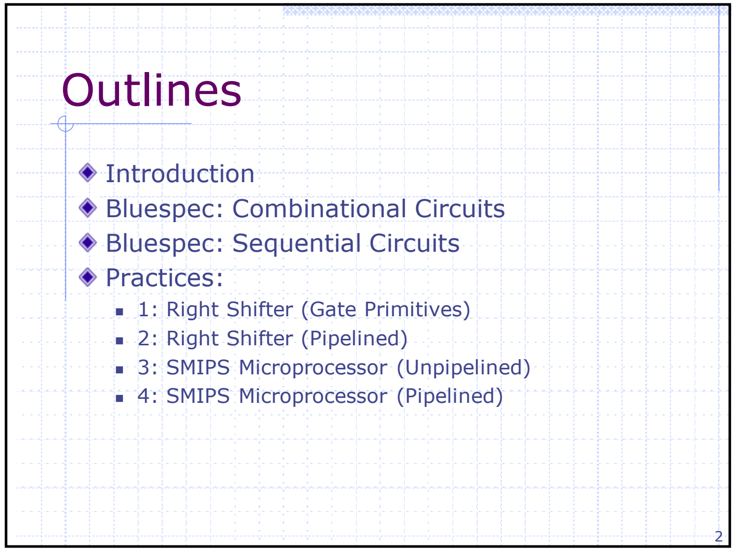
计算机系统结构实验课

JSI

Slides taken (with permission) from:
Derek Chiou (UT Austin)
Arvind and collaborators (MIT)

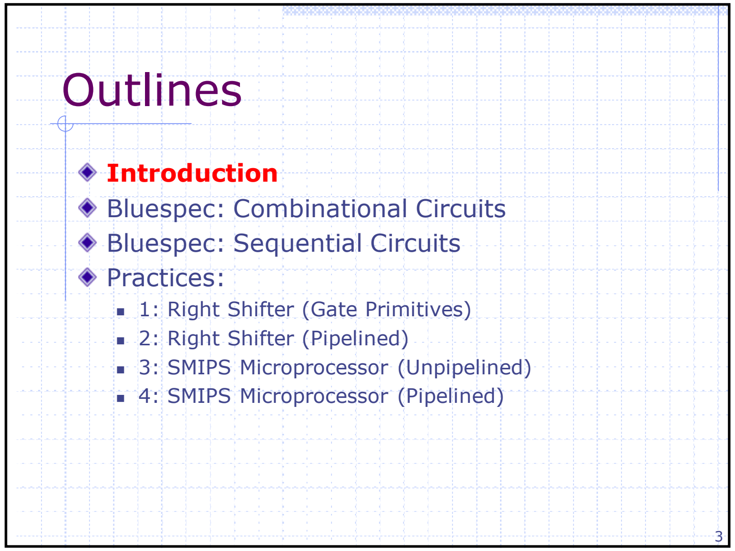
1



Outlines

- Introduction
- Bluespec: Combinational Circuits
- Bluespec: Sequential Circuits
- Practices:
  - 1: Right Shifter (Gate Primitives)
  - 2: Right Shifter (Pipelined)
  - 3: SMIPS Microprocessor (Unpipelined)
  - 4: SMIPS Microprocessor (Pipelined)

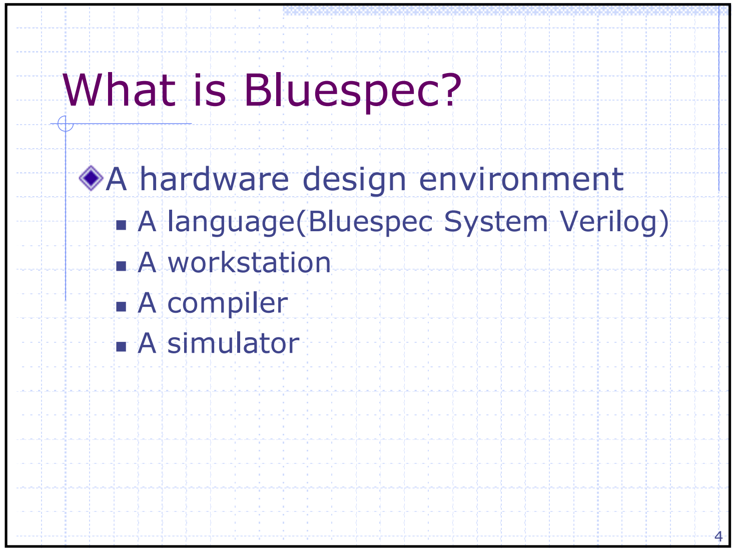
2



Outlines

- **Introduction**
- Bluespec: Combinational Circuits
- Bluespec: Sequential Circuits
- Practices:
  - 1: Right Shifter (Gate Primitives)
  - 2: Right Shifter (Pipelined)
  - 3: SMIPS Microprocessor (Unpipelined)
  - 4: SMIPS Microprocessor (Pipelined)

3



What is Bluespec?

