Bluespec-5: Modeling Processors
(revised after the lecture)

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Based on material prepared by Bluespec Inc, January 2005

Some New Types

- Enumerations
  - Sets of symbolic names
- Structs
  - Records with fields
- Tagged Unions
  - Unions, made “type-safe” with tags
Enumeration

```c
typedef enum {Red; Green; Blue} Color;
    Red = 00, Green = 01, Blue = 10

typedef enum {Waiting; Running; Done} State;
    Waiting = 00, Running = 01, Done = 10

typedef enum {R0;R1;R2;R3} RName;
    R0 = 00, R1 = 01, R2 = 10, R3 = 11
```

Enumerations define new, distinct types:
- Even though, of course, they are represented as bit vectors

Type safety
- Type checking guarantees that bit-vectors are consistently interpreted.
- A 2-bit vector which is used as a Color in one place, cannot accidentally be used as a State in another location:

```c
Reg#(Color) c();
Reg#(State) s();
...
    s <= c;
```
Structs

typedef Bool FP_Sign;
typedef Bit#(2) FP_RS;

typedef struct {
    FP_Sign sign; // sign bit
    Bit#(ee) exp; // exponent
    Bit#(ss) sfd; // significand
    FP_RS rs; // round and sticky bit
} FP_I#(type ee, type ss);

// exponent and significand sizes are
// *numeric* type parameters

Bit interpretation of structs

<table>
<thead>
<tr>
<th>sign</th>
<th>sfd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ee</td>
</tr>
<tr>
<td></td>
<td>ss</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

exp rs
Tagged Unions

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;
} Instr deriving(Bits, Eq);

The Maybe type

- The Maybe type can be regarded as a value together with a “valid” bit

  typedef union tagged {
    void Invalid;
    t Valid;
} Maybe#(type t) deriving(Eq,Bits);

- Example: a function that looks up a name in a telephone directory can have a return type Maybe#(TelNum)
  - If the name is not present in the directory it returns tagged Invalid
  - If the name is present with number x, it returns tagged Valid x
The Maybe type

The `isValid(m)` function
- returns `True` if `m` is tagged `Valid x`
- returns `False` if `m` is tagged `Invalid`

The `fromMaybe(y, m)` function
- returns `x` if `m` is tagged `Valid x`
- returns `y` if `m` is tagged `Invalid`

Deriving

When defining new types, by attaching a “deriving” clause to the type definition, we let the compiler automatically create the “natural” definition of certain operations on the type.

```c
typedef struct { ... } Foo
deriving (Eq);
```

Automatically generates the “==” and “!=” operations on the type.
Deriving Bits

typedef struct { … } Foo
deriving (Bits);

- Automatically generates the “pack” and “unpack” operations on the type (simple concatenation of bit representations of components)
- This is necessary, for example, if the type is going to be stored in a register, fifo, or other element that demands that the content type be in the Bits typeclass
  - (there are many types that may be used only during static elaboration or as intermediate values that do not need to be in typeclass Bits)
- It is possible to customize the pack/unpack operations to any specific desired representation

Pattern-matching

- Pattern-matching is a more readable way to:
  - test data for particular structure and content
  - extract data from a data structure, by binding “pattern variables” (.variable) to components

```markdown
    case (m) matches
        tagged Invalid  : return 0;
        tagged Valid .x : return x;
    endcase

    if (m matches (Valid .x) &&& (x > 10))
        ...
```

- The &&& is a conjunction, and allows pattern-variables to come into scope from left to right
Example:

A type for “cpu instruction operands”

typedef union tagged {
    bit [4:0] Register;
    bit [21:0] Literal;
    struct {
        bit [4:0] regAddr; bit [4:0] regIndex;
    } Indexed;
} InstrOperand;

case (oprand) matches
tagged Register .r : x = rf[r];
tagged Literal .n : x = n;
tagged Indexed { .ra, .ri } : begin
    Iaddress a = rf[ra]+rf[ri];
    x = mem.get(a);
end

Other types in BSV

String
- Character strings
Action
- What rules/interface methods do
Rule
- Behavior inside modules
Interface
- External view of module behavior

Useful during static elaboration
Processors

- Unpipelined processor
- Two-stage pipeline
- Bypass FIFO (next lecture)
- Five-stage pipeline (next lecture)

Instruction set

define enum {R0; R1; R2; ...; R31} RName;
define 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
}
    Instr deriving(Bits, Eq);
define Bit#(32) Iaddress;
define Bit#(32) Daddress;
define Bit#(32) Value;

