CPU Documentation

Contents

1. Overview

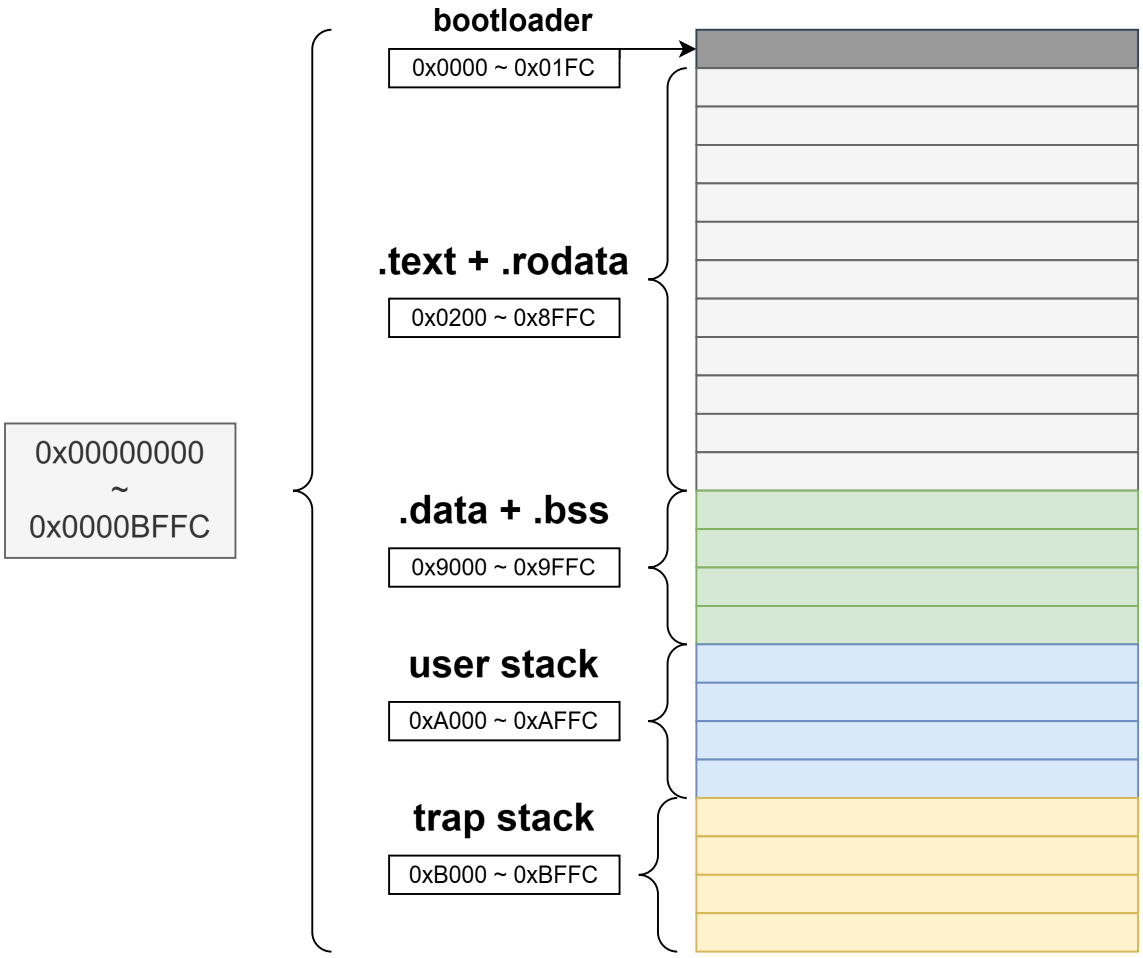
ISA	RISC-V (a subset of `rv32imzicsr` is implemented)
Clocks	A 40MHz clock for the core; a 25.20325 MHz clock for VGA driver
CPI	1.18[1]
Memory Arch	Von Neumann architecture, byte-addressable
Pipelining	5-stage pipelined with forwarding
I/O	MMIO support for VGA, keyboard, UART, 7-seg display, LEDs, buttons, switches
Speculation	Two-level global history predictor, branch target buffer, RAS
Trap	Trap handling mechanism for `ecall` and several exceptions

2. I/O Port Listing

```
module top(
    input clk_100,                  // System clock input (100MHz)
    input reset_n,                  // Asynchronous reset input (active low)

    input uart_rx_in,               // UART receive data input
    output uart_tx_out,             // UART transmit data output

    input [7:0] sws_l,              // Left switches input
    input [7:0] sws_r,              // Right switches input

    input [4:0] bts,                // Buttons input

    output [7:0] leds_l,            // Left LEDs output
    output [7:0] leds_r,            // Right LEDs output

    output [3:0] vga_r,             // VGA red channel output
    output [3:0] vga_g,             // VGA green channel output
    output [3:0] vga_b,             // VGA blue channel output
    output vga_h_sync,              // VGA horizontal sync output
    output vga_v_sync,              // VGA vertical sync output

    output [7:0] tube_ena,          // 7-segment display enable
    output [7:0] left_tube_content, // 7-segment display contents
    output [7:0] right_tube_content,

    input ps2_clk,                  // PS/2 keyboard clock input
    input ps2_data                  // PS/2 keyboard data input
);
```

[1] Assume that load hazard occurs 10% of the time, that control transfer instructions account for 20% of all instructions, that misprediction rate is 20%, and that misprediction penalty is 2 cycles. Then the CPI is $1 + 0.2 * 0.2 * 2 + 0.1 * 1 = 1.18$

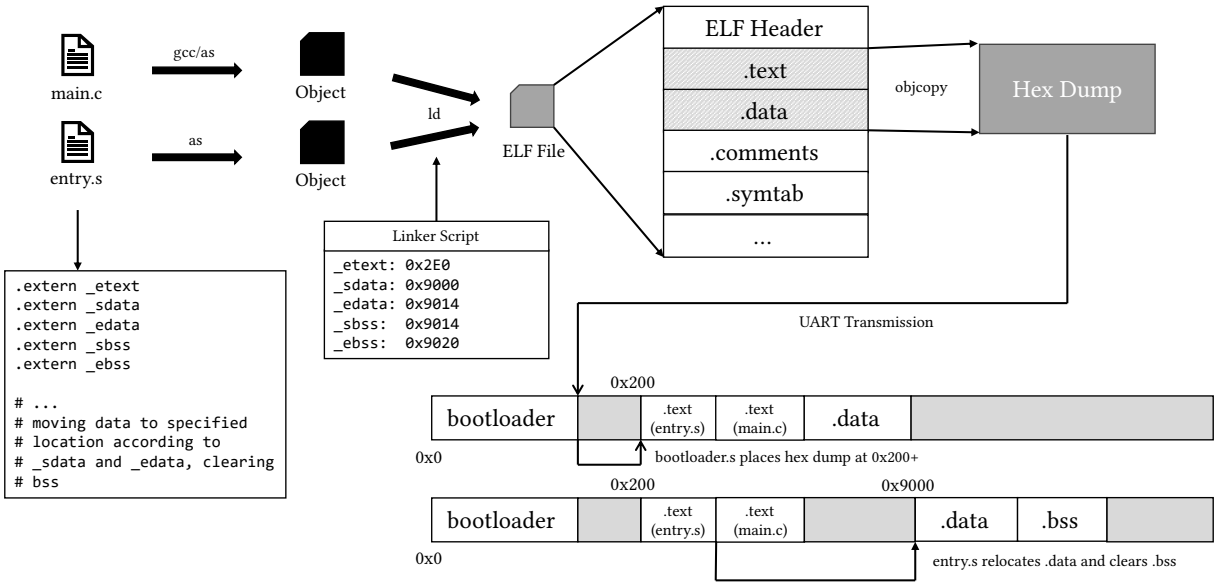
3. Memory Layout



- Memory-mapped I/O layout:

LEDs	`0x8020_0000` [15:0]
Switches	`0x8030_0000` [15:0]
VGA	`0x8040_0000` ~ `0x8042_57FC`
Keyboard	`0x8050_0000` [7:0]
Buttons	`0x8060_0000` [4:0] (up, down, left, right, center)
7-seg dis	`0x8070_0000` [31:0]
UART	`0x8080_0000` [7:0] for data register (write to send, read to receive) `0x8080_0004` [1:0] for status register (bit[0] for `rx_ready`, bit[1] for `tx_ready`) `0x8080_0008` [31:0] for control register (equal to `CLK_MAIN_FREQ` / `UART_FREQ`)

4. C Program Loading & Execution Flow



Our CPU is able to run bare-metal C programs elegantly. In the `program` folder of our Github repository, you will find the starter code, trap handler code, library drivers and example user programs we provided. You can freely change the user program that you want to run on our CPU by modifying `main.c`, and run `make` to compile them and generate ASCII texts to be loaded to the CPU via UART.

