System-on-Chip

Zihao Yu

Institute of Computing Technology
Chinese Academy of Sciences

Introduction

You have implemented SimpleBus in CPU

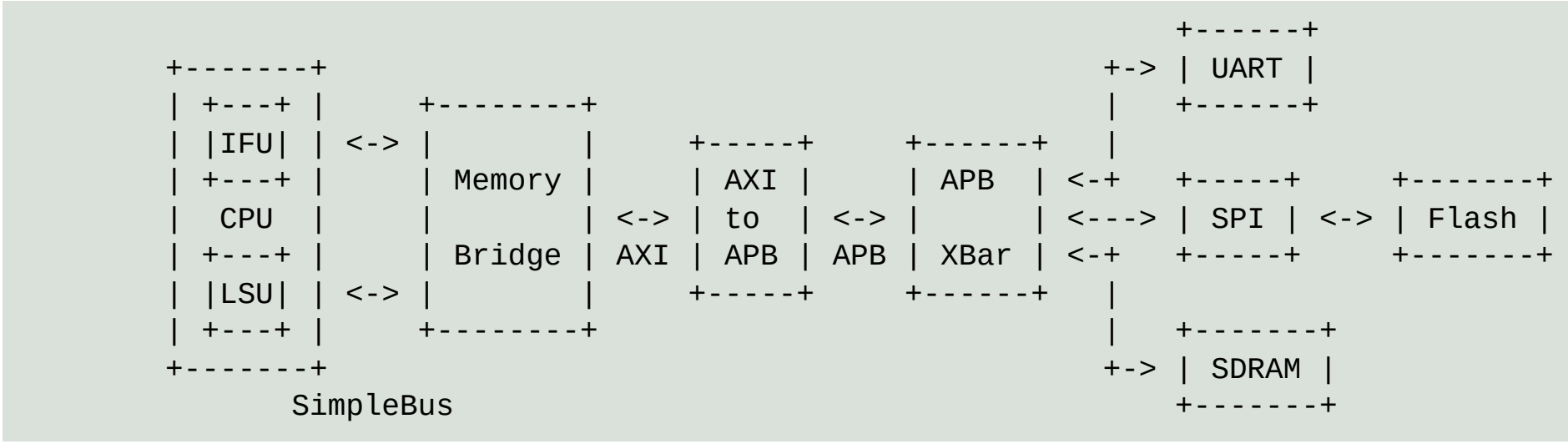
- But devices are still behavior models

With SimpleBus, CPU can connect to SoC

- to communicate with real device controllers in SoC

Connect to SoC

Block Diagram & Address Space



We provide a Memory Bridge to convert from SimpleBus to AXI

- You do not need to know the details of AXI, APB and devices

Device	Address Space
UART16550	`0x1000_0000~0x1000_0fff`
SPI master	`0x1000_1000~0x1000_1fff`
Flash	`0x3000_0000~0x3fff_ffff`
SDRAM	`0x8000_0000~0x81ff_ffff`
Reverse	Other

Connect Your CPU to SoC

TASK 1 - connect your CPU to SoC

1. Clone the repo

```
git clone https://github.com/OSCPU/ysyxSoC
```

2. Adjust the port of the top module according to the table below

- The name should be exactly the same
- No other port is allowed

input		clock	output	[31:0]	io_lsu_addr
input		reset	output	[1:0]	io_lsu_size
output		io_ifu_reqValid	output		io_lsu_wen
output	[31:0]	io_ifu_addr	output	[31:0]	io_lsu_wdata
input		io_ifu_respValid	output	[3:0]	io_lsu_wmask
input	[31:0]	io_ifu_rdata	input		io_lsu_respValid
output		io_lsu_reqValid	input	[31:0]	io_lsu_rdata

Connect Your CPU to SoC(2)

3. `io_lsu_size` is used for accessing devices
   - It is set according to the width of the access:
     - `2'b00` for 1-byte accesses
     - `2'b10` for 4-bytes accesses
4. Modify the reset value of PC to `0x30000000`
   - Let CPU fetch the first instruction from Flash after reset
   - Flash is a kind of non-volatile memory
5. Add all `.v` files under `ysyxSoC/perip` and its subdirectores to the filelist of verilator
6. Add the following directories to the verilator search path for `include`
   - `ysyxSoC/perip/uart16550/rtl`
   - `ysyxSoC/perip/spi/rtl`

Connect Your CPU to SoC(3)

7. Add options `--timescale "1ns/1ns"` and `--no-timing` to verilator
8. Add `ysyxSoC/ready-to-run/D-stage/ysyxSoCFull.v` to the filelist of verilator
9. Modify `ysyx_00000000` in `ysyxSoC/ready-to-run/D-stage/ysyxSoCFull.v` to the module name of your CPU
10. Add the following DPI-C functions to your simulator driver

```
extern "C" void flash_read(int32_t addr, int32_t *data) { assert(0); }
```

