CS683 Adv Computer Architecture

Course Project

Super se OoOpar

Group Members

Rishabh Ravi, Jujhaar Singh, Kalp Vyas, Sankalp Bhamare

Introduction

Hi! We are a group of 4 third year undergraduate students, who, as a part of the course CS683, undertook a challenging task, usually done by large teams of experts in the industrial setting, of designing a superscalar processor! It was a fantastic, intellectualy stimulating, and exciting experience and we would like to thank the TAs and Professor Virendra Singh for giving us the knowledge and opportunity to go through with this project.

Design of the Processor

Branch Predictor

```
entity bpt is
   port (rst, clk, b_obs, unstall: in std_logic;
         opcode: in  std_logic_vector(3 downto 0);
         pc_in: in std_logic_vector(15 downto 0);
         dest_reg: in std_logic_vector(2 downto 0);
         b_pred: out std_logic;
         pc_out: out std_logic_vector(15 downto 0)
        );
end entity;
```

Takes in an opcode to see if the inst is branching or not. It has a two-bit FSM that (00(strong NT),01(weak NT), and 10(weak T), and 11(strong T)) decides if the branch will be taken or not. If the predictor predicts it to be taken, it stalls the fetch stage and holds it like that till the execution of the branching instr is done. Information of the completed execution of the branching exec is received by a bit b_obs. Once this bit is high, it implies the branch speculated to be taken has finished execution, and it brings down b_pred (the prediction) to uninstall the

system. Moreover, the PC of the instr that caused the stall is also sent to the exec unit so that it knows when the same is executed so b_obs can be toggled high.
Furthermore, modification to R7 is found if the destination register of the instruction (dest_reg) is “111” and the opcode is such that it alters the value stored at r7; in that case, the prediction is set high.

Load Store RS and Load Store Execution Unit

This unit is responsible for handling the inorder execution of load store instructions, it is responsible for waiting until register dependencies are resolved and ensuring inorder write-back to the memory.

This unit takes two decoded instructions from the dispatch stage as the input and then loads them into a fifo queue.

At each cycle a instruction is popped from the fifo queue, and using the memory interface, appropriate mode is selected i.e. read/write and data is written to the memory bus and effectively memory access / writeback is achieved.
If an instruction is not ready, i.e. the operands are not resolved from the register file, the queue is stalled until the operands get resolved.

How is memory access done?
We have the following 4 ports:

```
mem_addr_bus: out std_logic_vector(15 downto 0); -- memory read addr
mem_read_data_bus: in std_logic_vector(15 downto 0); -- from data memory
mem_write_en: out std_logic; -- to memory to enable write
mem_write_data_bus: out std_logic_vector(15 downto 0);
```

Mem_addr_bus, carries the address on which the memory access is to be done.
Mem_read_data_bus: to receive data sent from the memory.
Mem_write_data_bus: to send data to be written to the memory
Mem_write_en: toggles read / write mode of memory.

This component also has access to PRF, (Physical Register File) using this it loads all the required operands(ra, rb) for the different memory access instructions. (This works exactly similar to the RS).
It also accesses the PRF to perform writeback of the loaded memory data.(lw instruction), where it writebacks to the prf the loaded data.
Since memory instructions are executed in order all memory conflicts are avoided.(RAW, WAR)

Reservation Station

The reservation station receives two decoded instructions from the dispatch stage along with renamed registers from prf.
An entry is made for each instruction in the RS. The entry consists of:
instr opcode, oper_1, validity of operand 1, same for operand 2, imm6, imm9,and a ready bit, age

Now the working of RS:
In each cycle all the entries of the RS check the validity of their operands if they are not valid, the RS using a PRF_ADDR_BUS and PRF_DATA_BUS fetches the entries from the PRF, if the renamed registers are no longer busy, their values are fetched and written into the RS, once both the operands are fetched the instruction is ready to be issued.

How are instructions issued?
Each entry in the rs has a field age, now since multiple instructions can be ready but only a few issued, (limited no. of execution units) we need to have some metric to decide which instruction to issue first.
We use the metric age: no of cycles the entry has been in the RS.
The one with the largest age is selected for execution.