The exact building & loading procedure is as follows.

PC Side:

Step 1: User programs (e.g., `main.c`), along with the starter code (`entry.s`), are compiled into objects.

Step 2: Objects are linked according to the linker script and form the ELF file. The linker script specifies where each section of the program is located (in particular, `entry.s` is always put at the first in `.text`) and provides symbols including `_etext`, `_sdata`, `_edata` etc. These symbols will be referenced in `start.s`.

Step 3: We use `objcopy` from GNU toolchain to extract the `.text` and `.data` sections in the ELF file, convert them to ASCII texts and send them to FPGA via UART.

FPGA Side:

Step 1: Once powered on, the CPU starts executing the bootloader code, which is pre-loaded to memory at address `0x000` ~ `0x1FF` via `.coe` file. The bootloader is essentially a big `while` loop that constantly checks if there are any data received from the UART interface. Every time a data word is available, it moves the data sequentially to address starting at `0x200`.

Step 2: When the bootloader has not detected data from UART for a certain amount of time since the last data reception, it decides that the entire program has been received and jumps to `0x200` to execute `entry.s`.

Step 3: `entry.s` relocates `.data` section and clears up a space in memory for `.bss` section based on the symbols provided by the linker. It also initializes key registers including `sp` and `mtvec` and saves information for trap handling in the trapframe.

Step 4: After `entry.s` is finished, it calls `main` function, starting the execution of user programs.

New Program / New RTL Design ?

- If we want to execute a new program, just push the reset button and the program counter will be set to 0 to re-execute the bootloader. And then we can send the new program to FPGA via UART.
- If we want to modify the RTL design or the bootloader itself, then we will have to rerun synthesis, implementation and send the new bitstream to FPGA.

5. Design Choices: HW/SW Comparison

In this section, I will show you some design choices we made that reflect the strengths and weaknesses of hardware and software.

5.1. UART Program Loading

- Initially, we had a hardware-focused program loading method. We let the CPU run in 2 modes. Once powered on, the CPU is in `LOADING` mode, and `uart_handler` module would constantly move the 32-bit instruction received from the UART port sequentially to the BRAM through port A. If no new instructions are received for a certain amount of time, the CPU switches to `RUNNING` mode, and starts fetching and executing instructions according to the program counter.
  Flaws of this approach include:
  - **Limited UART functionality.** The way how data coming from UART is moved to memory is hardwired in RTL (taking up the write port of port A) and is only possible in `LOADING` mode. Once the CPU enters `RUNNING` mode, it would be hard to use UART to send or receive messages from PC.
  - **Hard to reconfigure.** It can be difficult to reconfigure certain parameters, such as UART frequency, during runtime.
- In our current approach, the moving of data is done with the help of 3 registers as memory-mapped I/O. Benefits of the current approach include:
  - **Enhanced flexibility and control.** By writing to / reading from the data register, and checking the status register for send / receive availability, we can freely send / receive data from the PC. Also, writing to the control register changes the UART frequency.

5.2. Storage of ASCII Characters

- Initially, we had a dedicated piece of memory in the address space that holds a character set of $8 * 8$ ASCII characters. It is hard-coded in the RTL design. We need to access them by their specific address. This is highly inflexible: you cannot expand or reduce the character set without modifying the RTL design.
- With our new program loading & execution toolchain, we can define the character set in any of our C programs as global variables. The linker will put them in the `.rodata` or `.data` section and resolve all references to the character set. Then, the starter code will relocate those sections to the correct location.

6. Test Cases

Test cases are categorized into those that can only be run in simulation and those that can be run in both simulation and on actual FPGA board.

Test File/Name	Test Type	Test Method	Result
bram_tb.v	Unit Test	In simulation only	Passed

cpu_arith_test.v	Integration Test	Both	Passed
cpu_ascii_test.v	Integration Test	Both	Passed
cpu_branch_prediction_test.v	Integration Test	Both	Passed
cpu_branch_test.v	Integration Test	Both	Passed
cpu_ctrl_hazard_test.v	Integration Test	Both	Passed
cpu_data_hazard_test.v	Integration Test	Both	Passed
cpu_jump_test.v	Integration Test	Both	Passed
cpu_quicksort_test.v	Integration Test	Both	Passed
cpu_store_load_test.v	Integration Test	Both	Passed
cpu_vga_test.v	Integration Test	Both	Passed
imm_gen_tb.v	Unit Test	In simulation only	Passed
keyboard_tb.v	Unit Test	In simulation only	Passed
memory_tb.v	Unit Test	In simulation only	Passed
ring_test.v	Unit Test	In simulation only	Passed
top_tb.v	Integration Test	In simulation only	Passed
uart_load_tb.v	Unit Test	In simulation only	Passed
uart_tb.v	Unit Test	In simulation only	Passed
hazard.asm	Integration Test	Both	Passed
scene1.asm	Integration Test	Both	Passed
scene2.asm	Integration Test	Both	Passed
test_arith.asm	Integration Test	Both	Passed
test_branch_prediction.asm	Integration Test	Both	Passed
test_branch.asm	Integration Test	Both	Passed
test_data_hazard.asm	Integration Test	Both	Passed
test_deep_recursion.asm	Integration Test	Both	Passed
test_flash_display.asm	Integration Test	Both	Passed
main.c (multiplication tests, etc.)	Integration Test	Both	Passed

Conclusion: Everything's OK, both in simulation and on FPGA board.

7. Instruction Listing

RV32I Base Integer Instructions[2]	
add	rd = rs1 + rs2
sub	rd = rs1 − rs2
xor	rd = rs1 ^ rs2
or	rd = rs1 | rs2
and	rd = rs1 & rs2
sll	rd = rs1 ≪ rs2
srl	rd = rs1 ≫ rs2

[2] `ebreak` in `rv32i` is not implemented.

