

# 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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-1

## Some New Types

- ◆ Enumerations
  - Sets of symbolic names
- ◆ Structs
  - Records with fields
- ◆ Tagged Unions
  - unions, made "type-safe" with tags

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-2

# Enumeration

```
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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-3

# 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:

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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-4

# Structs

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

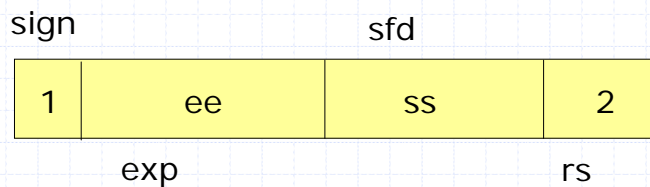
typedef struct {
    FP_Sign  sign; // sign bit
    Bit#(ee) exp; // exponent
    Bit#(ss) sfd; // significand
    FP_RS    rs;   // round and sticky bit
} FP_I#(type ee, type ss);
// exponent and significand sizes are
// *numeric* type parameters
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-5

# Bit interpretation of structs



March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-6

# 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);
```

00	dst	src1	src2
01		cond	addr
10		dst	addr
11		value	addr

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-7

# 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**

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-8

## 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`

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-9

## 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

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

- ◆ Automatically generates the “==” and “!=” operations on the type

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-10

## 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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-11

## 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

```
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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-12

## 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;
```

rf[r] means rf.sub(r)

```
case (operand) 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
endcase
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-13

## 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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-14

## Slide 13

---

a1

eans

arvind, 3/9/2005



# Processors

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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-15

# Instruction set

```
typedef enum {R0;R1;R2;...;R31} RName;
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);

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

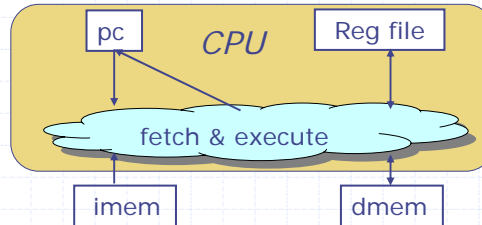
An instruction set can be implemented using many different microarchitectures

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-16

## Non-pipelined Pipeline



```
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
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-17

## Non-pipelined processor rule

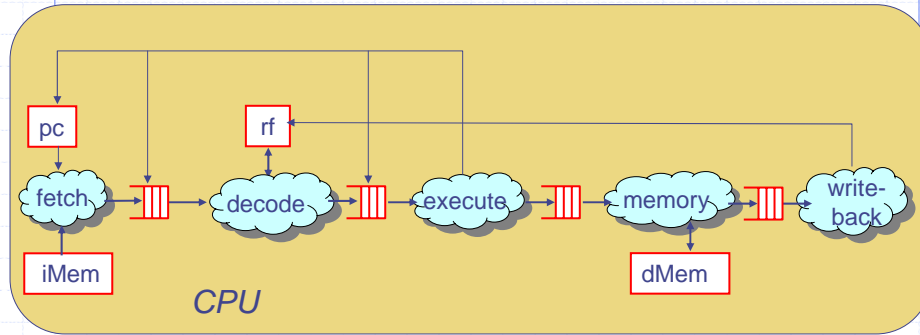
```
rule fetch_Execute (True);
  case (instr) matches
    tagged Add {dst:.rd,src1:.ra,src2:.rb}: begin
      rf.upd(rd, rf[ra]+rf[rb]);
      pc <= predIa
    end
    tagged Bz {cond:.rc,addr:.ra}: begin
      pc <= (rf[rc]==0) ? rf[ra] : predIa;
    end
    tagged Load {dest:.rd,addr:.ra}: begin
      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
  end
endcase
endrule
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-18

# Processor Pipelines and FIFOs



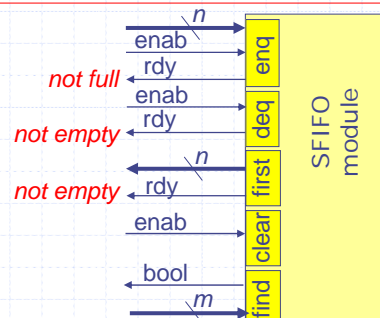
March 9, 2005

<http://csg.csaill.mit.edu/6.884/>

L12-19

# 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"

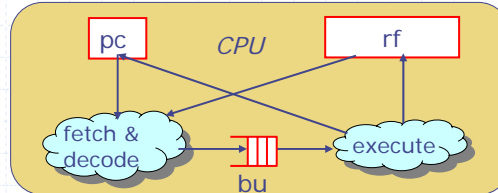
$m$  = # of bits needed to represent the values of type "tr"

