Constructive Computer Architecture

Tutorial 6

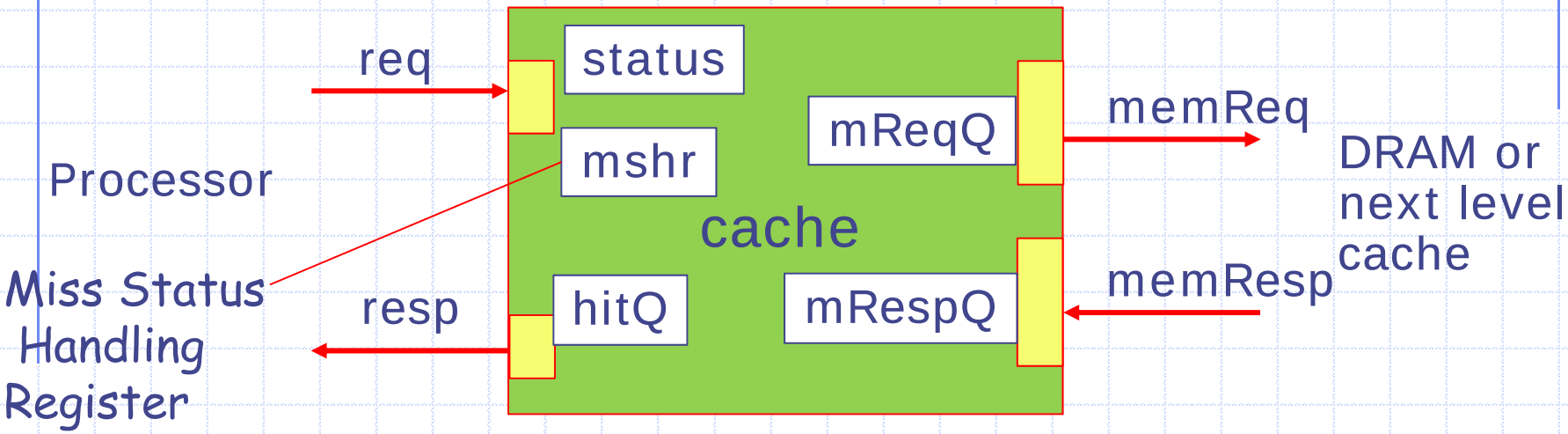
Cache & Exception

Sizhuo Zhang

6.175 TA

Blocking Cache Code

Blocking Cache Interface




```
interface Cache;
   method Action req(MemReq r);
   method ActionValue#(Data) resp;

   method ActionValue#(LineReq) memReq;
   method Action memResp(Line r);
endinterface
```

Blocking cache
state elements

```
module mkCache(Cache);
  RegFile#(CacheIndex, Line) dataArray <- mkRegFileFull;
  RegFile#(CacheIndex, Maybe#(CacheTag))
                               tagArray <- mkRegFileFull;
  RegFile#(CacheIndex, Bool) dirtyArray <- mkRegFileFull;

  Fifo#(1, Data)      hitQ <- mkBypassFifo;
  Reg#(MemReq)     missReq <- mkRegU;
  Reg#(CacheStatus) msrh <- mkReg(Ready);

  Fifo#(2, LineReq) memReqQ <- mkCFFifo;
  Fifo#(2, Line)   memRespQ <- mkCFFifo;

  function CacheIndex getIdx(Addr addr) = truncate(addr>>4);
  function CacheTag getTag(Addr addr)   = truncateLSB(addr);
```

Req method
hit processing

```
method Action req(MemReq r) if(mshr == Ready);
  let idx = getIdx(r.addr); let tag = getTag(r.addr);
  Bit#(2) wOffset = truncate(r.addr >> 2);
  let currTag = tagArray.sub(idx);
  let hit = isValid(currTag)?
            fromMaybe(?,currTag)==tag : False;
  if(hit) begin
    let x = dataArray.sub(idx);
    if(r.op == Ld) hitQ.enq(x[wOffset]);
    else begin
      x[wOffset]=r.data;
      dataArray.upd(idx, x); dirtyArray.upd(idx, True);
    end
  end else begin missReq <= r; mshr <= StartMiss; end
endmethod
```