sra	rd = rs1 ≫ rs2 (msb-extends)
slt	rd = (rs1 < rs2)? 1 : 0
sltu	rd = ((unsigned)rs1 < (unsigned)rs2)? 1 : 0 (zero-extends)
addi	rd = rs1 + imm
xori	rd = rs1 ^ imm
ori	rd = rs1 | imm
andi	rd = rs1 & imm
slli	rd = rs1 ≪ imm[0:4]
srli	rd = rs1 ≫ imm[0:4]
srai	rd = rs1 ≫ imm[0:4] msb-extends
slti	rd = (rs1 < imm)? 1 : 0
sltiu	rd = ((unsigned)rs1 < (unsigned)imm)? 1 : 0 (zero-extends)
lb	rd = M[rs1 + imm][0:7]
lh	rd = M[rs1 + imm][0:15]
lw	rd = M[rs1 + imm][0:31]
lbu	rd = M[rs1 + imm][0:7] (zero-extends)
lhu	rd = M[rs1 + imm][0:15] (zero-extends)
sb	M[rs1 + imm][0:7] = rs2[0:7]
sh	M[rs1 + imm][0:15] = rs2[0:15]
sw	M[rs1 + imm][0:31] = rs2[0:31]
beq	if (rs1 == rs2) PC += imm
bne	if (rs1 ≠ rs2) PC += imm
blt	if (rs1 < rs2) PC += imm
bge	if (rs1 ≥ rs2) PC += imm
bltu	if ((unsigned)rs1 < (unsigned)rs2) PC += imm (zero-extends)
bgeu	if ((unsigned)rs1 ≥ (unsigned)rs2) PC += imm (zero-extends)
jal	rd = PC + 4; PC += imm
jalr	rd = PC + 4; PC = rs1 + imm
lui	rd = imm ≪ 12
auipc	rd = PC + (imm ≪ 12)
ecall	Jump to trap handler
RV32M Multiply Extension[3]	
mul	rd = (rs1 ∗ rs2)[31:0]
mulh	rd = (rs1 ∗ rs2)[63:32]
mulsu	rd = (rs1 ∗ (unsigned)rs2)[63:32]
mulu	rd = ((unsigned)rs1 ∗ (unsigned)rs2)[63:32]
RV32 Zicsr Extension for CSR Instructions	
csrrw	rd = csr; csr = rs1

[3] `div` , `divu` , `rem` , `remu` in `rv32m` are not implemented due to timing limitations: it seems unlikely that these operations can be completed within the duration of EX stage, which lasts for only 1 cycle (25 ns).

csrrs	rd = csr; csr = csr & rs1
csrrc	rd = csr; csr = csr | ~rs1
csrrwi	rd = csr; csr = imm[0:4] (zero-extends)
csrrsi	rd = csr; csr = csr & imm[0:4] (zero-extends)
csrrci	rd = csr; csr = csr | ~imm[0:4] (zero-extends)
Miscellaneous	
mret	Return from trap

8. Register Listing

General Purpose Registers	
zero	Zero constant
ra	Return address
sp	Stack pointer
gp	Global pointer
tp	Thread pointer
t0~t6	Temporary registers
s0~s11	Saved registers
a0~a7	Function arguments / return values
Control & Status Registers	
mscratch	the address of the trapframe
mepc	the address of the trapped instruction
mcause	the cause for the last trap
mtvec	the address of the trap handler
mboot	a customized CSR register at address `0x7C0` that indicates whether user code or boot code is currently running
mtval	These registers are implemented in the CSR file merely as placeholders, and are largely unused in our design.
mie	
mip	
mstatus	

9. Bonuses

9.1. Pipelining

Our CPU is 5-stage pipelined (IF, ID, EX, MEM, WB). Registers are placed between stages to transfer data / control signals. Hazards are resolved by forwarding and stalling mechanism.

9.1.1. Data Hazards

Module `hazard_unit` is used for forwarding data to EX stage when needed and sending flush / stall signals to stage transfer registers, so as to resolve data hazards.

```
module hazard_unit(
    input [4:0] MEM_rd,
    input [31:0] MEM_reg_w_data,
```

```
  input MEM_reg_w_en,
  input [4:0] WB_rd,
  input [31:0] WB_reg_w_data,
  input WB_reg_w_en,
  input [2:0] EX_csr_op,
  input [4:0] EX_rs1,
  input [4:0] EX_rs2,
  input EX_ecall,
  input EX_mret,
  input [11:0] EX_csr_addr,
  input [11:0] MEM_csr_addr,
  input [11:0] WB_csr_addr,
  input [31:0] MEM_csr_w_data,
  input [31:0] WB_csr_w_data,
  input MEM_csr_w_en,
  input WB_csr_w_en,
  output [31:0] MEM_reg_w_data_forwarded,
  output [31:0] WB_reg_w_data_forwarded,
  output [31:0] MEM_csr_w_data_forwarded,
  output [31:0] WB_csr_w_data_forwarded,
  output [1:0] forward_rs1_sel,
  output [1:0] forward_rs2_sel,
  output [1:0] forward_csr_sel,
  input [4:0] ID_rs1,
  input [4:0] ID_rs2,
  input [4:0] EX_rd,
  input EX_load,
  output load_stall,
  output load_flush
  );
```

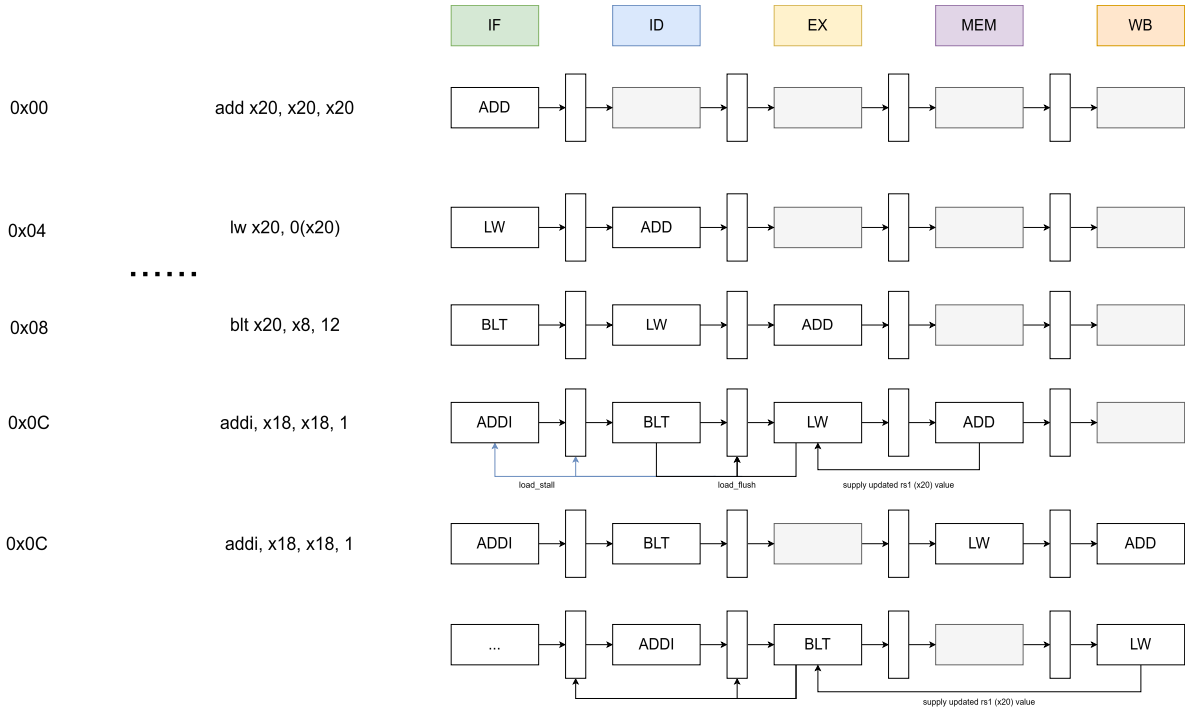
- `forward_rs1_sel` decides:
  - Whether the data of `rs1` needs to be replaced with forwarded data
  - Where exactly should the forwarded data come from (previous stage / double previous stage)

```
 // when to forward? the same register (MEM_rd == EX_rs1) is previously written to
(MEM_reg_w_en) and currently read (EX_rs1 != 5'b0)
 assign forward_rs1_sel =
   (MEM_reg_w_en & (EX_rs1 != 5'b0) & (MEM_rd == EX_rs1)) ? `FORWARD_PREV :
   (WB_reg_w_en  & (EX_rs1 != 5'b0) & (WB_rd  == EX_rs1)) ? `FORWARD_PREV_PREV :
   `FORWARD_NONE;
```

