Can we derive the 5-stage pipeline by successive refinement of a 2-stage pipeline?

A 2-Stage Processor in RTL

1) Design Microarchitecture
2) Locate Datapaths/Memories and create modules
3) Identify input/output ports
4) Design the Controller (FSM)

module regfile (Controller
    input [4:0] wa, //address for write port
    input [31:0] wd, //write data
    input we, //write enable (active high)
    input [4:0] ral, //address for read port 1
    output [31:0] rdl, //read data for port 1

    interface RegisterFile#(addr_T, data_T);
        data_T read(addr_T);
        Action update(addr_T, data_T);
    endinterface;

    // Code for regfile
)

Designing a 2-Stage Processor with GAA

1) Design Microarchitecture
2) Locate Datapaths/Memories and create modules
3) Define Interface methods: read, action, action value
Outline

- Single module structure
  - Performance issue
- Modular structure issues

Instructions & Templates

typedef union tagged {
struct {RName dst; RName src1; RName src2} Add;
struct {RName cond; RName addr} Bz;
struct {RName dst; RName addr} Load;
struct {RName value; RName addr} Store;
} Inst deriving(Bits, Eq);

typedef union tagged {
struct {RName dst; Value op1; Value op2} EAdd;
struct {Value cond; Iaddress tAddr} EBz;
struct {RName dst; Daddress addr} ELoad;
struct {Value data; Daddress addr} EStore;
} Inst deriving(Eq, Bits);

typedef Bit#(32) Iaddress;
typedef Bit#(32) Daddress;
typedef Bit#(32) Value;

CPU as one module

module mkCPU#(Mem iMem, Mem dMem)();
// Instantiating state elements
Reg#(Iaddress) pc <- mkReg(0);
RegFile#(RName, Value) rf <- mkRegFileFull();
SFIFO#(InstTemplate, RName) bu <- mkSFifo(findf);

// Some definitions
Instr instr = iMem.read(pc);
Iaddress predIa = pc + 1;

// Rules
rule fetch_decode ... you have seen this before
rule execute ...
endmodule
Fetch & Decode Rule

```verilog
rule fetch_and_decode (!stallfunc(instr, bu));
  bu.enq(newIt(instr, rf));
  pc <= predIa;
endrule
```

function InstrTemplate newIt(Instr instr, RegFile#(RName, Value) rf);
  case (instr) matches
    tagged Add {dst:.rd, src1:.ra, src2:.rb}:
      return EAdd{dst:rd, op1:rf[ra], op2:rf[rb]};
    tagged Bz {cond:.rc, addr:.addr}:
      return EBz{cond:rf[rc], addr:rf[addr]};
    tagged Load {dst:.rd, addr:.addr}:
      return ELoad{dst:rd, addr:rf[addr]};
    tagged Store {value:.v, addr:.addr}:
      return EStore{value:rf[v], addr:rf[addr]};
  endcase
endfunction

The Stall Function

```verilog
function Bool stallfunc (Instr instr, SFIFO#(InstrTemplate, RName) bu);
  case (instr) matches
    tagged Add {dst:.rd, src1:.ra, src2:.rb}:
      return (bu.find(ra) || bu.find(rb));
    tagged Bz {cond:.rc, addr:.addr}:
      return (bu.find(rc) || bu.find(addr));
    tagged Load {dst:.rd, addr:.addr}:
      return (bu.find(addr));
    tagged Store {value:.v, addr:.addr}:
      return (bu.find(v)) || bu.find(addr));
  endcase
endfunction
```

The findf function

```verilog
function Bool findf (RName r, InstrTemplate it);
  case (it) matches
    tagged EAdd{dst:.rd, op1:.ra, op2:.rb}:
      return (r == rd);
    tagged EBz {cond:.c, addr:.a}:
      return (False);
    tagged ELoad{dst:.rd, addr:.a}:
      return (r == rd);
    tagged EStore{value:.v, addr:.a}:
      return (False);
  endcase
endfunction
```