Start-miss rule

Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready

```
rule startMiss(mshr == StartMiss);
  let idx = getIdx(missReq.addr);
  let tag=tagArray.sub(idx);
  let dirty=dirtyArray.sub(idx);
  if(isValid(tag) && dirty) begin // write-back
    let addr = {fromMaybe(?,tag), idx, 4'b0};
    let data = dataArray.sub(idx);
    memReqQ.enq(LineReq{op: St, addr: addr, data: data});
  end
  mshr <= SendFillReq;
endrule
```

Send-fill rule

```
Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready
```

```
rule sendFillReq (mshr == SendFillReq);
  memReqQ.enq(LineReq{op:Ld, addr:missReq.addr, data:?});
  mshr <= WaitFillResp;
endrule
```

Wait-fill rule

Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready

```
rule waitFillResp(mshr == WaitFillResp);
  let idx = getIdx(missReq.addr);
  let tag = getTag(missReq.addr);
  let data = memRespQ.first;
  tagArray.upd(idx, Valid (tag));
  if(missReq.op == Ld) begin
    dirtyArray.upd(idx,False);dataArray.upd(idx, data);
    hitQ.enq(data[wOffset]);
  end else begin
    data[wOffset] = missReq.data;
    dirtyArray.upd(idx,True); dataArray.upd(idx, data);
  end
  memRespQ.deq; mshr <= Ready;
endrule
```

Rest of the methods

```
method ActionValue#(Data) resp;
  hitQ.deq;
  return hitQ.first;
endmethod

method ActionValue#(MemReq) memReq;
  memReqQ.deq;
  return memReqQ.first;
endmethod

method Action memResp(Line r);
  memRespQ.enq(r);
endmethod
```

Memory side methods

Store Buffer Code

Store Buff: Req method
hit processing

```
method Action req(MemReq r) if(mshr == Ready);
  ... get idx, tag and wOffset
 if(r.op == Ld) begin // search stb
   let x = stb.search(r.addr); lockL1[0] <= True;
   if (isValid(x)) hitQ.enq(fromMaybe(?, x));
   else begin // search L1
     let currTag = tagArray.sub(idx);
     let hit = isValid(currTag) ?
                 fromMaybe(?,currTag)==tag : False;
     if(hit) begin
       let x = dataArray.sub(idx); hitQ.enq(x[wOffset]);
     end else begin missReq <= r; mshr <= StartMiss; end
   end
 end else stb.enq(r.addr,r.data) // r.op == St
endmethod
```