- `load_stall` and `load_flush` are generated when a load data hazard is detected. The pipeline must be stalled for 1 cycle to resolve the hazard.

```
    assign load_stall =
      (EX_load & (ID_rs1 != 5'b0) & (EX_rd == ID_rs1)) |
      (EX_load & (ID_rs2 != 5'b0) & (EX_rd == ID_rs2));
    assign load_flush =
      (EX_load & (ID_rs1 != 5'b0) & (EX_rd == ID_rs1)) |
      (EX_load & (ID_rs2 != 5'b0) & (EX_rd == ID_rs2));
```

  The following image illustrates how we resolve a **non-load data hazard** and a **load data hazard**:

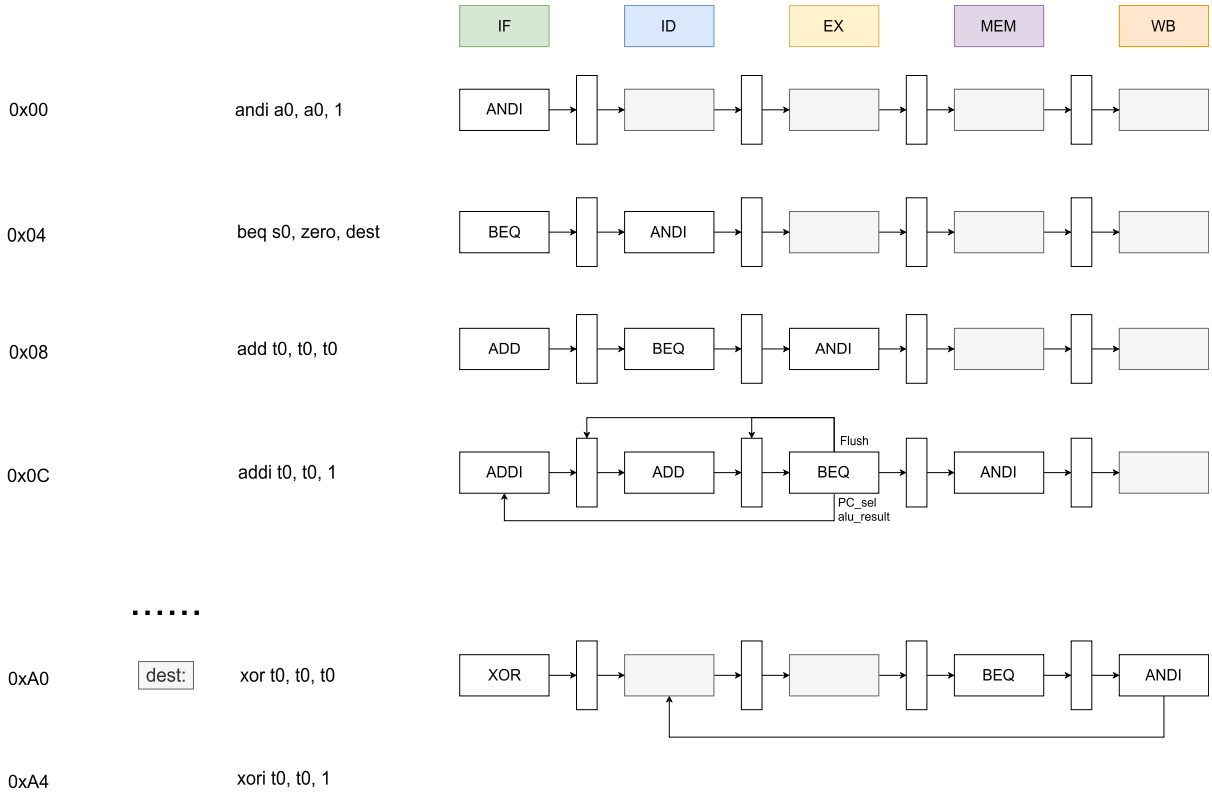


- When `lw x20, 0(x20)` enters `EX` stage, it is detected that its previous instruction `add x20, x20, x20` tries to write to `x20` with result from `ALU`. This constitutes a non-load data hazard.
  This is solved directly by forwarding: the hazard unit forwards the `ALU` result to `EX` stage for `lw` to use.
- When `blt x20, x8, x12` enters `ID` stage, it is detected that `lw x20, 0(x20)`, which is executed just before it, tries to load new data to `x20`, which is needed by `blt`. This constitutes a load data hazard.
  This requires a stall to resolve. The hazard unit then sends a stall signal to `IF` and `IF/ID`, making `IF` and `ID` retain their original value; it also sends a flush to `ID/EX`, effective creating a bubble between `blt` and `lw`.
  1 cycle later, `blt` finally enters `EX` stage, and now the hazard unit can forward the loaded data to `EX` and let `blt` do its job.

- `CSR` registers require dedicated data path for forwarding. For further information, please refer to the code.

9.1.2. Control Hazards

In this section, we will show you how control hazards are resolved **without branch prediction**. Since we always assume that branch is taken, all we do to resolve control hazard is to send a flush signal to `IF/ID` and `ID/EX` to eliminate the mis-fetched instructions, whenever `pc_sel` in `EX` is detected. As shown in the image below:



9.2. Branch Predictor

Our CPU employs a sophisticated branch prediction unit (`branch_prediction_unit.v`) to mitigate control hazards and improve pipeline efficiency. Here's how it works:

1. Prediction Mechanisms:

The `branch_prediction_unit` uses a combination of techniques:

- **Branch Prediction Table (BPT) / Pattern History Table (PHT):**
  - A table (`prediction_table`) of 2-bit saturating counters (states: `STRONG_NOT_TAKEN` , `WEAK_NOT_TAKEN` , `WEAK_TAKEN` , `STRONG_TAKEN`).
  - It predicts the **direction** of conditional branches (taken or not-taken).
  - **Indexing:** The BPT is indexed by `IF_prediction_index` . This index is formed by concatenating the lower bits of the current instruction's PC (`IF_pc[2+:PREDICTOR_DEPTH_LOG-BRANCH_HISTORY_SELECTED_BITS]`) with a portion of the global `branch_history` register (`branch_history[0+:BRANCH_HISTORY_SELECTED_BITS]`). This is a common scheme (similar to gshare) that correlates branch behavior with both its address and recent global branch outcomes.
- **Branch Target Buffer (BTB):**
  - A table (`branch_target_buffer`) that stores the **predicted target addresses** for branches.
  - **Indexing:** The BTB is indexed using only the lower bits of the instruction's PC (`IF_pc[2+:PREDICTOR_DEPTH_LOG]`). The target of a specific branch instruction is generally fixed, so global history isn't typically used for BTB indexing.
- **Return Address Stack (RAS):**
  - Implemented using the `ring` module.
  - Specifically predicts the target addresses for `jalr` instructions that are function returns.
  - **Operation:**

  - When a `call` instruction (identified by `IF_call` based on `jal`/`jalr` opcode and `rd`/`rs1` register conventions, e.g., `rd` is `x1` or `x5`) is fetched, the predicted return address (`IF_pc + 4`) is pushed onto the RAS.
  - When a `return` instruction (identified by `IF_return` based on `jalr` and `rs1` being `x1` or `x5`) is fetched, the address at the top of the RAS (`RAS_out`) is used as the predicted target.
- **Global Branch History Register (BHR):**
  - A shift register (`branch_history`) that records the actual outcomes (taken/not-taken) of the most recent branches.
  - It's updated in the EX stage with `EX_pc_sel` (the actual outcome of the branch).