Execute Rule

```verilog
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd, src1:.va, src2:.vb}:
      begin
        rf.upd(rd, va+vb); bu.deq();
      end
    tagged Bz {cond:.cv, addr:.av}:
      if (cv == 0) then begin
        pc <= av; bu.clear(); end
      else bu.deq();
    tagged ELoad{dst:.rd, addr:.av}:
      begin
        rf.upd(rd, dMem.read(av)); bu.deq();
      end
    tagged EStore{value:.vv, addr:.av}:
      begin
        dMem.write(av, vv); bu.deq();
      end
  endcase
endrule
```
Transformation for Performance

```vhd
rule fetch_and_decode (!stallfunc(instr, bu)1);
  bu.enq1(newIt(instr, rf));
  pc <= predIa;
endrule

rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd, src1:.va, src2:.vb}:
      begin
        rf.upd0(rd, va+vb);
        bu.deq0();
      end
    tagged ESub{cond:.cv, addr:.av}:
      if (cv == 0) then begin
        pc <= av;
        bu.clear0();
      end;
    tagged ELoad{dst:.rd, addr:.av}:
      begin
        rf.upd0(rd, dMem.read(av));
        bu.deq0();
      end
    tagged EStore{value:.vv, addr:.av}:
      begin
        dMem.write(av, vv);
        bu.deq0();
      end
  endcase endrule
```

After Renaming

- Things will work
  - both rules can fire concurrently

Programmer Specifies:

\[ R_{execute} < R_{fetch} \]

Compiler Derives:

\[ (\text{first}^0, \text{deq}^0) < (\text{find}^1, \text{deq}^1) \]

What if the programmer wrote this?

\[ R_{execute} < R_{execute} < R_{fetch} < R_{fetch} \]

Outline

- Single module structure
- Modular structure issues

A Modular organization:

Recursive modules

- Modules call each other
  - bu part of Execute
  - rf and pc part of Fetch&Decode
  - fetch delivers decoded instructions to Execute
Recursive modular organization

module mkCPU2#(Mem iMem, Mem dMem)();
    Execute execute <- mkExecute(dMem, fetch);
    Fetch fetch <- mkFetch(iMem, execute);
endmodule

interface Fetch;
    method Action setPC (Iaddress cpc);
    method Action writeback (RName dst, Value v);
endinterface

interface Execute;
    method Action enqIt(InstTemplate it);
    method Bool stall(Instr instr)
endinterface
Issue

- A recursive call structure can be wrong in the sense of “circular calls”; fortunately the compiler can perform this check.
- Unfortunately recursive call structure amongst modules is not supported by the compiler.

So what should we do?

Connectable Methods

```haskell
interface Get#(data_T);
    ActionValue#(data_T) get();
endinterface

interface Put#(data_T);
    Action put(data_T x);
endinterface

module mkConnection#(Get#(data_T) m1, Put#(data_T) m2) ();
    rule stitch(True);
        data_T res <- m1.get();
        m2.put(res);
    endrule
endmodule
```

m1 and m2 are separately compilable

Connectable Organization

- Both ends completely separately compilable
  - bu still part of Execute
  - rf still part of Fetch&Decode

Can we automatically transform the recursive structure into this get-put structure?

Step 1: Break up Rules

only one recursive method call per rule ...

```haskell
rule exec_EAdd(it matches EAdd{dst:.rd, op1:.va, op2:.vb});
    fetch.writeback(rd, va+vb); bu.deq();
endrule

rule exec_EBz_Taken (it matches EBz{cond:.cv, addr:.av}) && cv == 0);
    fetch.setPC(av); bu.clear();
endrule

rule exec_EBz_Not Taken (it matches EBz{cond:.cv, addr:.av} && cv != 0);
    bu.deq();
endrule

rule exec_ELoad(it matches ELoad {dst:.rd, addr:.av});
    fetch.writeback(rd, dMem.read(av)); bu.deq();
endrule

rule exec_EStore(it matches EStore{value:.vv, addr:.av};
    dMem.write(av,vv); bu.deq();
endrule
```

Don’t need to change these rules
Step 2: Change a rule to a method

```plaintext
rule exec_EBz_Taken (it matches EBz{cond:.cv, addr:.av})
  && cv == 0);
  fetch.setPC(av); bu.clear();
endrule

method ActionValue#(IAddress) getNewPC()
  if ((it matches EBz{cond:.cv, addr:.av} &&& (cv == 0));
  bu.clear();
  return(av);
endmethod
```

Instead of sending av to the fetch module, we are simply providing av to the outside world under suitable conditions.