An instruction set can be implemented using many different microarchitectures
Non-pipelined Processor

module mkCPU#(Mem iMem, Mem dMem)(Empty);
    Reg#(Iaddress) pc <- mkReg(0);
    RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
    Iaddress i32 = iMem.get(pc);
    Instr instr = unpack(i32[16:0]);
    Iaddress predIa = pc + 1;
    rule fetch_Execute ...
endmodule

Non-pipelined processor rule

rule fetch_Execute (True);
    case (instr) matches
        tagged Add {dst:.rd, src1:.ra, src2:.rb}: begin
            rf.upd(rd, rf[ra]+rf[rb]);
            pc <= predIa;
        end
        tagged Bz {cond:.rc, addr:.ra}: begin
            pc <= (rf[rc]==0) ? rf[ra] : predIa;
        end
        tagged Load {dest:.rd, addr:.ra}: begin
            rf.upd(rd, dMem.get(rf[ra]));
            pc <= predIa;
        end
        tagged Store {value:.rv, addr:.ra}: begin
            dMem.put(rf[ra], rf[rv]);
            pc <= predIa;
        end
    endcase
eンドrule
Processor Pipelines and FIFOs

CPU

iMem

pc

fetch

decode

execute

memory

write-back

dMem

Processor Pipelines and FIFOs

SFIFO (glue between stages)

interface SFIFO#(type t, type tr);
method Action enq(t); // enqueue an item
method Action deq(); // remove oldest entry
method t first(); // inspect oldest item
method Action clear(); // make FIFO empty
method Bool find(tr); // search FIFO
endinterface

n = # of bits needed to represent the values of type "t"
m = # of bits needed to represent the values of type "tr"
Two-Stage Pipeline

```
module mkCPU#(Mem iMem, Mem dMem)(Empty);
    Reg#(Iaddress) pc <- mkReg(0);
    RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
    SFIFO#(Tuple2#(Iaddress, InstTemplate)) bu <- mkSFifo(findf);
    Iaddress i32 = iMem.get(pc);
    Instr instr = unpack(i32[16:0]);
    Iaddress predIa = pc + 1;
    match{.ipc, .it} = bu.first;
    rule fetch_decode ...
endmodule
```

Instruction Template

```haskell
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 dst; Daddress addr} Store;
} InstTemplate deriving(Eq, Bits);
```

decoded instruction with operands

```haskell
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;
} InstTemplate deriving(Eq, Bits);
```
Rules for Add

rule decodeAdd (instr matches Add{.rd,.ra,.rb})
    bu.enq (tuple2(pc, EAdd{rd, rf[ra], rf[rb]}));
    pc <= predIa;
endrule

rule executeAdd (it matches EAdd{.rd,.va,.vb})
    rf.upd(rd, va + vb);
    bu.deq();
endrule

Fetch & Decode Rule: Reexamined

Wrong! Because instructions in bu may be modifying ra or rb

stall!
Fetch & Decode Rule: corrected

rule decodeAdd (instr matches Add{.rd,.ra,.rb} &&&
  !bu.find(ra) &&& !bu.find(rb))
  bu.enq (tuple2(pc, EAdd{rd, rf[ra], rf[rb]}));
  pc <= predIa;
endrule

Rules for Branch

rule decodeBz (instr matches Bz{.rc,.addr}) &&&
  !bu.find(rc) &&& !bu.find(addr));
  bu.enq (tuple2(pc, EBz{rf[rc],rf[addr]}));
  pc <= predIa;
endrule

rule bzTaken (it matches EBz{.vc,.va}) &&& (vc == 0));
  pc <= va; bu.clear(); endrule
rule bzNotTaken (it matches EBz{.vc,.va}) &&& (vc != 0));
  bu.deq; endrule
### The Stall Signal

```plaintext
Bool stall =
case (instr) matches
tagged Add {.rd,.ra,.rb}: return (bu.find(ra) || bu.find(rb));
tagged Bz {.rc,.addr}: return (bu.find(rc) || bu.find(addr));
tagged Load {.rd,.addr}: return (bu.find(addr));
tagged Store {.v,.addr}: return (bu.find(v)) || bu.find(addr));
endcase;
```

```plaintext
function Bool findf (RName r,
  Tuple2#(Iaddress, InstrTemplate) tup);
case (snd(tup)) matches
tagged EAdd{.rd,.ra,.rb}: return (r == rd);
tagged EBz {.c,.a}:       return (False);
tagged ELoad{.rd,.a}:     return (r == rd);
tagged EStore{.v,.a}:     return (False);
endcase
endfunction
```

Need to extend the fifo interface with the “find” method where “find” searches the fifo using the findf function.

```plaintext
SFIFO#(Tuple2(Iaddress, InstrTemplate)) bu <- mkFifo(findf)
```