2. Prediction in the IF Stage:

When an instruction is fetched in the IF stage:

- **Direction Prediction (`branch_predict` output of BPU):**
  - Unconditional jumps (`jal`, `jalr`) are always predicted taken (`1'b1`).
  - For conditional branches (`IF_B`), the BPT entry corresponding to `IF_prediction_index` is consulted. If the 2-bit counter is `WEAK_TAKEN` or `STRONG_TAKEN`, the branch is predicted taken. Otherwise, it's predicted not-taken.
- **Target Prediction (`branch_target` output of BPU):**
  - If the instruction is identified as a `return` (`IF_return`), the `RAS_out` is used as the target.
  - Otherwise (for other branches and jumps), the `branch_target_buffer` entry indexed by `IF_pc[2+:PREDICTOR_DEPTH_LOG]` is used.
- The IF stage (IF.v) uses these `branch_predict` and `branch_target` signals to speculatively fetch the next instruction. The `IF_branch_predict` signal (the prediction made for the current IF instruction) is passed down the pipeline via the `IF_ID` and `ID_EX` registers.

3. Verification and Update in the EX Stage:

When a branch instruction reaches the EX stage, its actual outcome and target are calculated:

- **Misprediction Detection:**
  - **`EX_false_direction`:** The BPU compares the prediction made in IF (now `EX_branch_predict`, latched from `ID_branch_predict`) with the actual outcome (`EX_pc_sel`). If they differ, a direction misprediction occurred.
  - **`EX_false_target`:** If the branch was predicted taken, the BPU compares the predicted target (latched as `branch_target_prev_2` or `EX_RAS_out` for returns) with the actual calculated target (`EX_alu_result`). If they differ, a target misprediction occurred.
- **Pipeline Flush (`EX_branch_flush`):**
  - If either `EX_false_direction` is true, or if `EX_false_target` is true **and** the branch was predicted taken, `EX_branch_flush` is asserted.
  - In `core.v`, this signal, along with `EX_out_excp` or `EX_out_mret`, resets the `IF_ID` and `ID_EX` pipeline registers, squashing incorrectly fetched instructions. The `IF` stage is also redirected by `EX_trap_dest` or corrected branch outcomes/targets.
- **BPT Update:**
  - The 2-bit counter in `prediction_table` at `prediction_index_update` (which is the `IF_prediction_index` latched from when the instruction was in IF) is updated based on the actual outcome (`EX_pc_sel`). It's incremented if taken (towards `STRONG_TAKEN`) and decremented if not-taken (towards `STRONG_NOT_TAKEN`). Updates are skipped for calls and returns to preserve entries for other branches.
- **BTB Update:**

- If the instruction in EX is a branch or jump (but not a return), the `branch_target_buffer` entry at `EX_pc[2+:PREDICTOR_DEPTH_LOG]` is updated with the actual target address (`EX_alu_result`).
- **BHR Update:**
  - The `branch_history` register is shifted, and the new bit is the actual outcome (`EX_pc_sel`) of the branch in the EX stage.
- **RAS Correction:**
  - If `EX_branch_flush` is asserted due to a misprediction:
    - If a `call` was mispredicted (`ID_call` was true), the RAS performs a `RING_POP` to undo the speculative push.
    - If a `return` was mispredicted (`ID_return` was true), the RAS performs a `RING_CANCEL_POP` to effectively undo the speculative pop (by adjusting its internal pointers).

4. Integration in `core.v`:

- The `branch_prediction_unit_0` is instantiated in `core.v`.
- The IF stage receives `branch_predict` and `branch_target` to guide fetching.
- Control signals like `EX_branch_flush`, `EX_false_target`, and `EX_false_direction` from the BPU, along with exception/mret signals from the EX stage, are used to manage pipeline stalls and flushes.
- The prediction made in IF (`IF_branch_predict`) is passed down through `ID_in_branch_predict` (to `IF_ID`), `ID_out_branch_predict` (from `ID`), `EX_in_branch_predict` (to `ID_EX`), and finally `EX_branch_predict` (from `ID_EX`, which is an input to the BPU for verification).

5. Determining `pc_next`

With the addition of the branch predictor, the `pc_next` needs to be carefully chosen. We have to first recover from previous misprediction, if there is any, and then predict the current branch.

- First, we check if there are any exceptions or a `mret` detected in `EX` stage. If so, jump to trap destination.
- Second, we check if we have previously jumped to a false target. If so,
  - if we shouldn't have jumped in the first place, go to `EX_pc_plus_4`.
  - if we should jump but the target was wrong, go to `EX_alu_result`, which is the correct target.
- Third, we check if we have mispredicted the branch. If so,
  - if we jumped, we can recover by going to `EX_pc_plus_4`.
  - if we didn't jump, we can recover by going to `EX_alu_result`.
- Fourth, we check the current branch prediction result coming from the branch prediction unit and jump accordingly.

```
`ifdef BRANCH_PREDICT_ENA
    wire [31:0] pc_next = (EX_excp | EX_mret) ? EX_trap_dest :
            EX_false_target ? (EX_pc_sel ? EX_alu_result : EX_pc_plus_4) :
            EX_false_direction ? (EX_branch_predict ? EX_pc_plus_4 : EX_alu_result) :
            branch_predict ? branch_target :
            (pc + 4);
`else
    wire [31:0] pc_next = (EX_excp | EX_mret) ? EX_trap_dest :
            EX_pc_sel ? EX_alu_result : (pc + 4);
`endif
```

9.3. CSR instructions

We added a `CSR` file in `ID`, alongside general purpose register file, and added all 6 instructions in the `Zicsr` RISC-V extension.

```
csr csr_0(
    .clk            (clk),
    .reset          (reset),
    .csr_w_en       (WB_csr_w_en),
    .csr_w_data     (WB_csr_w_data),
    .csr_w_addr     (WB_csr_addr),
    .csr_r_addr     (csr_addr),
    .csr_r_data     (csr_r_data),

    .w_mstatus      (WB_w_mstatus),
    .w_mie          (`CSR_NO_WRITE),
    .w_mtvec        (`CSR_NO_WRITE),
    .w_mscratch     (`CSR_NO_WRITE),
    .w_mepc         (WB_w_mepc),
    .w_mcause       (WB_w_mcause),
    .w_mtval        (`CSR_NO_WRITE),
    .w_mip          (`CSR_NO_WRITE),
    .w_mboot        (`CSR_NO_WRITE),

    .r_mstatus      (),
    .r_mie          (),
    .r_mtvec        (ID_mtvec),
    .r_mscratch     (),
    .r_mepc         (ID_mepc),
    .r_mcause       (),
    .r_mtval        (),
    .r_mip          (),
    .r_mboot        (ID_mboot)
);
```

The `CSR` instructions are decoded in `ID` and executed just like other instructions.

```
always @(*) begin
    case (ID_csr_op)
        `CSRRW:     EX_csr_w_data = fwd_rs1_data;
        `CSRRS:     EX_csr_w_data = fwd_csr_data | fwd_rs1_data;
        `CSRRC:     EX_csr_w_data = fwd_csr_data & ~fwd_rs1_data;
        // ID_rs1 is treated as 5-bit immediate and
        // zero-extended in below instructions
        `CSRRWI:    EX_csr_w_data = {27'b0, ID_rs1};
        `CSRRSI:    EX_csr_w_data = fwd_csr_data | {27'b0, ID_rs1};
        `CSRRCI:    EX_csr_w_data = fwd_csr_data & ~{27'b0, ID_rs1};
        default:    EX_csr_w_data = 32'h0;
    endcase
end
```

9.4. Trap Handling

Trap handling in our CPU design is a coordinated effort between hardware and software to manage exceptions (like illegal memory access or `ecall` instructions).