March 9, 2005

<http://csg.csaill.mit.edu/6.884/>

L12-20

## 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 ...

```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-21

## Instruction Template

```

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);

```

decoded instruction with operands

```

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);

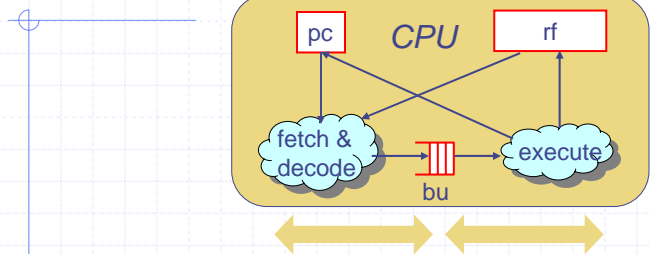
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-22

# Rules for Add



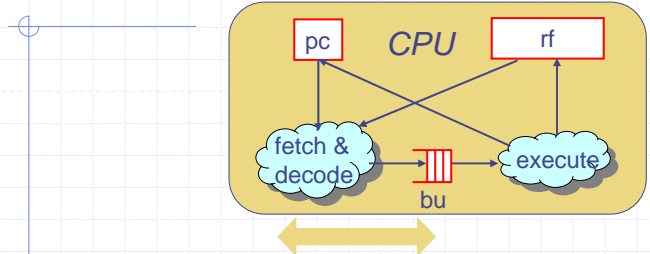
implicit checks:  
bu notfull

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

bu notempty

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

# Fetch & Decode Rule: *Reexamined*

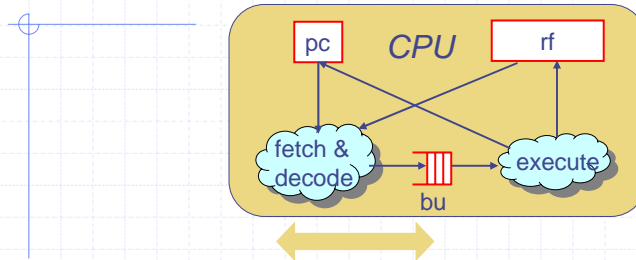


```
rule decodeAdd (instr matches Add{.rd,.ra,.rb})
  bu.enq (tuple2(pc, EAdd{rd, rf[ra], rf[rb]}));
  pc <= predIa;
endrule
```

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

stall !

## Fetch & Decode Rule: *corrected*



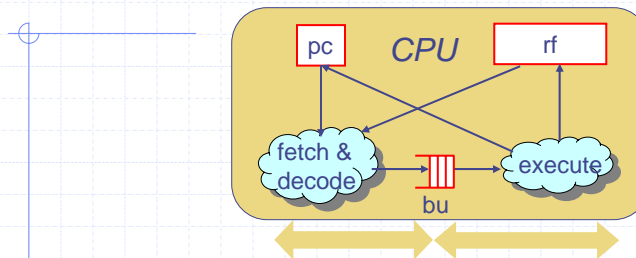
```
rule decodeAdd (instr matches Add{.rd,.ra,.rb} &&&
    !bu.find(ra) &&& !bu.find(rb))
  bu.enq (tuple2(pc, EAdd{rd, rf[ra], rf[rb]}));
  pc <= predIa;
endrule
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-25

## Rules for Branch



*rule-atomicity ensures that pc update, and discard of pre-fetched instrs in bu, are done consistently*

```
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
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-26

# The Stall Signal

```
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;
```

```
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

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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-27

# Fetch & Decode Rule

```
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
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-28

# 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
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-29

# 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();
    end
    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
```

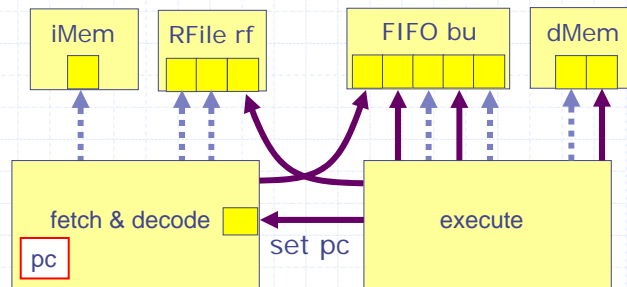
March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-30



## Modular organization



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



March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-31

## Modularizing Two-Stage Pipeline

```

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
    
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-32

## 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
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-33

## 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
```

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-34

## Is this a good modular organization?

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

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-35

## Next time

- ◆ Bypassing issues
- ◆ Designing the FIFO

March 9, 2005

<http://csg.csail.mit.edu/6.884/>

L12-36