Tampereen yliopisto
Tampere University

Practical SoC Development

Excercise Project Introduction

Antti Nurmi
Tampere University
antti.nurmi@tuni.fi

Outline

- Project Introduction
- Building your own DMA
- Introduction to Verilator Simulation
- Embedded Software Workflow
- Project Practicalities

Project Introduction

Project Overview (1/4)

Your task is to design an AXI4 DMA.

- Implementation in SystemVerilog.
- Conformance to standard AXI4 interfaces.
- Integration to Didactic & baseline application.

Project Overview (2/4)

Part 1: Module-Level Design

- We will provide a testbench to design against and excercise the basic design functionality while you construct your design.
- Fixed test sequence with **[pass/fail]** report.
- Early feedback and testing rather than exhaustive verification.
  - Architecture & hierarchy
  - Compilation, linting, formatting

Project Overview (3/4)

Part 2: System-Level Integration

- **Integrate** your DMA to an instance of Didactic.
- **Modify** an existing baseline application to replace CPU-based memory operations with DMA transfers.
- **Analyse** the performance of the baseline application and DMA-accelerated version.

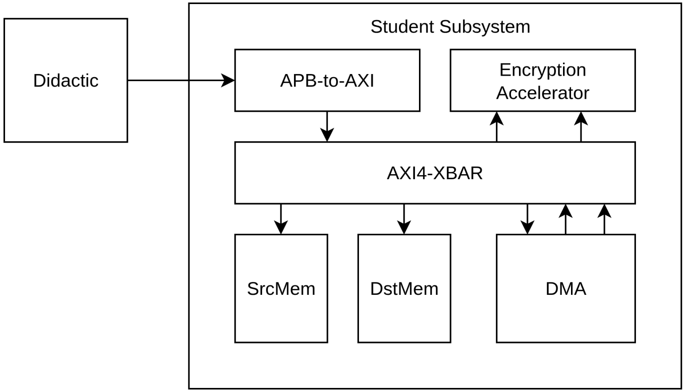
Project Overview (4/4)

- **Part 3: Packaging**
  - Put yourself in the shoes of someone seeing your design for the first time:
    - What is this, what does it do?
    - What features and interfaces does this support?
    - How do I use and program this?
  - Write **at least** a `README.md` file to document your design and answer these questions.
    - The more the better: extra points available by writing more extensive and better documentation in the `./doc` folder.

Exact Specification

… under construction. Will be located in Plussa when ready.

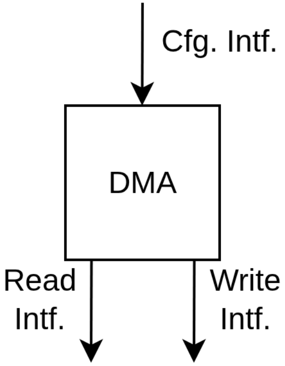
Full System Overview



Building your own (Simple) DMA

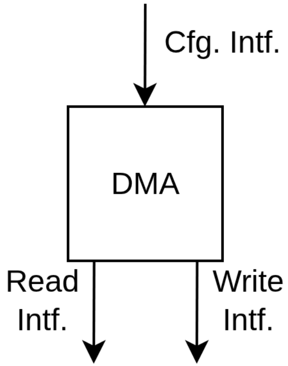
What is a DMA? (1/2)

- A unit for Direct Memory Access (DMA).
  - May also be called a Data Movement Accelerator.
- Offloads data movement from CPU.
  - CPU can do other useful work, or sleep.
- Used in some forms in most data-heavy applications. [1]

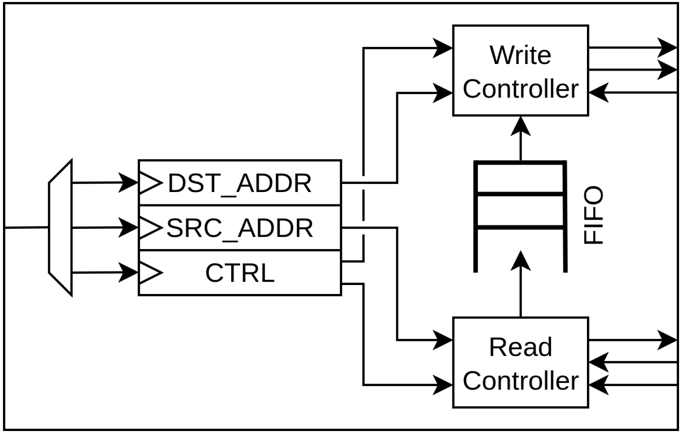


What is a DMA?