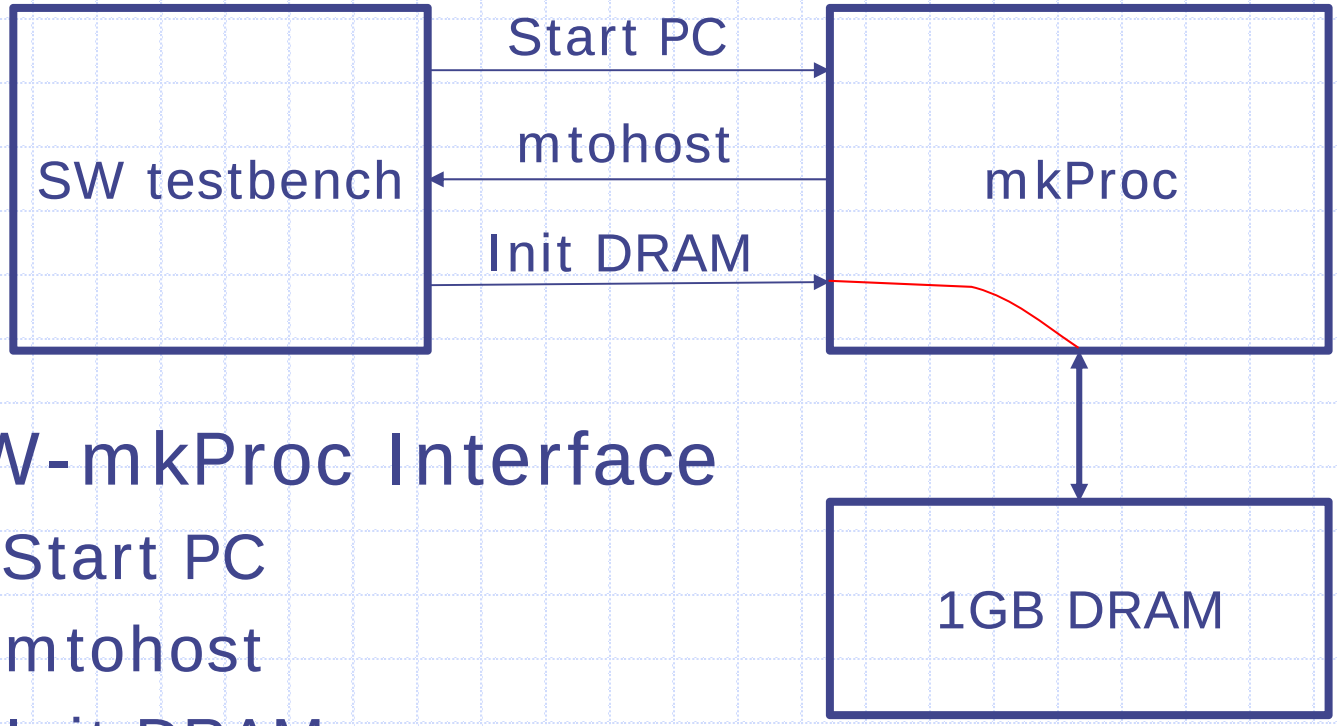
Store Buff to mReqQ

```
rule mvStbToL1 (mshr == Ready && !lockL1[1]);
  stb.deq; match {.addr, .data} = stb.first;
  ... get idx, tag and wOffset
  let currTag = tagArray.sub(idx);
  let hit = isValid(currTag) ?
              fromMaybe(?,currTag)==tag : False;
  if(hit) begin
     let x = dataArray.sub(idx);
     x[wOffset] = data; dataArray.upd(idx,x)
  end else begin
    missReq <= r; mshr <= StartMiss;
  end
endrule

rule clearL1Lock; lockL1[1] <= False; endrule
```

Lab 7 FPGA Infrastructure

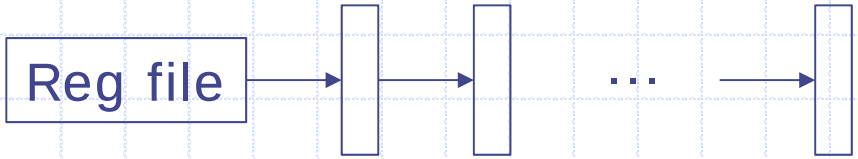
FPGA Infrastructure



- SW-mkProc Interface
  - Start PC
  - mtohost
  - Init DRAM
- Avoid re-programming FPGA
  - Reset FPGA after each test