10. Compile RTL code to C++ code by verilator
    - If you encounter combinational loop, modify your code
11. Try to start simulation
    - Your CPU should try to fetch instruction from Flash, and should trigger `assert(0)` in `flash_read()`
    - If not, check your implementation of SimpleBus

Run Program on SoC

TASK 2 - run program on SoC

1. Define a 16MB array for Flash, then implement `flash_read()`
   - There is no `flash_write()`, since Flash is not writable by store instructions
   - But the program should not try to write to Flash directly
2. Load `.bin` file from command line to the array of Flash
3. Try to run `ysyxSoC/ready-to-run/D-stage/hello-minirv-ysyxsoc.bin`
   - The program should output `Hello World!`
   - It costs about 1 minute to finish

Run More Programs

From Flash to SDRAM

There are two types of memory in SoC:

- Flash - non-volatile, but not writable by store instructions
  - It can store program before the system is on
  - But it can not support program execution
    - It is reasonable for a program to write to variables
- SDRAM - volatile, and writable by store instructions
  - There is nothing in SDRAM after the system is on
  - It can support program execution

Solution: After the system is on, load the program from Flash to SDRAM

- Then jump to SDRAM to execute the program

Bootloader

We need a loader to load the program from Flash to SDRAM

- It is also called bootloader, since it works at the boot time

But it may be difficult for you to implement a bootloader now

- It requires some knownledge about linking or ELF file structure

In fact, there is a loader inside `ysyxSoC/ready-to-run/D-stage/hello-minirv-ysyxsoc.bin`

- We provide a script to reuse the loader inside this `.bin` file

Bootloader(2)

TASK 3 - run `dummy` on SoC

- Do the following:

```
# First, compile the program to minirv-npc as before
cd am-kernels/tests/cpu-tests
make ARCH=minirv-npc ALL=dummy

# Then, invoke gen.sh with the path to the ELF file
cd ysyxSoC/ready-to-run/D-stage
bash gen.sh am-kernels/tests/cpu-tests/build/dummy-minirv-npc.elf

# Finally, you will get `new.bin` for SoC,
# which will load the program to SDRAM and execute it
ls new.bin
```

- load `new.bin` to Flash and run the simulation
  - Since `dummy` does not output anything, the simulation should end with `a0 == 0`

Modify `putch()`

Now the UART controller is changed to a real one

- You should modify `putch()` in AM to adapt to the real UART controller
- The behavior should be logically the same as the simple behavior model
  - But the address of registers and the bits inside registers may be different

TASK 4 - modify `putch()` to correctly pull the status of UART

- Refer to `ysyxSoC/perip/uart16550/doc/UART_spec.pdf` for more details
- Then try to run `am-kernels/kernels/hello` in the same way as `dummy` above

Control Status Register

Control Status Register

In RISC-V, there is a kind of register to identify the CPU state

- Control Status Register, or CSR

To access CSR, RISC-V provide CSR instruction

- A CSR instruction may actomically read and write a single CSR
- CSR use I-type format
  - There is a 12-bit address space for CSRs, which supports at most 4096 different CSRs
    - But you will only implement very few of them

Refer to RISC-V Privileged Manual for CSR listing

- But you do not need to know every detail of every CSR

Add Your ID

TASK 5 - implement CSRs to encode your ID

- Add `marchid` CSR (read-only), which stores a unique number
  - For example, student ID, or birthday, or something else
  - If your birthday is `19990101`, then store the number `0x1310655` in `marchid`
- Add `mvendorid` CSR (read-only), which stores a unique string no more than 4 characters
  - For example, store `0x6e686f4a` in `mvendorid`, which is the ASCII encoding of `John`
- Implement `csrrs` instruction in CPU
- Write a program to read the above CSR values by `csrrs` instruction in inline assembly, them display them
  - Ask Google or AI for inline assembly

Add Cycle CSR

TASK 6 - implement `mcycle` CSR, a 64-bit read-only counter which will increase `1` every cycle

- In RV32, it is splitted into `mcycle` and `mcycleh`, due to the width of register is 32-bit
  - Refer to the manual for their addresses
- Write a program to read `mcycle` to check whether it is increasing
- Modify `__am_timer_uptime()` by accessing `mcycle` in `abstract-machine/am/src/riscv/npc/timer.c`
  - Divide the value by a constant factor to get the number of microsecond
- Tune this factor to let the test for timer display one line of message every one second
  - Do not worry! We can tune the factor according to the actual frequency after the chip is back

Run more programs

TASK 7 - run more programs on SoC for testing

- `microbench`, Mario, LLaMa, ...

END