- A hardware design environment
  - A language(Bluespec System Verilog)
  - A workstation
  - A compiler
  - A simulator

4

Practice Environment

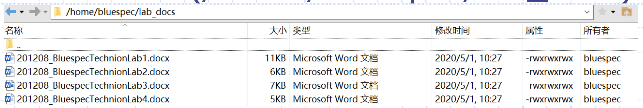
- Lab Server (BUAA network):
  - IP Address: 10.254.52.1
  - SSH Port: 22
  - Username: your student ID (e.g., BY2006118)
  - Password: your student ID (e.g., BY2006118)

  Change the password after you login in.

Note that Your account is created temporarily and shall be deleted at the end of this course, so don’t leave important data there.

Practice Materials

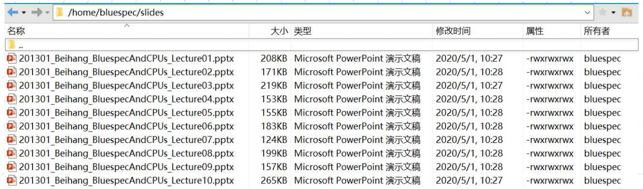
- Lab Docs (/home/bluespec/lab_docs)



/home/bluespec/lab_docs

名称	大小	类型	修改时间	属性	所有者
..					
201208_BluespecTechnionLab1.docx	11KB	Microsoft Word 文档	2020/5/1, 10:27	-rwxrwxrwx	bluespec
201208_BluespecTechnionLab2.docx	6KB	Microsoft Word 文档	2020/5/1, 10:27	-rwxrwxrwx	bluespec
201208_BluespecTechnionLab3.docx	7KB	Microsoft Word 文档	2020/5/1, 10:27	-rwxrwxrwx	bluespec
201208_BluespecTechnionLab4.docx	5KB	Microsoft Word 文档	2020/5/1, 10:27	-rwxrwxrwx	bluespec

- Slides (/home/bluespec/slides)



/home/bluespec/slides

名称	大小	类型	修改时间	属性	所有者
..					
201301_Beihang_BluespecAndCPUs_Lecture01.pptx	208KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:27	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture02.pptx	171KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:28	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture03.pptx	219KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:27	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture04.pptx	153KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:28	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture05.pptx	155KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:28	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture06.pptx	183KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:28	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture07.pptx	124KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:28	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture08.pptx	199KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:28	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture09.pptx	157KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:28	-rwxrwxrwx	bluespec
201301_Beihang_BluespecAndCPUs_Lecture10.pptx	265KB	Microsoft PowerPoint 演示文稿	2020/5/1, 10:27	-rwxrwxrwx	bluespec

License Server

- Open file ~/.bashrc
- Set the Environment Variable

export BLUESPEC_HOME=/tools/bluespec

export BLUESPECDIR=$BLUESPEC_HOME/lib

export PATH=$PATH:$BLUESPEC_HOME/bin

export LM_LICENSE_FILE=/home/bluespec/license/Bluespec_20210518.lic

- source ~/.bashrc

HOWTO

1. Copy the lab_template folder to your own dir.

```
[BY2006118@cu07 ~]$ cp –r /home/bluespec/lab_template/ ./
```

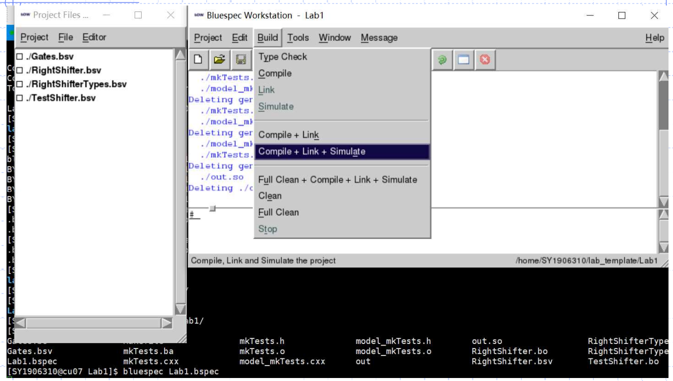
2. Untar the Lab.tgz files (Lab1 as an example).

```
[BY2006118@cu07 ~]$ cd lab_template/
[BY2006118@cu07 lab_template]$ tar -xvf Lab1.tgz
```

3. Enter the directory and run the spec file.

```
[BY2006118@cu07 lab_template]$ cd Lab1/
[BY2006118@cu07 Lab1]$ bluespec Lab1.bspec
```