Simulation

- DRAM
  - RegFile + pipelined delay of resp



- We also simulate DRAM initialization
  - Longer simulation time

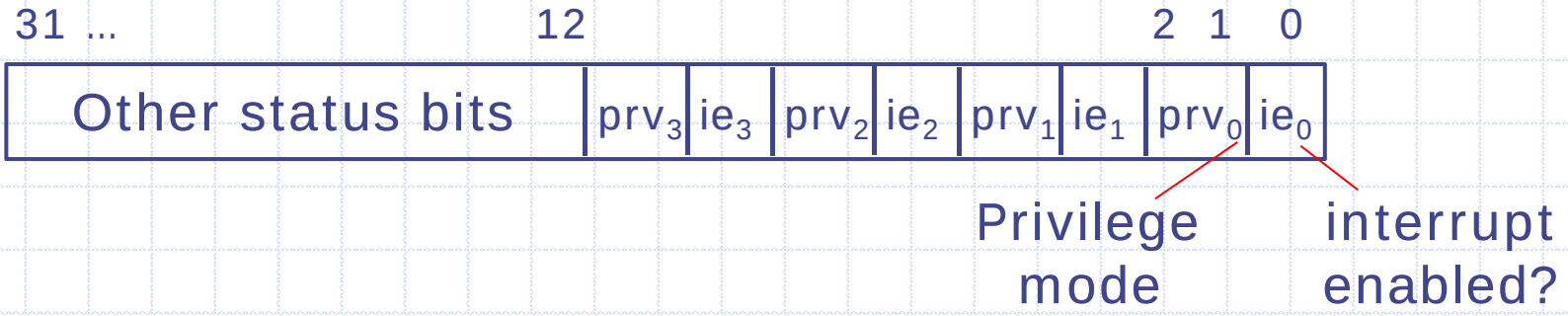
Cross Clock Domain Issues

- mkProc runs at 50MHz
- DRAM controller runs at 200MHz
- Cross clock domain
  - Non-atomic behavior at reset
  - Deq DRAM resp FIFO to empty before start new test
  - Guard all rules with processor started
- Lab 7: src/DDR3Example.bsv

Exception Code

CSR

- `mepc` holds pc of the instruction that causes the interrupt
- `mcause` indicates the cause of the interrupt
- `mscratch` holds the pointer to HW-thread local storage for saving context before handling the interrupt
- `mstatus`



- Mode: 0 – user, 3 – machine

Interrupt handler- SW

```
handler_entry: # fixed entry point for each mode
               # for user mode the address is 0x100
  j interrupt_handler # One level of indirection

interrupt_handler: # Common wrapper for all IH
  # get the pointer to HW-thread local stack
  csrrw sp, mscratch, sp # swap sp and mscratch
  # save x1, x3 ~ x31 to stack (x2 is sp, save later)
  addi sp, sp, -128
  sw x1, 4(sp)
  sw x3, 12(sp)
  ...
  sw x31, 124(sp)
  # save original sp (now in mscratch) to stack
  csrr s0, mscratch # store mscratch to s0
  sw s0, 8(sp)
```

Interrupt handler- SW *cont.*

```
interrupt_handler:
  ... # we have saved all GPRs to stack
  # call C function to handle interrupt
  csrr a0, mcause # arg 0: cause
  csrr a1, mepc # arg 1: epc
  mv a2, sp # arg 2: sp – pointer to all saved GPRs
  jal c_handler # call C function
  # return value is the PC to resume
  csrw mepc, a0
  # restore mscratch and all GPRs
  addi s0, sp, 128; csrw mscratch, s0
  lw x1, 4(sp); lw x3, 12(sp); ...; lw x31, 124(sp)
  lw x2, 8(sp) # restore sp at last
  eret # finish handling interrupt
```

C function to handle interrupt System call

```
long c_handler(long cause, long epc, long *regs) {
  // regs[i] refers to GPR xi stored in stack
  if(cause == 0x08) { // cause code for SCALL is 8
    // figure out the type of SCALL (stored in a7/x17)
    // args are in a0/x10, a1/x11, a2/x12
    long type = regs[17]; long arg0 = regs[10];
    long arg1 = regs[11]; long arg2 = regs[12];
    if(type == SysPrintChar) { ... }
    else if(type == SysPrintInt) { ... }
    else if(type == SysExit) { ... }
    else ...
    // SCALL finshes, we need to resume to epc + 4
    return epc + 4;
  }
  else if ...
```

C function to handle interrupt Emulated instruction – MUL

```
long c_handler(long cause, long epc, long *regs) {
  if(cause == 0x08) { ... /* handle SCALL */ }
  else if(cause == 0x02) {
    // cause code for Illegal Instruction is 2
    uint32_t inst = *((uint32_t*)epc); // fetch inst
    // check opcode & function codes
    if((inst & MASK_MUL) == MATCH_MUL) {
      // is MUL, extract rd, rs1, rs2 fields
      int rd = (inst >> 7) & 0x01F;
      int rs1 = ...; int rs2 = ...;
      // emulate regs[rd] = regs[rs1] * regs[rs2]
      emulate_multiply(rd, rs1, rs2, regs);
      return epc + 4; // done, resume at epc+4
    } else ... // try to match other inst
  } else ... // other causes
}
```