining together existing blocks using configuration
- Vitis AI DPU
- Vitis AI Models
- Simulink HDL Blocks

Amaranth Example

```python
from nmigen import *

class TestModule( Elaboratable ):
  def __init__( self ):
    self.count = Signal( 16, reset = 0 )
    self.ncount = Signal( 16, reset = 0 )

  def elaborate( self, platform ):
    m = Module()
    m.d.comb += self.ncount.eq( ~self.count )
    m.d.sync += self.count.eq( self.count + 1 )
    return m
```

Amaranth Examples

- Basics: Modules
- If/Else
- More detailed tutorial
- Other tutorials

Chisel Examples

Notebook

Chisel Example: Encryption Core

GitHub

System-on-Chip

- SoC = System-on-chip, generally includes a CPU, memories, and peripherals
- Tied together using a bus interface such as wishbone, AXI, TileLink, etc.
- Peripherals are usually memory-mapped

Automated SoC Generation

- Instantiating and wiring together components onto a bus, along with all of the bus interconnections, is very tedious to do by hand
- Since it is so templated and repetitive, there are software-like tools which generate SoCs given a configuration
- Example SoC Design

SoC Generation: CPUs

Many open-source RISC-V CPUs:

- SERV (smallest RISC-V CPU)
- PicoRV32 (small, simple design)
- VexRiscv (simple design)
- biRISC-V (simple in-order superscalar)
- Rocket-Core (UC Berkeley)
- BOOM (UC Berkeley)
- SweRV (WD Corp.)

SoC Generation: Busses

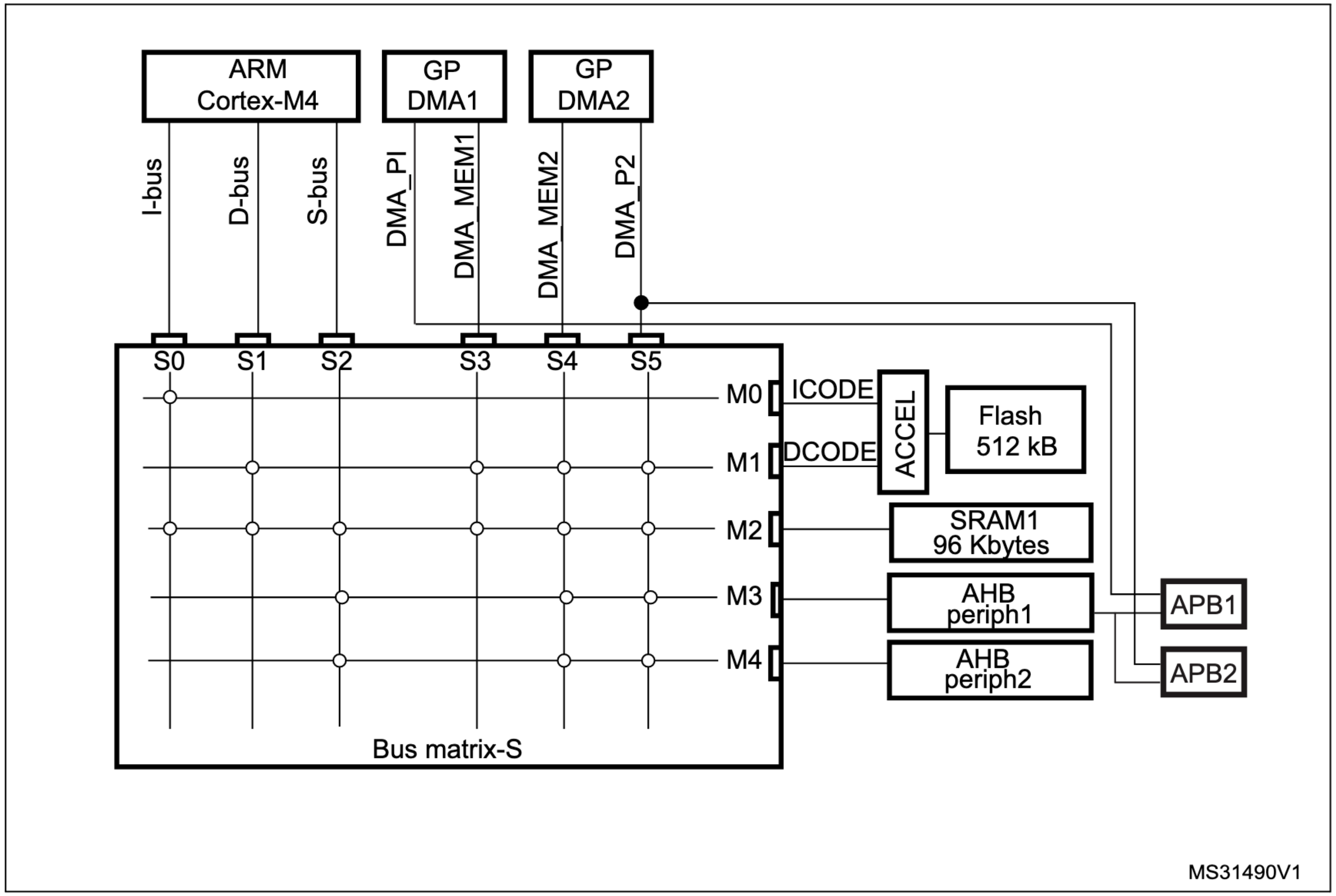
- Bus interface in an SoC is usually in the form of a request-response, memory-style interface
- Request: coordinator sends an address and read or write request
- Response: participant which handles said address responds with acknowledgement and data (if read)

SoC Generation: Interconnects

- Arbiter: Connect multiple coordinators to one participant
- Decoder: Connect one coordinator to multiple participants
- Interconnect: Connect multiple coordinators to multiple participants, one device at a time (arbiter + decoder)
- Crossbar: Connect multiple coordinators to multiple participants, multiple parallel connections allowed

SoC Generation: Bus Matrix

Figure 4. Multi-AHB matrix



SoC Generation: Memories

- Instruction RAMs and caches
- Data RAMs and caches
- External memories may have high latency which requires caching or prefetching; this must be accounted for when setting timeouts on the bus interface.

SoC Generation: I/O Devices

- CPU is not very useful if it can’t communicate with the outside world
- Low-speed peripherals: UART, SPI, I2C, etc.
- High-speed peripherals: USB, Ethernet, HDMI, etc.
- Usually use a DMA when sending large amounts of data through an I/O device
  - DMA = direct memory access; sends a block of RAM to a peripheral device without CPU intervention

SoC Generation Example

GitHub

Exercise: Chisel3 Interactive Tutorial

https://mybinder.org/v2/gh/freechipsproject/chisel-bootcamp/master

https://inst.eecs.berkeley.edu/~cs250/sp17/handouts/chisel-cheatsheet3.pdf

- Chisel interactive tutorial, in a Jupyter notebook style
- Start with either 2.1 or 2.5 depending on your adventurousness