Lab1 Example

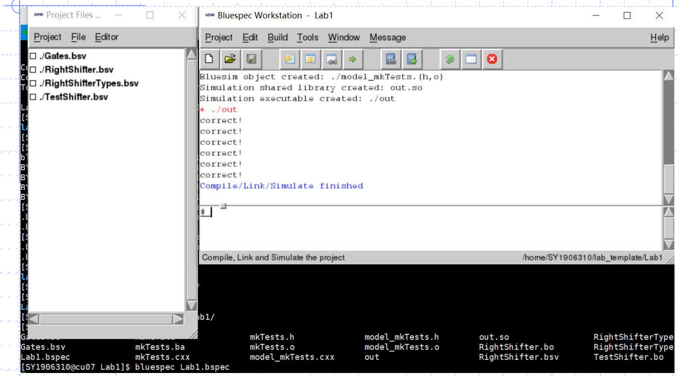

Project Files ...
Project File Editor
□ ./Gates.bsv
□ ./RightShifter.bsv
□ ./RightShifterTypes.bsv
□ ./TestShifter.bsv
Bluespec Workstation - Lab1
Project Edit Build Tools Window Message Help
Type Check
Compile
Link
Simulate
Compile + Link
Compile + Link + Simulate
Full Clean + Compile + Link + Simulate
Clean
Full Clean
Stop
Compile, Link and Simulate the project
/home/SY1906310/lab_template/Lab1

```
  ./mkTests.
  ./model_mk
Deleting ger
  ./mkTests.
  ./model_mk
Deleting ger
  ./model_mk
  ./mkTests.
Deleting ger
  ./out.so
Deleting ./c
```

```
Gates.bsv           mkTests.ba          mkTests.o           model_mkTests.o     RightShifter.bo     RightShifterType
Lab1.bspec          mkTests.cxx         model_mkTests.cxx   out                 RightShifter.bsv    TestShifter.bo
[SY1906310@cu07 Lab1]$ bluespec Lab1.bspec
```

9

Lab1 Example


Project Files ...
Project File Editor
□ ./Gates.bsv
□ ./RightShifter.bsv
□ ./RightShifterTypes.bsv
□ ./TestShifter.bsv
Bluespec Workstation - Lab1
Project Edit Build Tools Window Message Help
Compile, Link and Simulate the project
/home/SY1906310/lab_template/Lab1

```
Bluesim object created: ./model_mkTests.{h,o}
Simulation shared library created: out.so
Simulation executable created: ./out
+ ./out
correct!
correct!
correct!
correct!
correct!
correct!
Compile/Link/Simulate finished
```

```
                                        mkTests.h           model_mkTests.h     out.so              RightShifterType
Gates.bsv           mkTests.ba          mkTests.o           model_mkTests.o     RightShifter.bo     RightShifterType
Lab1.bspec          mkTests.cxx         model_mkTests.cxx   out                 RightShifter.bsv    TestShifter.bo
[SY1906310@cu07 Lab1]$ bluespec Lab1.bspec
```

10

Outlines

- Introduction
- **Bluespec: Combinational Circuits**
- Bluespec: Sequential Circuits
- Practices:
  - 1: Right Shifter (Gate Primitives)
  - 2: Right Shifter (Pipelined)
  - 3: SMIPS Microprocessor (Unpipelined)
  - 4: SMIPS Microprocessor (Pipelined)

11

Content

- Design of a combinational ALU starting with primitive gates And, Or and Not
- Combinational circuits as acyclic wiring diagrams of primitive gates
- Introduction to BSV
  - Intro to types – enum, typedefs, numeric types, int#(32) vs integer, bool vs bit#(1), vectors
  - Simple operations: concatenation, conditionals, loops
  - Functions
  - Static elaboration and a structural interpretation of the textual code

12

Combinational circuits are acyclic interconnections of gates

- And, Or, Not
- Nand, Nor, Xor
- …

Simple combinational circuits:

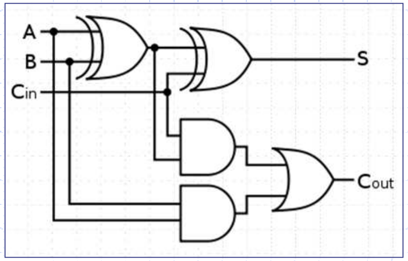
Ripple-carry Adder

Full Adder: A one-bit adder

```
function fa(a, b, c_in);
   s = (a ^ b)^ c_in;
   c_out = (a & b) | (c_in & (a ^ b));
   return {c_out,s};
endfunction
```

Structural code – only specifies interconnection between boxes

Not quite correct – needs type annotations



Full Adder: A one-bit adder

corrected

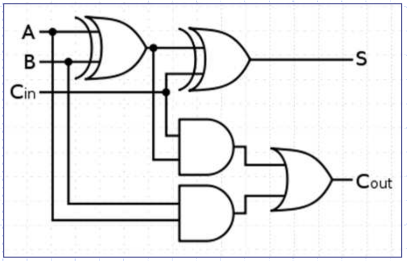
```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b,
                                 Bit#(1) c_in);
   Bit#(1) s = (a ^ b)^ c_in;
   Bit#(1) c_out = (a & b) | (c_in & (a ^ b));
   return {c_out,s};
endfunction
```

"`Bit#(1) a`" type declaration says that `a` is one bit wide

`{c_out,s}` represents bit concatenation

How big is `{c_out,s}`?

2 bits



Types

- A type is a grouping of values
  - Integer: `1, 2, 3, …`
  - Bool: `True, False`
  - Bit: `0,1`
  - A pair of Integers: `Tuple2#(Integer, Integer)`
  - A function `fname` from Integers to Integers:

```
function Integer fname (Integer arg)
```

- Every expression and variable in a Bluespec program has a type; sometimes it is specified explicitly and sometimes it is deduced by the compiler
- Thus we say an expression has a type or belongs to a type