Here's how it works:

1. Hardware Detection and Initial Response (Primarily in EX Stage):

- **Exception Detection (`EX.v`):**
  - The Execution (EX) stage is primarily responsible for detecting synchronous exceptions.
  - `INST_ACCESS_FAULT` : If the program counter (`ID_pc`) points outside the allowed code region (`S_TEXT` to `S_DATA`) after the bootloader phase (`ID_mboot == 32'h0`).
  - `LOAD_ACCESS_FAULT` : If a load instruction tries to read from an invalid memory region (e.g., below `S_TEXT` when not in bootloader).
  - `STORE_ACCESS_FAULT` : If a store instruction tries to write to an invalid memory region (e.g., below `S_DATA` when not in bootloader).
  - `ECALL_M` : When an `ecall` instruction (decoded in ID, signaled by `ID_ecall`) reaches the EX stage.
  - An internal `excp_code` is set based on the type of exception.
- **CSR Updates (`EX.v`):**
  - `EX_w_mcause` : The `mcause` (Machine Cause) CSR is set to `excp_code` if an exception (`EX_excp`) occurs.
  - `EX_w_mepc` : The `mepc` (Machine Exception Program Counter) CSR is set to the address of the faulting instruction (`ID_pc`) if an exception occurs.
  - `EX_w_mstatus` : The `mstatus` (Machine Status) CSR is updated. For an `ecall` , interrupts are typically disabled. For an `mret` , the previous interrupt enable state is restored.
- **Pipeline Control (`core.v` , `EX.v`):**
  - If an exception (`EX_excp`) or `mret` (`EX_out_mret`) occurs, earlier pipeline stages (IF_ID, ID_EX) are flushed/reset to prevent incorrect instructions from proceeding.
  - For memory access faults, the write enable (`EX_reg_w_en`) or store/load width signals (`EX_store_width` , `EX_load_width`) might be overridden to prevent the faulty memory operation from completing.
- **Trap Handler Address Calculation (`EX.v`):**
  - `EX_trap_dest` determines the next PC:
    - For an `ecall` or other exceptions: It's the value of the `mtvec` (Machine Trap Vector) CSR (potentially forwarded if recently written).
    - For an `mret` instruction: It's the value of the `mepc` CSR (potentially forwarded).
- **PC Redirection (`IF.v`):**
  - The `EX_trap_dest` is fed to the Instruction Fetch (IF) stage. If an exception or `mret` is active, the IF stage will fetch the next instruction from `EX_trap_dest` instead of `pc_plus_4` or a branch target.

2. Software Setup (Boot Time - `entry.s`):

- **`mtvec` Initialization:** The `mtvec` CSR is loaded with the address of the `m_trap` assembly label (defined in `trap.s`). This is the common entry point for all machine-mode traps.
- **`mscratch` Initialization:** The `mscratch` CSR is loaded with the address of the `trapframe` global variable (defined in `trap_handler.c` and `trap_handler.h`). This `trapframe` structure is used to save and restore the processor's context.
- **Trapframe Pointers:** The first two words of the `trapframe` are initialized:
  - `trapframe.trap_sp` : Stores the top of a dedicated trap stack.
  - `trapframe.trap_handler` : Stores the address of the C function `m_trap_handler` .

3. Software Trap Entry (Assembly - `trap.s` `m_trap` label):

- **Context Save:**
  - `csrrw a0, mscratch, a0` : Atomically swaps the contents of register `a0` and the `mscratch` CSR.
    - `a0` now holds the address of the `trapframe` (which was set up in `mscratch`).

    - `mscratch` now holds the original value of `a0` from the trapped user code.
  - The assembly code then saves most of the general-purpose registers (ra, sp, gp, tp, t0-t2, s0-s1, a1-a7, s2-s11, t3-t6) into the `trapframe` structure, using `a0` as the base pointer. The original `a0` (now in `mscratch`) is saved specifically to `trapframe.a0`.
- **Stack Switch:** The stack pointer `sp` is set to `trapframe.trap_sp` (the dedicated trap stack).
- **Jump to C Handler:** The address of the C trap handler (`m_trap_handler`, previously stored in `trapframe.trap_handler`) is loaded, and a `jalr` instruction calls it.

4. Software Trap Handling (C - `trap_handler.c`):

- **`m_trap_handler()` function:**
  - `trapframe.user_pc = r_mepc();`: Saves the current `mepc` (address of the trapped instruction) into the `trapframe`.
  - `uint32_t cause = r_mcause();`: Reads the `mcause` CSR to determine the reason for the trap.
  - A `switch` statement handles different `cause` values:
    - `ECALL_M`:
      - `trapframe.user_pc += 4;`: Increments the saved PC to point to the instruction **after** the `ecall` so that execution resumes there upon return.
      - Calls the `ecall()` C function.
    - `INST_ACCESS_FAULT`, `STORE_ACCESS_FAULT`, `LOAD_ACCESS_FAULT`:
      - Prints diagnostic information (cause, mepc) to the UART.
      - Enters an infinite loop (`while(1);`), effectively halting the CPU on these critical errors.
  - Calls `m_trap_done()` to prepare for returning from the trap.
- **`ecall()` function:**
  - This function acts as a simple operating system service call dispatcher.
  - It reads `trapframe.a7` (which, by RISC-V convention, holds the ecall/syscall number).
  - A `switch` on `a7` provides different services:
    - `case 0`: Prints a message "I'm an ecall!".
    - `case 1`: Enters an infinite loop.
    - `case 2`: Prints the entire content of the `trapframe` (all saved registers) to UART for debugging purposes. It then waits for a button press to continue.

5. Software Trap Return Preparation (C - `trap_handler.c`):

- **`m_trap_done()` function:**
  - `w_mepc(trapframe.user_pc);`: Writes the (potentially updated) `user_pc` from the `trapframe` back to the `mepc` CSR. This is the address where execution will resume after `mret`.
  - `m_ret((uint32_t)(&trapframe));`: Calls the assembly function `m_ret` (in `trap.s`), passing the address of the `trapframe` in `a0`.

6. Software Trap Return (Assembly - `trap.s` `m_ret` label):

- **Context Restore:**
  - `a0` (argument from C) holds the `trapframe` address.
  - The original user `a0` is loaded from `trapframe.a0` and placed into `mscratch`.
  - All other saved registers are restored from the `trapframe`.
  - `csrrw a0, mscratch, a0`: Swaps `a0` and `mscratch` again.
    - `a0` now holds its original pre-trap value.
    - `mscratch` holds the `trapframe` address (this value is not critical for the `mret` itself).
- **`mret` Instruction:** The `mret` (Machine Return) instruction is executed. The hardware then:
  - Sets the PC to the value currently in `mepc`.

This comprehensive process ensures that when a trap occurs, the CPU's state is saved, the appropriate handler is invoked, and the state can be correctly restored to resume normal execution or handle the error.

9.5. UART

The UART (Universal Asynchronous Receiver/Transmitter) module in our CPU design enables serial communication. It's implemented in Verilog and controlled by the CPU via memory-mapped I/O.