Execution Unit

This unit is responsible for the out-of-order execution of the instructions. There are 3 separate pipelines where execution takes place, one each for load store instructions, branch instructions and arithmetic instructions.

```
entity exec_unit is
   port(
   clk, stall, reset: in std_logic;
   alu1_dest , ls1_dest : in std_logic_vector(15 downto 0);
   alu1_oper1, alu1_oper2, ls1_oper: in std_logic_vector(15 downto 0);
   alu1_imm6 : in std_logic_vector(5 downto 0);

   ls1_imm6 : in std_logic_vector(5 downto 0);
   ls1_imm9 : in std_logic_vector(8 downto 0);
   br1_mode: in std_logic_vector(3 downto 0);
   br1_oper1, br1_oper2, br1_pc_in, ls1_pc_in, alu1_pc_in  : in std_logic_vector(15
downto 0); --need 2 operands to check for BEQ
   br1_imm6 : in std_logic_vector(5 downto 0);
   br1_imm9 : in std_logic_vector(8 downto 0);
   alu1_mode: in std_logic_vector(5 downto 0);
   ls1_mode: in std_logic_vector(3 downto 0);
   branch_mode: in std_logic_vector(2 downto 0);
```

```
   stall_pc1, stall_pc2 : in std_logic_vector(15 downto 0);

   ls_dest,ls_value : out std_logic_vector(15 downto 0);
   br_value, br_pc, alu_pc, ls_pc : out std_logic_vector(15 downto 0);
   br_mode: out std_logic_vector(3 downto 0);
   ls_mode: out std_logic_vector(3 downto 0);
   alu_c, alu_z: out std_logic;
   br_c, br_z: out std_logic;
   ls_c, ls_z: out std_logic;
   br_eq1, br_eq2 : out std_logic;
   stall_out : out std_logic;
   --connections to prf
   alu1_reg_data: out std_logic_vector(15 downto 0);
   alu1_reg_addr: out std_logic_vector(15 downto 0);
   alu1_reg_en: out std_logic
   );



end entity exec_unit;
```

For each type of the instruction, we get the required value to be stored/loaded and the destination register for it. Further, we also send the br_eq bit to see if the BEQ branch is taken or not and this information is sent to the branch predictor. Another bit called stall_out is used to decide whether to stop stalling the processor or not.

Instruction Fetch

```
entity IF_STAGE is
port(clk, stall,unstall,rst: in std_logic;
                mem_data_in_1, dest_pc, mem_data_in_2, pc_in: in std_logic_vector(15
downto 0);
                instr_1, instr_2, mem_addr_1, mem_addr_2, pc_out, pc_out_1, pc_out_2: out
std_logic_vector(15 downto 0)
        );
end IF_STAGE;
```

The IF stage takes two instructions per cycle from the main memory by sending the address PC and PC+1. It returns the same PC to an incrementor that outputs PC+2, which is taken in as input by the IF stage. The two extra pc_out are the PCs of the two instr fetched.

When stall goes high the IF stage stops fetching, and when rst goes high PC goes to 0, so it reads from the start.
Unstall is a bit that is sent by the exec unit to indicate the completion of execution of a branch inst that the predictor speculates to be taken. When this bit goes high it allows writing to the PC, by the value sent by the exec unit.

Instruction Decode

```
entity ID_STAGE is
port(clk, stall, rst: in std_logic;
        instr_in_1, instr_in_2, pc_in_1, pc_in_2: in std_logic_vector(15 downto 0);
        r_1, r_2, r_3, r_4, r_5, r_6: out std_logic_vector(3 downto 0); -- 1111 don't care
        opcode_1, opcode_2: out std_logic_vector(3 downto 0);
        alu_op_1, alu_op_2: out std_logic_vector(1 downto 0);
        imm6_1, imm6_2 : out std_logic_vector(5 downto 0);
   imm9_1, imm9_2 : out std_logic_vector(8 downto 0);
        pc_out_2, pc_out_1  : out std_logic_vector(15 downto 0)
        );
end ID_STAGE;
```

The decode stage takes in the two instructions and PCs from the fetch stage. It finds out the opcode, and extended opcode for alu instruction (ADZ, ADC, ADD, etc). It also returns the register immediate fields of the instr. If the instr doest use any reg or immediate field those bits are set to high.