Each expression has a unique type

Type declaration versus deduction

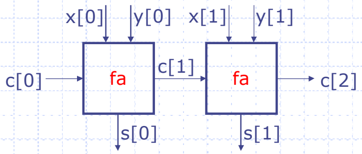
- The programmer writes down types of some expressions in a program and the compiler deduces the types of the rest of expressions
- If the type deduction cannot be performed or the type declarations are inconsistent then the compiler complains

```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b,
                                   Bit#(1) c_in);
   Bit#(1) s = (a ^ b)^ c_in;
   Bit#(2) c_out = (a & b) | (c_in & (a ^ b));
   return {c_out,s};
endfunction
```

type error

Type checking prevents lots of silly mistakes

2-bit Ripple-Carry Adder



Fa can be used as a black box as long as we understand its type signature and its function

```
function Bit#(3) add(Bit#(2) x, Bit#(2) y,
                                 Bit#(1) c0);
   Bit#(2) s = 0;    Bit#(3) c=0; c[0] = c0;
   let cs0 = fa(x[0], y[0], c[0]);
             c[1] = cs0[1];  s[0] = cs0[0];
   let cs1 = fa(x[1], y[1], c[1]);
             c[2] = cs1[1];  s[1] = cs1[0];
  return {c[2],s};
endfunction
```

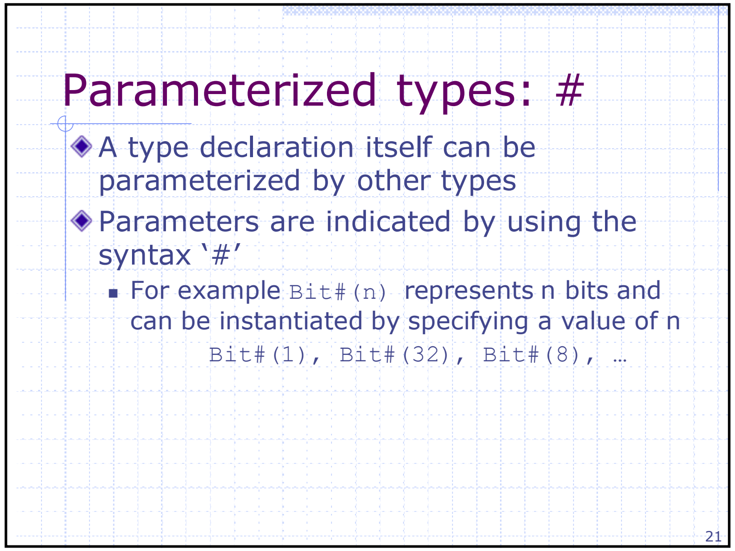
The "let" syntax avoids having to write down types explicitly

"let" syntax

- The "let" syntax: asks compiler to infer type
  - avoids having to write down types explicitly

```
let cs0 = fa(x[0], y[0], c[0]);
Bits#(2) cs0 = fa(x[0], y[0], c[0]);
```

The same



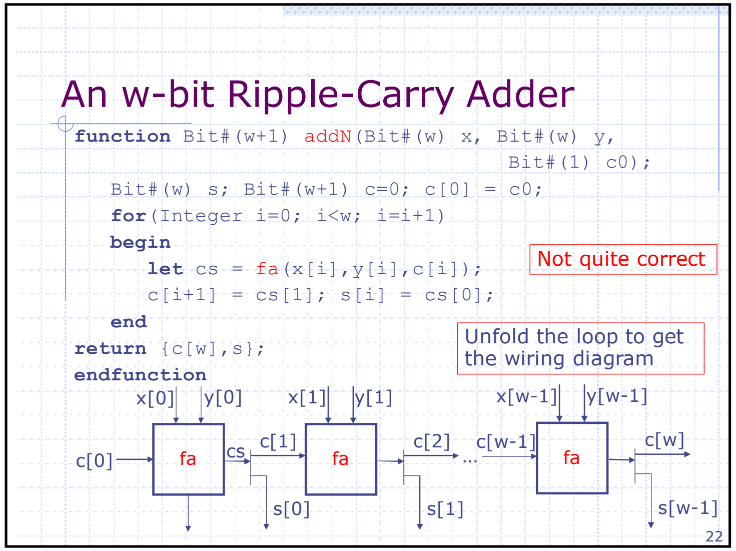
Parameterized types:

- A type declaration itself can be parameterized by other types
- Parameters are indicated by using the syntax '#'
  - For example `Bit#(n)` represents n bits and can be instantiated by specifying a value of n

    `Bit#(1), Bit#(32), Bit#(8), …`

21

21



An w-bit Ripple-Carry Adder

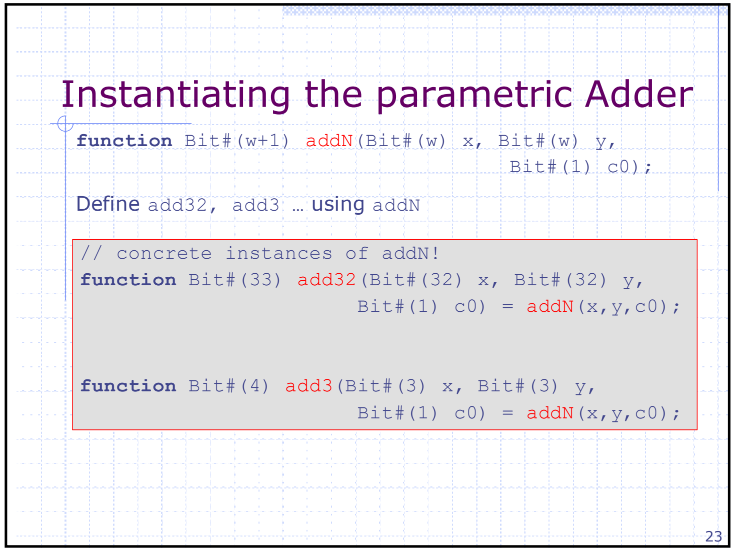
```
function Bit#(w+1) addN(Bit#(w) x, Bit#(w) y,
                        Bit#(1) c0);
   Bit#(w) s; Bit#(w+1) c=0; c[0] = c0;
   for(Integer i=0; i<w; i=i+1)
   begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
   end
return {c[w],s};
endfunction
```