9.5.1. Hardware Implementation

- **Main Module (`uart.v`):**

```
module uart(
    input clk,
    input reset,

    input read,
    input rx_in,
    output [7:0] rx_out,
    output reg rx_ready,

    input write,
    input [7:0] tx_in,
    output tx_out,
    output reg tx_ready,

    input [31:0] ctrl
);
```

  - Located at `uart.v`.
  - It instantiates two sub-modules:
    - `uart_rx.v`: Handles receiving serial data. It detects start/stop bits and assembles 8 data bits.
    - `uart_tx.v`: Handles transmitting serial data. It takes an 8-bit byte and sends it serially with start/stop bits.
  - **Baud Rate Generation:** The baud rate is determined by a 32-bit control input `ctrl`. This value acts as a divisor for the system clock (`clk`) to generate enable signals (`rx_en`, `tx_en`) at the desired baud rate for the `uart_rx` and `uart_tx` modules. The `ctrl` value is typically calculated as `(SystemClockFrequency / BaudRate)`.
  - **CPU Interface:** It communicates with the CPU using signals like `read` (CPU acknowledges received data), `write` (CPU initiates data transmission), `tx_in` (data byte from CPU), `rx_out` (data byte to CPU), `rx_ready` (indicates new data available), and `tx_ready` (indicates transmitter is ready for new data).
- **Integration (`memory.v`, `core.v`, `top.v`):**
  - The UART is memory-mapped, with its registers accessible at a base address defined by `ADDR_MMIO_UART` (`0x80800000` from `params.v`).
  - The `memory.v` module decodes memory accesses:
    - **Data Register (`UART_DATA_REG`, offset 0x0 from base):**
      - Writing to this address sends the byte out via UART. The `memory` module asserts `uart_write` and passes the data to the `uart` module's `tx_in`.
      - Reading from this address retrieves a received byte from the `uart` module's `rx_out`. This also clears `rx_ready`.
    - **Status Register (`UART_STATUS_REG`, offset 0x4 from base):**
      - Reading this provides:

    - Bit 0 (`UART_RX_RDY`): High if a byte is received and ready.
    - Bit 1 (`UART_TX_RDY`): High if the UART is ready to transmit another byte.
  - **Control Register (`UART_CONTROL_REG` , offset 0x8 from base):**
    - Writing to this register sets the baud rate divisor (`uart.ctrl`).
    - Reading from this register returns the current divisor.
    - It's initialized on reset in `memory.v` using `CLK_MAIN_FREQ` / `UART_FREQ` from `params.v` .
- The `core.v` module instantiates the `memory` module.
- The `top.v` module instantiates the `uart` module and connects its serial lines (`rx_in` , `tx_out`) to FPGA pins. It also routes the MMIO interface signals through the `core` to the `uart` module.

9.5.2. Software Control

The C library functions in `program/lib/uart/` provide an API for the CPU to use the UART:

- **`uart.h`** : Defines macros for the register addresses (e.g., `UART_DATA_REG` , `UART_STATUS_REG` , `UART_CONTROL_REG`) and status bits (`UART_RX_RDY` , `UART_TX_RDY`).
- **`uart.c`** :
  - `uart_putc(char c)` : Polls `UART_TX_RDY` in `UART_STATUS_REG` until the transmitter is ready, then writes the character `c` to `UART_DATA_REG` .
  - `uart_getc()` : Polls `UART_RX_RDY` in `UART_STATUS_REG` until a character is received, then reads it from `UART_DATA_REG` .
  - `uart_set_freq(uint32_t cnt_max)` : Writes the `cnt_max` (baud rate divisor) to `UART_CONTROL_REG` .
  - `uart_get_freq()` : Reads the current baud rate divisor from `UART_CONTROL_REG` .

An example of dynamic UART frequency control is shown in `program/user/uart_edit.c` , where button inputs are used to query or set the UART baud rate divisor.

9.6. VGA

The VGA (Video Graphics Array) module in our CPU project allows for displaying graphics on a connected VGA monitor. It involves both hardware (Verilog) and software (C) components.

9.6.1. Hardware Implementation

1. **VGA Controller (`mycpu.srcs/sources_1/new/vga_top.v`):**

```
module vga_top(
  input clk,
  input reset,
  output reg [3:0] r,
  output reg [3:0] g,
  output reg [3:0] b,
  output h_sync,
  output v_sync,
  output reg [31:0] vga_addr,         // used to fetch pixel data from memory
  input [31:0] vga_data               // pixel data from memory
);
```

- This module is responsible for generating the VGA timing signals: Horizontal Sync (`h_sync`) and Vertical Sync (`v_sync`).
- It maintains internal counters (`x` , `y`) to track the current pixel being drawn on the screen.
- It generates a `vga_addr` that sequentially scans through a dedicated VGA memory region. This address is used to fetch pixel data.

- The `vga_data` input receives a 32-bit word from the VGA memory. Since each pixel is 4 bits (allowing 16 colors), one 32-bit word contains data for 8 pixels.
- An internal `vga_addr_offset` (3 bits) selects which of the 8 pixels within the `vga_data` word corresponds to the current `x` coordinate.
- The selected 4-bit `vga_data_pixel` is then decoded into 12-bit RGB values (4 bits for Red, 4 for Green, 4 for Blue) using a series of conditional assignments based on color codes defined in params.v. These RGB values are output as `r`, `g`, `b`.
- The `data_en` signal indicates when the `x` and `y` counters are within the active display area.

2. **VGA Memory (Frame Buffer):**
   - If the `VGA macro is defined in params.v, the `memory.v` module instantiates a block RAM (`blk_mem_vga`) to serve as the VGA frame buffer.
   - This memory is dual-ported:
     - **CPU Write Port:** The CPU can write pixel data to this memory. Accesses are determined by `D_addr` (specifically `D_addr[17:2]` for the address within the VGA memory region) and `D_store_data`. The write enable `wevga` is asserted when `io_en` is high, `io_sel` matches `VGA` (11'd4), and a store operation is active.
     - **VGA Read Port:** The vga_top.v module reads from this memory using `vga_addr` (specifically `vga_addr[17:2]`) to get `vga_data`. This port is driven by `clk_pixel`.
3. **Memory-Mapped I/O (`mycpu.srcs/sources_1/new/memory.v`):**
   - The VGA frame buffer is mapped into the CPU's address space. The base address for MMIO is `ADDR_MMIO_BASE` (0x80000000), and the VGA region starts at an offset `ADDR_MMIO_VGA` (0x00400000), as defined in params.v. So, the VGA memory starts at `0x80400000`.
   - The `io_sel` signal in `memory.v` decodes `D_addr[30:20]` to select the VGA peripheral when it's `11'd4`.
4. **Integration (core.v and `mycpu.srcs/sources_1/new/top.v`):**
   - The `core.v` module instantiates the `memory` module.
   - The `top.v` module instantiates both the `core` and `vga_top` modules, connecting the `vga_addr` and `vga_data` signals between them (indirectly via the `memory` module for CPU access, and directly for `vga_top`'s read access if `blk_mem_vga` is within memory.v). The physical VGA output pins (`vga_r`, `vga_g`, `vga_b`, `vga_h_sync`, `vga_v_sync`) are driven by vga_top.v.

9.6.2. Software Control

The C library in vga provides functions to interact with the VGA display:

- **`vga.h`:**
  - Defines `VGA_BASE_ADDR` as `0x80400000` (calculated from `ADDR_MMIO_BASE` + `ADDR_MMIO_VGA`).
  - Defines screen dimensions like `VGA_WIDTH_PIXELS` (640), `VGA_HEIGHT_PIXELS` (480), and `VGA_WIDTH_BYTES` (320, since 2 pixels per byte).
  - Defines 4-bit color constants (e.g., `BLACK`, `WHITE`, `RED`).
  - Declares an external 8x8 pixel font map: `font_88[128][8]`.
  - Declares functions for drawing: `vga_draw_point`, `vga_clear`, and `vga_print_char`.
- **`vga.c`:**
  - **`font_88` Array:** Contains the bitmap data for ASCII characters U+0000 to U+007F. Each character is 8 pixels wide and 8 pixels high. Each `char` in the inner array represents a row of 8 pixels.