Dispatch stage

This stage is used to send the decoded instructions to their respective reservation stations.

```
entity dispatch is
   port(
        clk, reset: in std_logic;  --system ip

        -- input coming from ID
        id_r_1, id_r_2, id_r_3, id_r_4, id_r_5, id_r_6: in std_logic_vector(3 downto
0); -- 1111 don't care
        id_opcode_1, id_opcode_2: in std_logic_vector(3 downto 0);
        id_alu_op_1, id_alu_op_2: in std_logic_vector(1 downto 0);
```

```
       id_imm6_1, id_imm6_2 : in std_logic_vector(5 downto 0);
       id_imm9_1, id_imm9_2 : in std_logic_vector(8 downto 0);
       id_pc_out_2, id_pc_out_1  : in std_logic_vector(15 downto 0);
       prf_r_1, prf_r_2, prf_r_3, prf_r_4, prf_r_5, prf_r_6: in std_logic_vector(15
downto 0);  --rr stage
       prf_v_2, prf_v_3, prf_v_5, prf_v_6 : in std_logic;  --from rr

       -- dispatch to regular rs (everything except load-store instructions)
       rs_pc_in_1, rs_pc_in_2:  out std_logic_vector(15 downto 0); --id stage
       rs_opcode_1, rs_opcode_2: out std_logic_vector(3 downto 0);  --id stage
       rs_alu_op_1, rs_alu_op_2: out std_logic_vector(1 downto 0);  -- id stage
       rs_imm6_1, rs_imm6_2: out std_logic_vector(5 downto 0);  -- id stage
       rs_imm9_1, rs_imm9_2 : out std_logic_vector(8 downto 0);  -- id stage
       rs_r_1, rs_r_2, rs_r_3, rs_r_4, rs_r_5, rs_r_6: out std_logic_vector(15 downto
0);  --rr stage
       rs_v_2, rs_v_3, rs_v_5, rs_v_6 : out std_logic;  --from rr

       -- dispatch to load store rs (only load-store instructions)
       ls_rs_pc_in_1, ls_rs_pc_in_2:  out std_logic_vector(15 downto 0); --id stage
       ls_rs_opcode_1, ls_rs_opcode_2: out std_logic_vector(3 downto 0); --id stage
       ls_rs_imm6_1, ls_rs_imm6_2: out std_logic_vector(5 downto 0);  -- id stage
       ls_rs_imm9_1, ls_rs_imm9_2 : out std_logic_vector(8 downto 0);  -- id stage
       ls_rs_r_a_1, ls_rs_r_b_1, ls_rs_r_a_2, ls_rs_r_b_2: out std_logic_vector(15
downto 0);  --rr stage
       ls_rs_v_a_1 ,ls_rs_v_a_2, ls_rs_v_b_1, ls_rs_v_b_2: out std_logic;  --from rr

       -- dispatch to rob (all instructions)
       rob_enable1, rob_enable2: out std_logic; -- enables for channels (each channel
corresponds to one entry to be added)
       rob_r1, rob_r4: out std_logic_vector(15 downto 0); -- dest regs
       rob_rr1, rob_rr4: out std_logic_vector(15 downto 0); -- rename regs assigned to
the instr
       rob_PC1, rob_PC4: out std_logic_vector(15 downto 0)
   );
end entity dispatch;
```

Depending on the opcode of the instruction we get, we either send the decoded instruction values to the specialized reservation station or our normal reservation station used for branch and arithmetic instructions. All the instructions are sent to the ROB too.

Handling LM and SM

We decided to handle LM and SM by splitting them into multiple instructions. This would mean multiple stores for SM and multiple loads for LM. These are then treated as usual in our load-store unit and so on. One complexity which arose was that we may ingest an LM into the dispatch stage which has, say, 4 registers to be loaded, so then we would require 4 load instructions that would be equivalent. However, since we can only dispatch upto 2 instructions in a cycle due to our two-wide pipeline design, we require a FIFO buffer to store these equivalent instructions in the mean time. This FIFO buffer was implemented as a circular queue in a similar manner to how it was implemented in the ROB.