Not quite correct

Unfold the loop to get the wiring diagram

22

22



Instantiating the parametric Adder

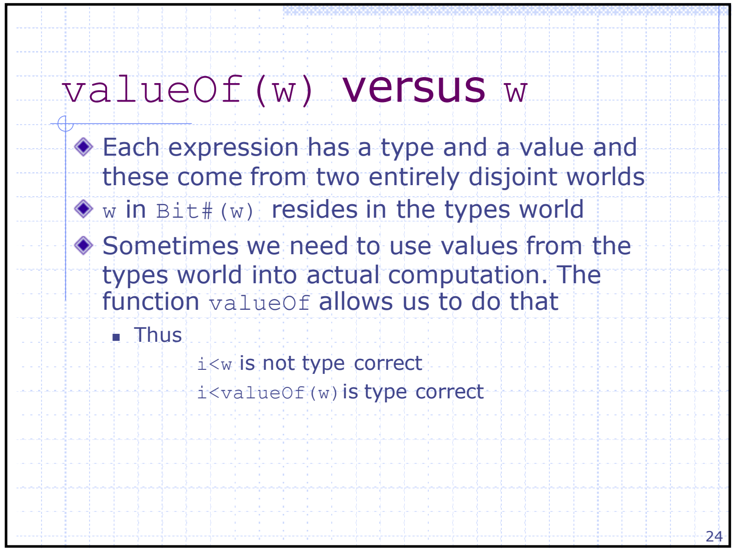
```
function Bit#(w+1) addN(Bit#(w) x, Bit#(w) y,
                        Bit#(1) c0);
```

Define `add32, add3` … using `addN`

```
// concrete instances of addN!
function Bit#(33) add32(Bit#(32) x, Bit#(32) y,
                        Bit#(1) c0) = addN(x,y,c0);


function Bit#(4) add3(Bit#(3) x, Bit#(3) y,
                      Bit#(1) c0) = addN(x,y,c0);
```

23

23



`valueOf(w)` versus `w`

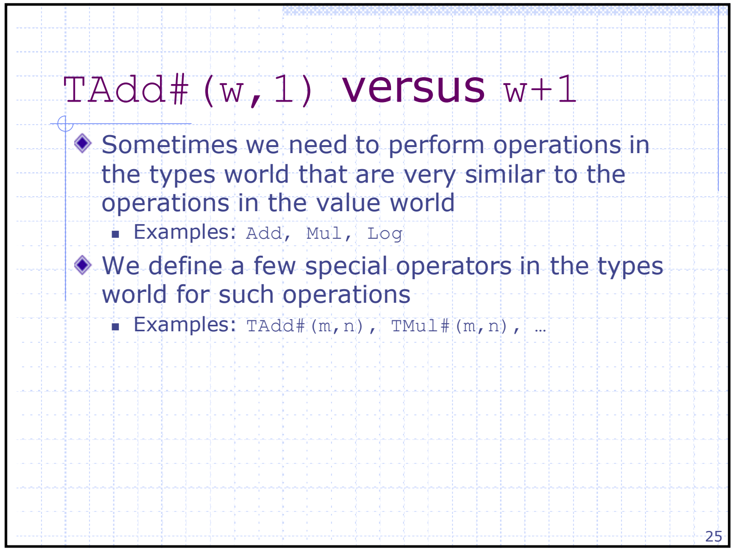
- Each expression has a type and a value and these come from two entirely disjoint worlds
- `w` in `Bit#(w)` resides in the types world
- Sometimes we need to use values from the types world into actual computation. The function `valueOf` allows us to do that
  - Thus