Step 2: Merging multiple rules into one method  

```plaintext
rule exec_EAdd(it matches EAdd{dst:.rd, op1:.va, op2:.vb});
  fetch.writeback(rd, va + vb); bu.deq();
endrule

rule exec_ELoad(it matches ELoad {dst:.rd, addr:.av});
  fetch.writeback(rd, dMem.get(av)); bu.deq();
endrule
```

Need to combine all calls to fetch.writeback into one method!

```plaintext
method Tuple2#(RName, Value) getWriteback() if (canDoWB);
  bu.deq();
  case (it) matches
    tagged EAdd {dst:.rd, op1:.va, op2:.vb}:
      return(tuple2(rd, va+vb));
    tagged ELoad{dst:.rd, addr:.av}:
      return(tuple2(rd, dMem.get(av)));
    default: return (?); // should never occur
  endcase
endmethod
```

canDoWB means (it) matches “EAdd or ELoad”

---

Step-1 is not always possible: Jump&Link instruction

```plaintext
rule exec_EJAL(it matches EJAL{rd:.rd, pc:.pc, addr:.av};
  fetch.writeback(rd, pc);
  fetch.setPC(av); bu.clear();
endrule
```

RWire to the rescue

1. Create an RWire for each method
2. Replace calls with RWire writes
3. Connect methods to RWire reads
4. Restrict schedule to maintain atomicity

Using RWires

```plaintext
rule exec_EBz_Taken (it matches EBz{cond:.cv, addr:.av})
  && cv == 0);
  PC_wire.wset(av); bu.clear();
endrule
```

```plaintext
method ActionValue#(IAddress) getNewPC()
  if (PC_wire.wget matches tagged Valid .x);
  return(x);
endmethod
```

Dangerous -- if the outsider does not pick up the value, it is gone!

Reading and writing of a wire is not an atomic action
Jump&Link using RWires

steps 1 & 2

Rwire#(Tuple2#(RName, Value))  wb_wire <- mkRWire();
Rwire#(Iaddress)            getPC_wire <- mkRWire();

rule exec_EJAL(it matches EJAL{rd:.rd, pc: .pc, addr:.av};
   wb_wire.wset(tuple2(rd, pc));
   getPC_wire.wset(av); bu.clear();
endrule

rule exec_EAdd(it matches EAdd{dst:.rd, op1:.va, op2:.vb});
   wb_wire.wset(tuple2(rd, va + vb)); bu.deq();
endrule

rule exec_EBz_Taken(it matches EBz{cond:.cv, addr:.av}
   && cv == 0);
   getPC_wire.wset(av); bu.clear();
endrule

rule exec_ELoad(it matches ELoad {dst:.rd, addr:.av});
   wb_wire.wset(tuple2(rd, dMem.get(av))); bu.deq();
endrule

Jump&Link Connectable Version

step 3

method ActionValue#(...) writeback_get()
   if (wb_wire.wget() matches tagged Valid .x);
      return x;
endmethod

method ActionValue#(Iaddress) setPC_get()
   if (getPC_wire.wget() matches tagged Valid .x);
      return x;
endmethod

Atomicity violations?
1. dropped values on RWires
2. Get-Put rule is no longer a single atomic action

My recommendation

- If recursive modules are the natural way to express a design – do that first
- Transform it by turning some rules into methods
- Sometimes EHRs and bypass FIFO can solve the problem (we have not shown you this)
- If all fails consult the staff

Modular Structure

- Different modular structures generate the “same hardware”
  - modular structure choice is more about design convenience
- Recursive modular organizations are natural but
  - there are some theoretical complications
- Transforming a recursive structure into a non-recursive one is always possible using RWires
  - but prides avenues for abuse