- Three interfaces: Read (M), Write (M), Cfg (S).
- Read & write interfaces typically on same, high-bandwidth data plane.
- Configuration interface programmed by CPU.



Internal Structure



Programming Model (1/2)

- Pseudocode for programming a generic DMA:

```
void dma_move( uint32_t src, uint32_t dst, uint8_t len_words ) {
    write_u32(src, DMA_SRC_REG);       // Set source
    write_u32(dst, DMA_DST_REG);      // Set destination
    write_u8(len, DMA_TX_LEN_REG); // Set transaction length
    set_bit(DMA_START_BIT);
}
```

Programming Model (2/2)

- Functionally equivalent to:

```
void fake_dma_move( uint32_t src, uint32_t dst, uint8_t len_words ) {
  for (uint8_t i=0; i<len; i++) {
    uint32_t tmp = read_32(src + i*4);
    write_u32(tmp, (dst + i*4));
  }
}
```

Introduction to Verilator Simulation

What is Verilator[2]? (1/2)

- A cycle-accurate, 2-state RTL-simulator.
- Free & open-source, developed since 1994.
- Compiles RTL into C++ model & native host executable.
  - Fast, lightweight.

What is Verilator[2]? (2/2)

- Motivating Principle: Performing simulations on the **highest appropriate abstraction level**.
  - (Strawman) Would you want to run your **first** LED-blinker simulation in gate-level simulation?

- Verilator is not a replacement for full RTL simulators, but **can offload a lot of work from them.**

Simulation Flow

- The Verilator flow is structured to be efficient and modular.
  - Aim for this generally, regardless of what simulator you use.
  - Better efficiency, suitability to CI, etc.
- Aim for fast interations, only recompile what’s changed.

Simulation Flow (2/3)

- Commands:

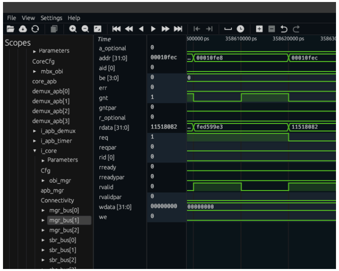
```
make elf TEST=<testname>     # Compile C source, produce ELF
make verilate                # Compile HW simulation model
make simv TEST=<testname> # Run Verilator simulation with testcase
```

- Can be combined, e.g.:

```
make elf simv TEST=<testname> # Recompile C source & run
```

Simulation Flow

- Running simulations produces a `.fst` waveform file.
- Can be opened on a dedicated viewer.
  - GTKWave
  - Surfer[1]



[1] https://surfer-project.org/

Embedded Software Workflow

Motivation

- Recall lecture 3: processors are everywhere.
  - Software-driven development is a flexible and powerful methodology for SoC develpment.
- Simulating embedded software in RTL is reasonably fast.
  - At least for small designs & simple software.
  - Gives **complete** visibility into the system state.

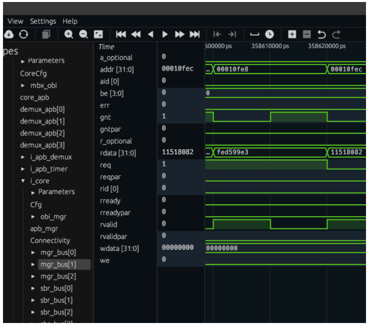
From C Source to Waveform

- How do we bridge this gap?

```c
#include <stdio.h>

int main() {
    printf("Hello World");
    return 0;
}
```

$\longrightarrow$



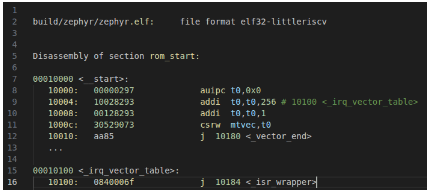
Step 1: C to ELF

- `make elf` invokes the RISC-V compiler (`riscv{32,64}-unknown-elf-gcc`) to produce an **ELF**[2]-file.
  - A common format for loading compiled programs.
  - Can be used to extract hexadecimal stimulus dumps.
  - Binary format, **not human-readable**.