    `i<w` is not type correct

    `i<valueOf(w)` is type correct

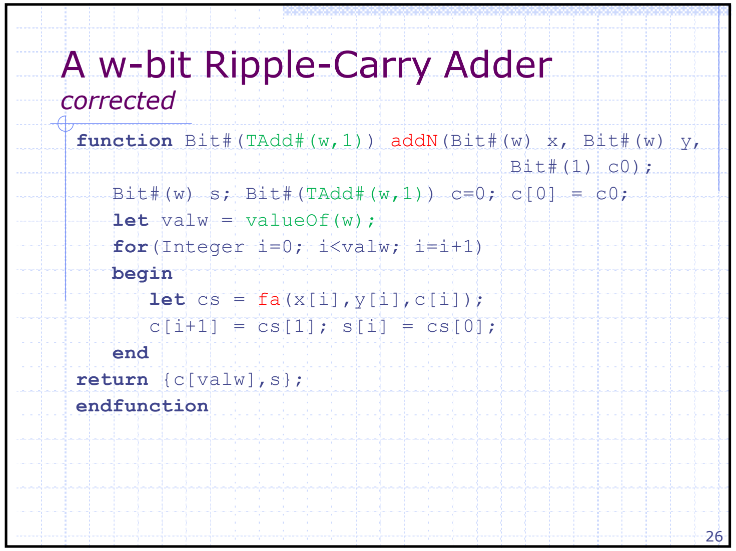
24

24



TAdd#(w,1) versus w+1

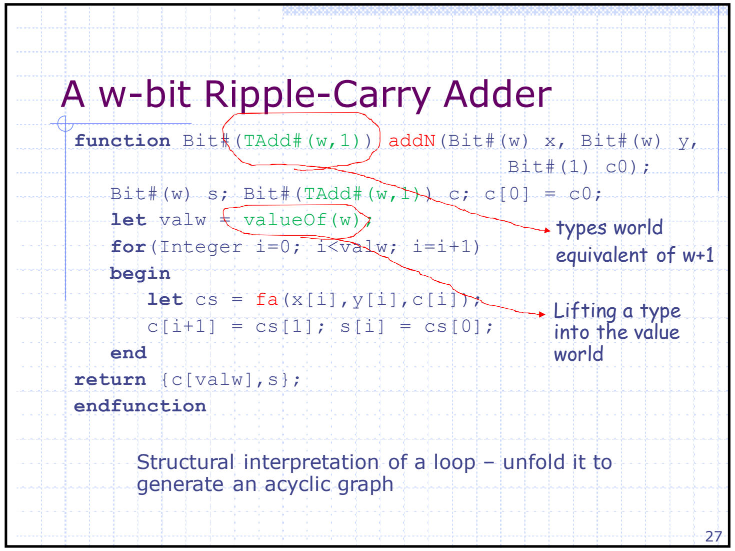
- Sometimes we need to perform operations in the types world that are very similar to the operations in the value world
  - Examples: `Add, Mul, Log`
- We define a few special operators in the types world for such operations
  - Examples: `TAdd#(m,n), TMul#(m,n), …`



A w-bit Ripple-Carry Adder

corrected

```
function Bit#(TAdd#(w,1)) addN(Bit#(w) x, Bit#(w) y,
                                          Bit#(1) c0);
   Bit#(w) s; Bit#(TAdd#(w,1)) c=0; c[0] = c0;
   let valw = valueOf(w);
   for(Integer i=0; i<valw; i=i+1)
   begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
   end
return {c[valw],s};
endfunction
```

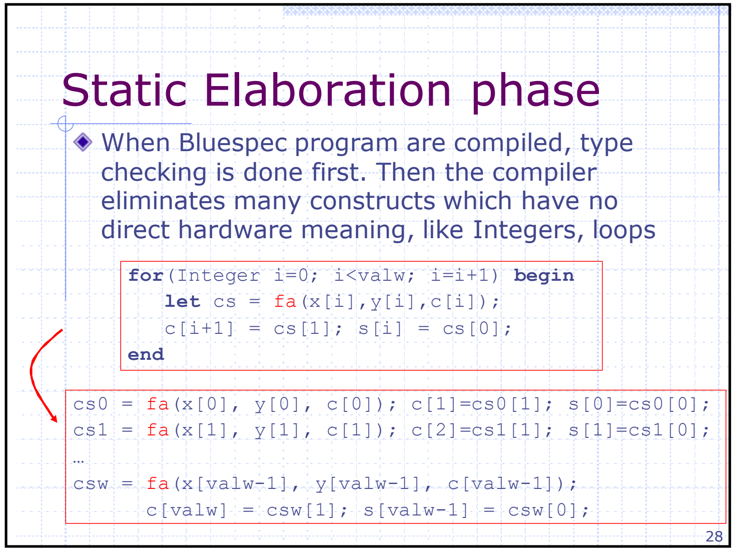


A w-bit Ripple-Carry Adder

```
function Bit#(TAdd#(w,1)) addN(Bit#(w) x, Bit#(w) y,
                                          Bit#(1) c0);
   Bit#(w) s; Bit#(TAdd#(w,1)) c; c[0] = c0;
   let valw = valueOf(w);
   for(Integer i=0; i<valw; i=i+1)
   begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
   end
return {c[valw],s};
endfunction
```