Re-order Buffer

```
entity rob is
   generic(
      rob_size: integer := 20
      -- there are 8 architectural registers
   );
   port(
      clk, stall, reset, exec: in std_logic; --exec is bp taken or not

      -- ports for adding to ROB (from the dispatch stage)(actually coming from rs)
      enable1, enable2: in std_logic; -- enables for channels (each channel corresponds to one
entry to be added)
      r1, r4: in std_logic_vector(15 downto 0); -- dest regs
      rr1, rr4: in std_logic_vector(15 downto 0); -- rename regs assigned to the instr
      PC1, PC4, PC_BP1, PC_BP2: in std_logic_vector(15 downto 0);

      -- ports for writing back to the prf
      write_en: out std_logic_vector(0 to rob_size-1);
      write_rreg: out bit16_vector(0 to rob_size-1);
      write_destreg: out bit16_vector(0 to rob_size-1);

      -- to prf
      addr_bus: out addr_array(0 to rob_size-1);
      busy_bus: in std_logic_vector(0 to rob_size-1)

      -- inputs from the execution unit
      -- exec_inp_en: in std_logic_vector(0 to 1);
      -- exec_inp_pc: in exec_inps;
      -- exec_inp_vals: in exec_inps

   );
end entity rob;
```

The columns of the ROB are:

Entry_valid: if the given entry in the ROB is valid or not
Spec: if the instruction is speculative or not
Dest_reg: the final destination architectural register
Rename_reg: rename reg assigned for writeback
PCs: the PC value of the instruction

```
architecture idnar of rob is
   -- defining the cols of the ROB
   signal entry_valid: std_logic_vector(0 to rob_size-1) := (others=>'0');
   signal spec : std_logic(0 to rob_size-1) := (others=>'0');
   signal dest_reg: bit16_vector(0 to rob_size-1) := (others => (others=>'0'));
   signal rename_reg: bit16_vector(0 to rob_size-1) := (others => (others=>'0'));
   signal PCs: bit16_vector(0 to rob_size-1) := (others => (others=>'0'));

   -- other state variables
   signal head: integer := 0;
   signal insert_loc: integer := 0;
   signal is_full: std_logic := '0';
Begin
```

A circular queue data structure is maintained inside the ROB to implement its functionality. The head pointer is called “head” and the tail pointer is called “insert_loc”. There is also a variable called “is_full” that keeps track of whether there are any free spots in the ROB or not.

Broadly, the ROB performs the following operations:
Adding entries to the ROB(from dispatch)
There are ports from the dispatch stage to the ROB which help us add new entries to the ROB as and when available. These include:

```
    enable1, enable2: in std_logic; -- enables for channels (each channel corresponds to one entry to be added)
    r1, r4: in std_logic_vector(15 downto 0); -- dest regs
    rr1, rr4: in std_logic_vector(15 downto 0); -- rename regs assigned to the instr
    PC1, PC4, PC_BP1, PC_BP2: in std_logic_vector(15 downto 0);
```

When an entry is added at the tail, “insert_loc”, the variable is incremented by 1 and the values that we get from the dispatch stage are added to this new entry.

Removing entries and writing back to the PRF
When removing entries from the ROB, we constantly look at the head. If the busy value of the rename register assigned to the head is 0, it means that the instruction has been executed and the value in the PRF entry for that rename register is valid. Thus we can retire this instruction from the ROB. We increment by 1 and continue in the same fashion till we find an entry whose rename register in the PRF is busy. At that point we stop.

There are two sets of ports to communicate with the PRF. One is for receiving back the busy values for the rename registers:

addr_bus: out addr_array(0 to rob_size-1);
busy_bus: in std_logic_vector(0 to rob_size-1)

And the other set is for sending which registers to write back to in the PRF:

write_en: out std_logic_vector(0 to rob_size-1);
write_rreg: out bit16_vector(0 to rob_size-1);
write_destreg: out bit16_vector(0 to rob_size-1);

The actual writeback logic for rename and architectural registers is implemented in the PRF code.