---

### Fetch & Decode Rule

```plaintext
rule fetch_and_decode(!stall);
case (instr) matches
tagged Add {.rd,.ra,.rb}:
  bu.enq(tuple2(pc,EAdd{dst:rd,op1:rf[ra],op2:rf[rb]}));
tagged Bz {.rc,.addr}:
  bu.enq(tuple2(pc,EBz{cond:rf[rc],addr:rf[addr]}));
tagged Load {.rd,.addr}:
  bu.enq(tuple2(pc,ELoad{dst:rd,addr:rf[addr]}));
tagged Store{.v,.addr}:
  bu.enq(tuple2(pc,EStore{value:rf[v],addr:rf[addr]}));
endcase
pc<= predIa;
endrule
```
Fetch & Decode Rule

another style

InstrTemplate newIt =
  case (instr) matches
    tagged Add { .rd, .ra, .rb}:
      return EAdd{dst:rd,op1:rf[ra],op2:rf[rb]};
    tagged Bz { .rc, .addr}:
      return EBz{cond:rf[rc],addr:rf[addr]};
    tagged Load { .rd, .addr}:
      return ELoad{dst:rd,addr:rf[addr]};
    tagged Store{.v,.addr}:
      return EStore{value:rf[v],addr:rf[addr]};
  endcase;

rule fetch_and_decode (!stall);
  bu.enq(tuple2(pc, newIt));
  pc <= predIa;
endrule

Execute Rule

rule execute_rule(True);
  case (it) matches
    tagged EAdd{.rd,.va,.vb}:
      begin
        rf[rd] <= (va + vb);
        bu.deq();
      end
    tagged EBz{.cv,.av}:
      if (cv == 0) then begin
        pc <= av;
        bu.clear();
        else bu.deq();
    tagged ELoad{.rd,.av}:
      begin
        rf[rd] <= dMem[av];
        bu.deq();
      end
    tagged EStore{.vv,.av}:
      begin
        dMem[av] <= vv;
        bu.deq();
      end
  endcase
endrule
Modular organization

Method calls embody both data and control (i.e., protocol)

Modularizing Two-Stage Pipeline

```verilog
module mkCPU#(Mem iMem, Mem dMem)(Empty);
    RegFile#(RName, Bit#(32)) rf <- mkRegFileFull();
    SFIFO#(Tuple2(Iaddress, InstTemplate)) bu <- mkSFifo(findf);
    Fetch fetch <- mkFetch(iMem, bu, rf);
    Empty exec <- mkExecute(dMem, bu, rf, fetch);
endmodule

interface Fetch;
    method Action setPC(Iaddress x);
endinterface
```
Fetch & Decode Module

module mkFetch(Mem dMem, SFIFO#(Tuple2(Iaddress, InstTemplate)) bu, RegFile#(RName, Bit#(32)) rf)(Fetch);
  Reg#(Iaddress) pc <- mkReg(0);
  InstrTemplate newIt =
    case (instr) matches
      tagged Add {.rd,.ra,.rb}:
        return EAdd{dst:rd,op1:rf[ra],op2:rf[rb]};
      tagged Bz {.rc,.addr}:
        return EBz{cond:rf[rc],addr:rf[addr]};
      tagged Load {.rd,.addr}:
        return ELoad{dst:rd,addr:rf[addr]};
      tagged Store{.v,.addr}:
        return EStore{value:rf[v],addr:rf[addr]};
    endcase;
  rule fetch_and_decode (!stall);
    bu.enq(tuple2(pc, newIt));   pc <= predIa;
  endrule
  method Action setPC(Iaddress ia);
    pc <= ia;
  endmethod
endmodule

Execute Module

module mkExecute#(Mem dMem, SFIFO#(Tuple2(Iaddress, InstTemplate) bu, RegFile#(RName, Bit#(32)) rf, Fetch fetch)(Empty);
  rule execute_rule (True);
    case (it) matches
      tagged EAdd{.rd,.va,.vb}:
        begin
          rf.upd(rd, va + vb); bu.deq();
        end
      tagged EBz {.cv,.av}:
        if (cv == 0) then begin
          fetch.setPC(av); bu.clear(); end
        else bu.deq();
      tagged ELoad{.rd,.av}:
        begin
          rf[rd] <= dMem[av]; bu.deq();
        end
      tagged EStore{.vv,.av}:
        begin
          dMem[av] <= vv; bu.deq();
        end
    endcase
  endrule
endmodule
Is this a good modular organization?

- Separately compilable?
- Separately refinable?
- Good for verification?
- Physical properties:
  - Few connecting wires?
  - ...
- ...

Next time

- Bypassing issues
- Designing the FIFO