[2] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Step 2: ELF to Object Dump

- `riscv{32,64}-unkown-elf-objdump` allows us to **statically** extract meaningful information from the compiled ELF.
  - Machine instructions
  - Register usage
  - Memory layout
- Does **not** describe dynamic program & machine behaviour.



```

build/zephyr/zephyr.elf:     file format elf32-littleriscv


Disassembly of section rom_start:

00010000 <__start>:
   10000:   00000297            auipc t0,0x0
   10004:   10028293            addi  t0,t0,256 # 10100 <_irq_vector_table>
   10008:   00128293            addi  t0,t0,1
   1000c:   30529073            csrw  mtvec,t0
   10010:   aa85                j  10180 <_vector_end>
   ...

00010100 <_irq_vector_table>:
   10100:   0840006f            j  10184 <_isr_wrapper>
```

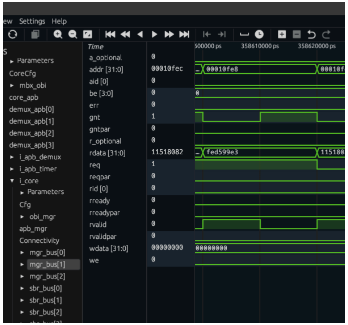
Step 3: Object Dump to Instruction Trace

- Ibex can generate a trace of executed instructions.
- Contains **dynamic** register and memory values.
- Generated in `./verilator/trace_core_00000000.log`.

```
PC        Insn.     Decoded Insn.      Register and memory contents
00010000  00000297  auipc x5,0x0        x5=0x00010000
00010004  10028293  addi  x5,x5,256     x5:0x00010000  x5=0x00010100
00010008  00128293  addi  x5,x5,1       x5:0x00010100  x5=0x00010101
0001000c  30529073  csrrw x0,mtvec,x5   x5:0x00010101  x0=0x00000000
00010010      aa85  c.j  10180
```

Step 4: Instruction Trace to Waveform

- The instruction trace can be cross-referenced to the waveform by understanding the system **architecture, hierarchy and memory mapping**.
  - CPUs $\longrightarrow$ interconnects
  - Interconnects $\longrightarrow$ memories, peripherals
- Note: All of the above steps are automated and provided to you, please use them!



Project Practicalities

Git gud (1/2)

- You are expected to know how to use Git.

Git gud (2/2)

- We may consider Git history and usage in grading.
  - Aim for relatively small, concise commits.
  - Write descriptive commit messages.
- **Check that any requested Git tags are pushed to Gitlab.**
  - The course staff uses tags in your repo to locate submissions.
  - If we dont’s see a tag $\longrightarrow$ no submission.

Veribile (1/2)

- Unlike VHDL, nice tooling actually exists for SystemVerilog.
- Classroom machines should have Verible[3] pre-installed.
- We are mainly concerned with **autoformatting** and **linting**.

[3]https://github.com/chipsalliance/verible

Verible

- Autoformatting (in place):

```
verible-verilog-format --inplace test_file.sv
```

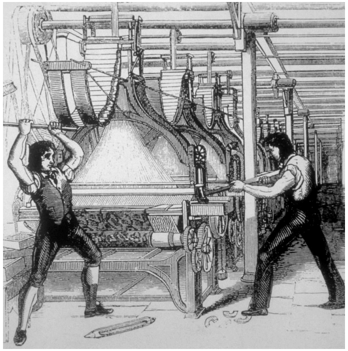
- Linting:

```
verible-verilog-lint test_file.sv
```

- We aim to setup a CI pipeline to automatically run these on student submissions and only accept clean source files.

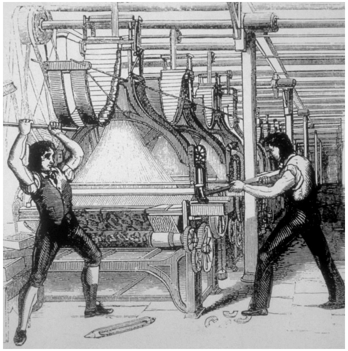
The Elephant in the Room (1/2)

- Using AI tools for the project is not explicitly forbidden, but remember:
  - I will award points for demonstrating **your own** understanding of the topics and questions.
  - Blatant LLM copy-paste answer $\longrightarrow$ 0 pts.
  - I will not help you debug LLM-generated code if even **you** don't know what it's trying to do.



The Elephant in the Room

- **Your own** capacity to reason about and solve problems is what makes you a valuable engineer. Don’t let it atrophy.



Schedule

- The project is under construction as of 18.2.
- We will try to get part 1 released ASAP.
    - Possibly by the 1st-2nd week of March, but we apologise in advance for potential delays.
- We aim for late-enough deadlines to allow for ample time to complete the project.

References

[1] T. Benz **et al.**, “A High-Performance, Energy-Efficient Modular DMA Engine Architecture,” **IEEE Transactions on Computers**, vol. 73, no. 1, pp. 263–277, 2024, doi: 10.1109/TC.2023.3329930.

[2] W. Snyder, P. Wasson, D. Galbi, G. Lore, and et al, “Verilator.” [Online]. Available: https://github.com/verilator/verilator

Thank you!

Questions?