Latency Insensitive Design in a Latency Sensitive World

Elliott Fleming
With copious assistance from Arvind, Jamey Hicks, and Chun-Chien Lin
Latency Sensitivity

• May require less logic
  – But savings is minimal
• Many modules/Implementers
  – Designers make undocumented timing assumptions
  – Do the assumptions of one effect the other?
  – When are these assumptions exposed?
• Modular refinement
  – Major implementation gains possible
  – Does refinement affect design correctness?
Sensitive Vs. Insensitive

What if Execute takes 5 cycles? 3 cycles?
Latency Insensitivity

• Taking a data centric approach
  – Timing is irrelevant, operations should occur only when data is available
  – Modules can be refined individually
  – Bluespec has no semantic notion of clock

• Lots of examples in hardware and software
  – Bus protocols

• Takes extra logic to control the data flow
  – But Bluespec automatically generate this
Some LI Practices

• Bounded Kahnian networks
  – All inputs available before operation begins
  – Sufficient space to buffer outputs
• Transmit Data and Control together
• Ready / Enable protocol
H.264 Memory: Basic configuration

- NAL unwrap
- Parse/CAVLC
- Inverse Quant Transformation
- CAVLC Memory
- Intra Prediction
- Inter Prediction
- Ref Frames
- Parameter Memory
- Deblock Filter
- Scale / YUV2RGB
- Output Buffer
H.264 Memory and LI

• H.264 uses a variety of memories of different sizes and access frequencies

• Different performance/price requirements for different modules
  – Shared DRAM may be cheaper
  – Dedicated SRAM may use less energy

• Modules should make no assumptions about memory response times
  – Otherwise, rapid design explorations may not be possible
H.264 – Another Memory Configuration

NAL unwrap → Parse/CAVLC → Inverse Quant Transformation

Internal SRAM Memory

Intra Prediction

Inter Prediction

Deblock Filter → Scale / YUV2RGB

External DRAM Memory
Latency Insensitivity and Bluespec

• Latency insensitivity easy inside of Bluespec
  – SRAM pinouts tend not to include RDY/EN
• Can handle latency sensitive components,
  – But prefer the latency insensitive modality
• What are some good mechanisms for handling Bluespec interfaces to the outside world?
• Key idea: When to perform latency sensitive operation
  – Like a bounded Kahnian network
Latency Insensitive BRAM

- Memories are usually external to Bluespec
  - Generated RAMs
  - External RAMs
- BRAM
  - Synchronous, Dual-ported SRAM
  - Primitive FPGA component
- Wrapping a Pipelined BRAM
  - Keep an internal counter of in-flight requests
  - Allow requests if room in the response buffer
  - To maintain throughput, buffer must be as large as latency

2007-08-27
LI BRAM – Bluespec Side

- Synthesis tools are picky about BRAMs.
- Importing Verilog:
  - Legacy modules
  - Library primitives
- Three methods:
  - Read Request
  - Write Request
  - Read Response
- Use schedule to specify scheduling constraints

```verilog
module BRAM#(idx_type, data_type) provisos
  Bits#(idx_type, idx), Bits#(data_type, data),
  Literal#(idx_type);

default_clock clk(CLK);

parameter addr_width = valueof(idx);
parameter data_width = valueof(data);
parameter lo = low;
parameter hi = high;

method DOUT read_resp() ready(DOUT_RDY enable(DOUT_EN);
method read_req(RD_ADDR) ready(RD_RDY enable(RD_EN);;
method write(WR_ADDR, WR_VAL) enable(WR_EN);

schedule read_req CF (read_resp, write);
schedule read_resp CF (write);
path(DOUT_EN, RD_RDY);
endmodule
```
### LI BRAM – Verilog Side

```verilog
module BRAM(CLK, RST_N, RD_ADDR, RD_RDY, RD_EN, DOUT, DOUT_RDY, DOUT_EN, WR_ADDR, WR_VAL, WR_EN);
parameter addr_width = 1, data_width = 1;
parameter lo = 0, hi = 1;
input CLK, RST_N;
input [addr_width - 1 : 0] RD_ADDR;
input RD_EN;
output RD_RDY;
// Read Resp Port
output [data_width - 1 : 0] DOUT;
output DOUT_RDY;
input DOUT_EN;
// Write Port
input [addr_width - 1 : 0] WR_ADDR;
input [data_width - 1 : 0] WR_VAL;
input WR_EN;

reg [data_width - 1 : 0] arr[lo:hi];
reg RD_REQ_MADE;
reg [data_width - 1 : 0] RAM_OUT;
reg [1:0] CTR;
FIFOL2#(.width(data_width)) q(.
  .RST_N(RST_N), .CLK(CLK),
  .D_IN(RAM_OUT), .ENQ(RD_REQ_MADE),
  .DOUT_EN, .CLR(1'b0),
  .D_OUT(DOUT), .FULL_N(),
  .EMPTY_N(DOUT_RDY));

always@(posedge CLK)
begin
  if (!RST_N)
    CTR <= 2;
  else
    RD_REQ_MADE <= RD_EN;
    if (WR_EN)
      arr[WR_ADDR] <= WR_VAL;
    CTR <= (RD_EN) ?
      (DOUT_EN) ? CTR : CTR - 1 :
              (DOUT_EN) ? CTR + 1 : CTR;
    RAM_OUT <= arr[RD_ADDR];
end
end // always@ (posedge CLK)
endmodule
```

---

**// Read Port**
- **Input:**
  - `CLK`, `RST_N`
  - `RD_ADDR`
  - `RD_EN`

**// Read Resp Port**
- **Output:**
  - `DOUT`
  - `DOUT_RDY`

**// Write Port**
- **Input:**
  - `WR_ADDR`
  - `WR_VAL`
  - `WR_EN`
- **Reg:**
  - `arr[lo:hi]`
  - `RD_REQ_MADE`
  - `RAM_OUT`
  - `CTR`
- **Queue:**
  - `FIFOL2#(.width(data_width)) q`
Latency Insensitive Bus

- **PLB Master**
  - Fast Memory Bus
  - Embedded
- **Latency Insensitivity**
  - Support different backends
- **Challenges**
  - Burst transfer
  - Bus errors
  - Fixed latency response times
- **More complex example**
  - But same principles apply
Latency Insensitive Bus

• Burst transfer
  – Provide space to buffer the entire data of the transaction

• Bus errors
  – Can’t allow bus input to immediately escape to module

• Fixed latency response times
  – Strict input – compute – output method/rule ordering
  – Requires RWires

• Same principles apply
  – Transaction start requires all inputs
  – Sufficient buffering for output
  – Transaction complete asserts all outputs valid
Questions?
kfleming@mit.edu