- **`vga_clear(uint8_t color)`:** Fills the entire VGA screen with a specified `color`. It iterates through the VGA memory region (from `VGA_BASE_ADDR`) and writes to each byte. Since each byte stores two 4-bit pixels, it writes `(color << 4) | (color & 0xF)` to set both pixels in the byte to the same color.
- **`vga_draw_point(uint32_t x, uint32_t y, uint8_t color)`:** Draws a single pixel at screen coordinates (`x`, `y`) with the given `color`.
  - It calculates the memory address: `pointer = VGA_BASE_ADDR + VGA_WIDTH_BYTES * y + (x >> 1)`. The `(x >> 1)` part is because each byte holds two horizontal pixels.
  - It then checks if `x` is odd or even (`x & 0x1`) to determine if the color should be written to the upper nibble (`*pointer = (*pointer & 0xF) | (color << 4);`) or the lower nibble (`*pointer = (*pointer & 0xF0) | (color & 0xF);`) of the byte, preserving the other pixel in that byte.
- **`vga_print_char(uint32_t x, uint32_t y, uint8_t char_code, uint8_t fill, uint8_t color)`:** Renders an 8x8 character on the screen.
  - `x` and `y` are character-grid coordinates (not pixel coordinates).
  - It iterates 8 times for each row of the character (`i` from 0 to 7).
  - It fetches the corresponding row data (`line = font_88[char_code][i];`).
  - It then iterates 8 times for each pixel in that row (`j` from 0 to 7).
  - If the `j`-th bit of `line` is set (`(line >> j) & 1`), it calls `vga_draw_point(x + j, y + i, color)` to draw the character's pixel.
  - If `fill` is true and the bit is not set, it draws a background pixel using `BACKGROUND_COLOR`.

In summary, the CPU writes pixel data into a memory-mapped frame buffer. The `vga_top` hardware module continuously reads this frame buffer and generates the necessary VGA signals to display the image on a monitor. The C library provides convenient functions for the CPU to draw points and text on this display.

9.7. PS/2 Keyboard

The keyboard module allows our CPU to receive input from a PS/2 keyboard. It involves hardware for PS/2 signal processing and scan code conversion, and software for the CPU to read the processed key codes.

9.7.1. Hardware Implementation

```
module keyboard(
    input clk,
    input reset,
    input ps2_clk,
    input ps2_data,
    output reg [7:0] key_code
);
```

1. **PS/2 Signal Handling:**
   - **Clock Stabilization:** The incoming `ps2_clk` from the keyboard is filtered using a shift register (`ps2_clk_filter`) to create a stable internal version, `ps2_clk_val`. This helps to debounce the clock signal.
   - **Negative Edge Detection:** Data bits from the keyboard are typically valid on the falling edge of the PS/2 clock. The module generates `ps2_negedge` to identify these moments.
2. **Serial Data Reception (FSM):**

- A simple Finite State Machine (FSM) with two states (`IDLE` , `RX`) manages the reception of the 11-bit serial frames from the keyboard (1 start bit, 8 data bits, 1 parity bit, 1 stop bit - though the FSM counts 10 bits, implying parity might be ignored or the count is for edges after start bit).
- **`IDLE` state:** Waits for a `ps2_negedge` (signifying the start bit). Upon detection, it transitions to the `RX` state and sets a counter (`num_next`) to 10.
- **`RX` state:** On each subsequent `ps2_negedge` , it shifts the `ps2_data` bit into an 11-bit register `rx_buffer_next` and decrements the counter.
- Once the counter reaches zero, a full frame has been received. The 8 data bits (scan code) are available in `rx_buffer[8:1]` . A `done` signal is asserted for one clock cycle.

3. **Scan Code Processing:**
   - When `done` is asserted:
     - The raw 8-bit scan code from `rx_buffer[8:1]` is captured into `scan_code` .
     - It detects the "break code" (`BREAK_SCAN` , typically 0xF0) which keyboards send when a key is released. This sets the `break_now` flag.
     - It handles the Caps Lock key (`CAPS_SCAN`). If `CAPS_SCAN` is detected on a key release (indicated by `break_now` from the previous cycle when `CAPS_SCAN` was pressed), the internal `caps` flag is toggled.
     - The `scan_code_prev` register stores the previous scan code to help with this logic.
     - The logic
       `scan_code <= ((rx_buffer[8:1] == scan_code_prev) & break_now) ? NONE_SCAN : rx_buffer[8:1];`
       aims to filter out certain sequences, potentially to ensure `NONE_SCAN` is output if a break code is immediately repeated or if it's the break of the `NONE_SCAN` itself (though `NONE_SCAN` is usually an output, not an input scan code). Primarily, `scan_code` takes the value of `rx_buffer[8:1]` .
4. **Key Code Mapping:**
   - A combinational `always @(*)` block uses a `case` statement to translate the processed `scan_code` into a final 8-bit `key_code` .
   - This mapping considers the state of the `caps` flag to output uppercase or lowercase letters and to handle shifted symbols for numbers.
   - The raw scan code constants (e.g., `A_SCAN` , `ONE_SCAN`) and the target ASCII-like key codes (e.g., `a` , `A` , `ONE` , `EXCLAMATION`) are defined in params.v.
   - If the `scan_code` is not recognized, `key_code` defaults to `NONE_SCAN` (0xFF).
5. **CPU Interface:**
   - The resulting 8-bit `key_code` is an output of the `keyboard` module.
   - This `key_code` is made available to the CPU through Memory-Mapped I/O (MMIO).
   - In the `memory.v` module, when the CPU reads from the keyboard's address (`ADDR_MMIO_BASE` + `ADDR_MMIO_KEYBOARD` , which is `0x80000000 + 0x00500000 = 0x80500000`), the value `{24'h0, key_code}` is returned.

9.7.2. Software Interface

- **`keyboard.h` :**
  - Defines `KEYBOARD_REG` as a volatile pointer to the MMIO address `0x80500000` .
  - Provides a set of `k_` constants (e.g., `k_A` , `k_a` , `k_SPACE` , `k_NONE`) that correspond to the `key_code` values generated by the hardware. These make the C code more readable.
  - Declares the function `uint8_t keyboard_read();` .
- **`keyboard.c` :**

- Implements `keyboard_read()`, which simply dereferences `KEYBOARD_REG` to read the current `key_code` from the hardware.

Integration:

- The `keyboard` Verilog module is instantiated in `top.v`, connecting its `ps2_clk` and `ps2_data` inputs to the FPGA pins for the PS/2 port.
- The `key_code` output is routed through the `core.v` module to the `memory.v` module.
- The CPU polls the `KEYBOARD_REG` by calling `keyboard_read()` to check for new key presses. The hardware continuously updates this register with the latest processed key event.

10. Challenges

Several of the many problems I encountered during the development are listed below in arbitrary order:

1. **UART drifting**
   - Problem: UART fails to receive data properly after the first few bytes.
   - Solution: Let the receiver re-calibrate its sampling points at the start of every incoming byte and try to sample each bit at its most stable point (the middle). This allows it to tolerate small deviations in the baud rate.
2. **Vivado Simulation Stuck**
   - Problem: Vivado simulation is stuck at some point.
   - Solution: Look for and eliminate circular dependencies in the design.
3. **Not Writing Back In WB Stage**
   - Problem: If register file is sensitive to positive edge of the clock, then its content will not be changed until the end of WB. This contradicts the 5-stage pipeline model described in textbook.
   - Solution: Make the register file write on negative edge.
4. **Not Getting Data On Time From BRAM**
   - Problem: If the BRAM is sensitive to positive edge of the clock, it will only output data the cycle after the address of that piece of data is fed to it. This contradicts the 5-stage pipeline model described in textbook.
   - Solution: Make the BRAM act on negative edge.
5. **Possible Bitstream Corruption**
   - Problem: Occasionally, when the bitstream generated by Vivado is loaded to FPGA, it simply does not work properly. Worse still, the FPGA becomes totally irresponsive (except for the `PROG` button) despite the LED signaling "bitstream successfully loaded" is on. Even the 7-segment display, which is lit up by default, is off.
   - Solution: Just regenerate the bitstream and the problem is solved. The exact cause of the problem remains unknown.

11. Use of AI & Online Resources

- AI helped in writing part of the documentation.
- We learned from MIT's `xv6` OS when implementing the software trap handler. We stripped off all the process-related functionality and support for interrupts, greatly simplifying the handler.

12. Conclusion

We should always simulate more.