types world
equivalent of w+1
Lifting a type
into the value
world

Structural interpretation of a loop – unfold it to generate an acyclic graph

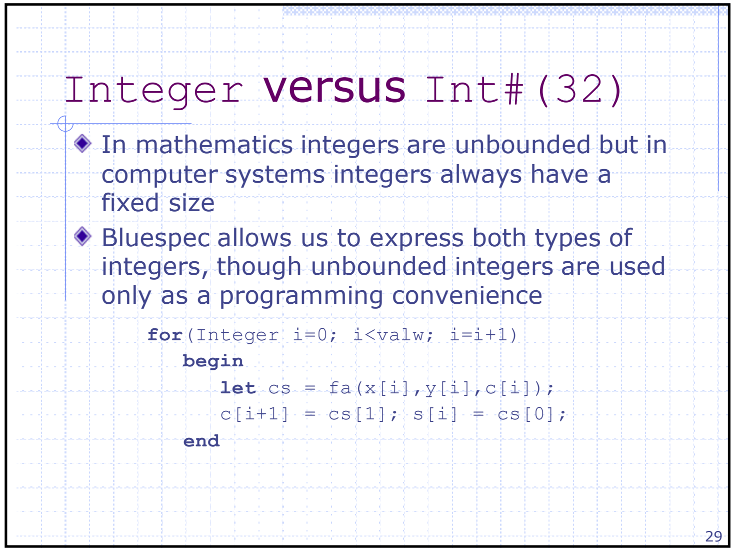


Static Elaboration phase

- When Bluespec program are compiled, type checking is done first. Then the compiler eliminates many constructs which have no direct hardware meaning, like Integers, loops

```
for(Integer i=0; i<valw; i=i+1) begin
   let cs = fa(x[i],y[i],c[i]);
   c[i+1] = cs[1]; s[i] = cs[0];
end
```

```
cs0 = fa(x[0], y[0], c[0]); c[1]=cs0[1]; s[0]=cs0[0];
cs1 = fa(x[1], y[1], c[1]); c[2]=cs1[1]; s[1]=cs1[0];
…
csw = fa(x[valw-1], y[valw-1], c[valw-1]);
          c[valw] = csw[1]; s[valw-1] = csw[0];
```



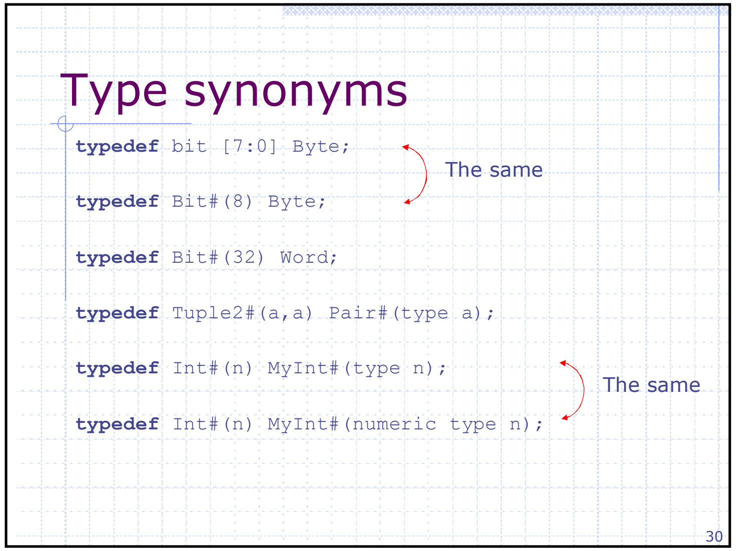
Integer versus Int#(32)

- In mathematics integers are unbounded but in computer systems integers always have a fixed size
- Bluespec allows us to express both types of integers, though unbounded integers are used only as a programming convenience

```
for(Integer i=0; i<valw; i=i+1)
   begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
   end
```

29

29



Type synonyms

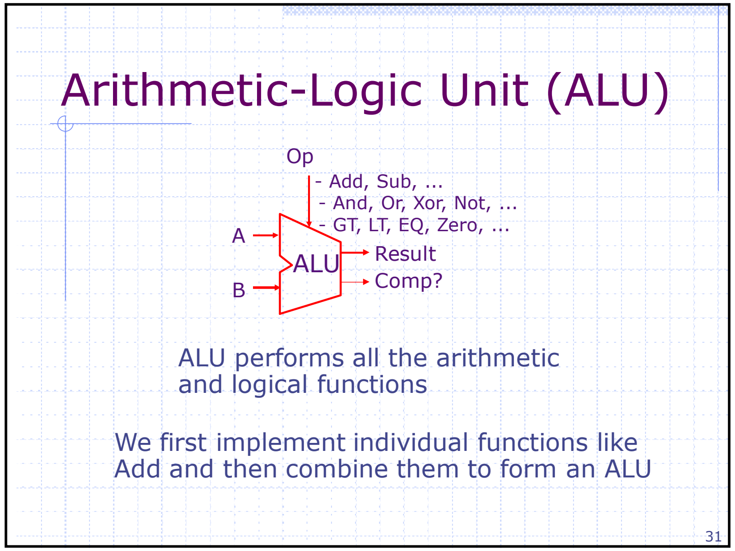
```
typedef bit [7:0] Byte;
typedef Bit#(8) Byte;
typedef Bit#(32) Word;
typedef Tuple2#(a,a) Pair#(type a);
typedef Int#(n) MyInt#(type n);
typedef Int#(n) MyInt#(numeric type n);
```

The same

The same

30

30



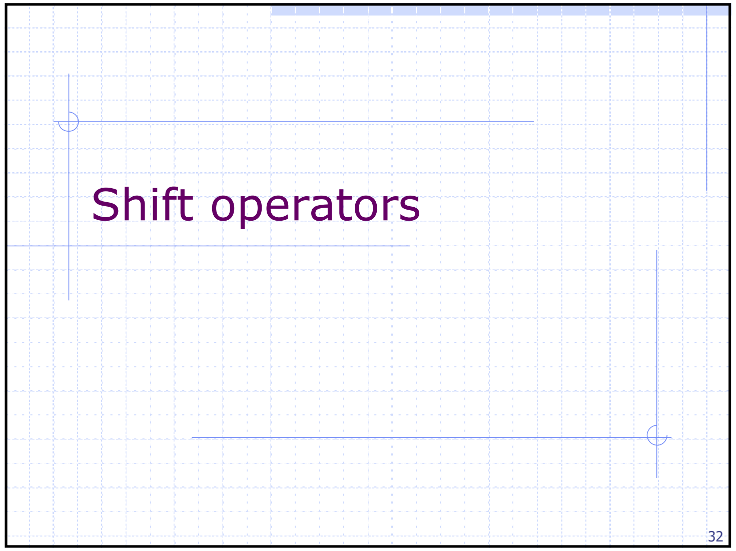
Arithmetic-Logic Unit (ALU)