Handling Branching

PC_BP1, PC_BP2, stall, exec are ports to handle branch misprediction. The two PCs tell the branch where the predictor speculates, so after which all instructions are marked as speculated. When the prediction is ‘taken’, the stall is high, as the system stops fetching, the exec bit tells us if the pred is true or not. If both are 1, the speculation bits are toggled off. If not those with speculation bits ‘1’ are made invalid allowing those entries in ROB to be free for overwriting. Alternatively, if it predicts it to not be taken, stall is ‘0’ and exec bit tells us the nature of the prediction. Handling misprediction is done appropriately.
The PC_BP1 and PC_BP2 tell us which PC were speculated and are used to mark all PC that are greater than either to marked as speculated. This works in all cases.

Physical Register File and Register Renaming

The register file is maintained as a physical reg file and a lookup table to point out which of the entries in the PRF are locations of one of the registers R1 to R7.

```
entity rename_registers is
   generic(
      prf_size: integer := 128;
      rs_size: integer := 16;
      rob_size: integer := 20
      -- there are 8 architectural registers
   );
   port(gr1,gr4,
   gr2,gr3,gr5,gr6: in std_logic_vector(2 downto 0);
   reset, clk, stall: in std_logic;

   -- gr1-6 are the registers from the two lines of code above(in that order)
   prf_addr_bus: in addr_array(0 to rs_size*2-1);
   prf_data_bus: out prf_data_array(0 to rs_size*2-1); -- (busy) (16 BIT DATA)
```

```
    -- prf inputs from rob:
    rob_write_en: in std_logic_vector(0 to rob_size-1);
    rob_write_rreg: in bit16_vector(0 to rob_size-1);
    rob_write_destreg: in bit16_vector(0 to rob_size-1);

    --prf inputs from alu wb:
    alu1_reg_data: in std_logic_vector(15 downto 0);
    alu1_reg_addr:  in std_logic_vector(15 downto 0);
    alu1_reg_en: in std_logic;

    -- from ls rs
    ls_rs_wb_addr: in std_logic_vector(15 downto 0); -- addr for writing
    ls_rs_wb_en: in std_logic; -- to memory to enable write
    ls_rs_wb_data: in std_logic_vector(15 downto 0);

    rr1,rr4,rr2,rr3,rr5,rr6:  out std_logic_vector(15 downto 0);
    v2,v3,v5,v6: out std_logic;
    -- for args: r2, r3, r5, r6 we return the addr of rename reg
    -- for dests: r1, r4 we return the new rename reg assigned to them

    -- to rob
    rob_addr_bus: in addr_array(0 to rob_size-1);
    rob_busy_bus: out std_logic_vector(0 to rob_size-1)
    );
end entity;
```

The “gr” are registers pointed to by instruction. This then undergoes renaming based on previous dependencies. “Rr”s are the rename registers that are output from this component after renaming is complete.

The instruction flow that is assumed is:
r1 = r2 + r3
r4 = r5 + r6
So, this makes r1 and r4 the destination registers while making r2, r3, r5, and r6 the argument registers.

The PRF broadly handles the following tasks:
Register Renaming
We perform regular register renaming as was explained in class. As an instruction enters, we assign a rename register to the destination. We also use the argument architectural registers and use the RAT to find which rename register they are currently assigned to and then use that to find the value that we must used while executing the instruction. The rest of the process is orchestrated by the reservation station which takes care of finding the instructions that are ready using the argument rename registers and issues them accordingly.

Write back to renamed registers from the execution unit

The values that are produced from the execution unit are piped through to the PRF. The PRF makes use of these values and writes them back to the appropriate rename registers. The ROB can then use this values during its writeback stage

Write back to renamed registers from the load store unit

In a similar manner to the writeback from execution unit we also perform a writeback from the load-store unit for the load instructions that are executed

Write back to architectural registers from their renamed register based on inputs from the ROB

When retiring instructions, we write back the value from the renamed register to the architectural register value field. It represents the value of the architectural register when the instructions that have been retired would have been executed on any regular processor.

Reference counting and Freeing renamed registers

We maintain a count of occurrences of each rename register. When renaming an instruction, we add 1 to the count of each of the rename registers used, two as arguments and one as the destination. We decrement the count of the arg rename registers when they are issued from the RS and we decrement the count of the dest registers when retiring from the ROB.