ALU performs all the arithmetic and logical functions

We first implement individual functions like Add and then combine them to form an ALU

31

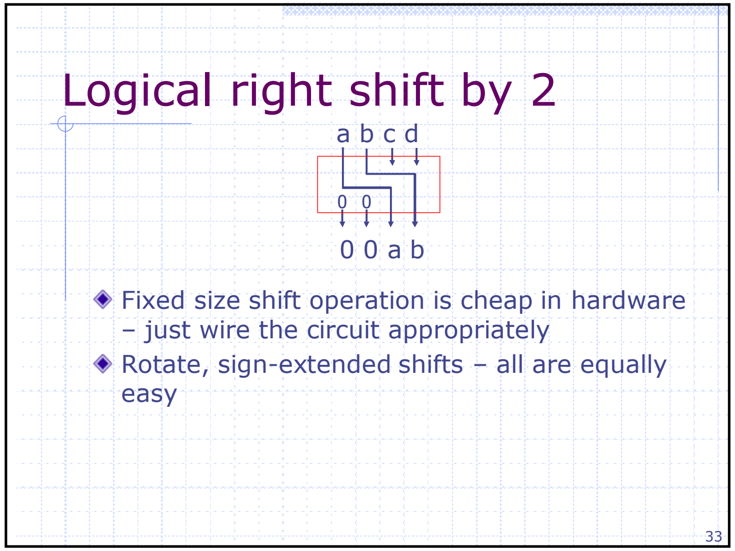
31



Shift operators

32

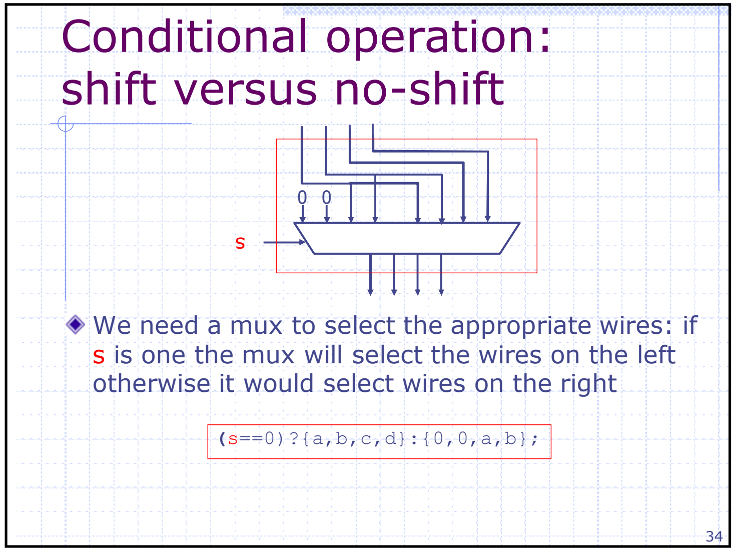
32



Logical right shift by 2

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately
- Rotate, sign-extended shifts – all are equally easy

33

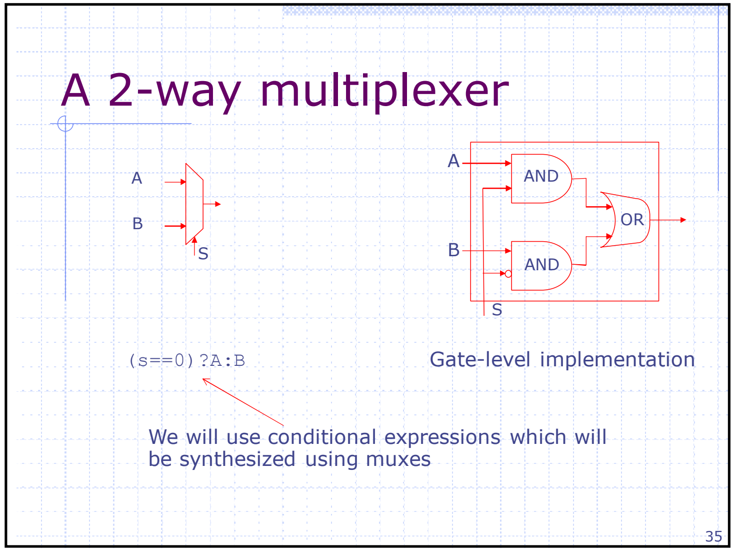


Conditional operation: shift versus no-shift

- We need a mux to select the appropriate wires: if s is one the mux will select the wires on the left otherwise it would select wires on the right

```
(s==0)?{a,b,c,d}:{0,0,a,b};
```

34



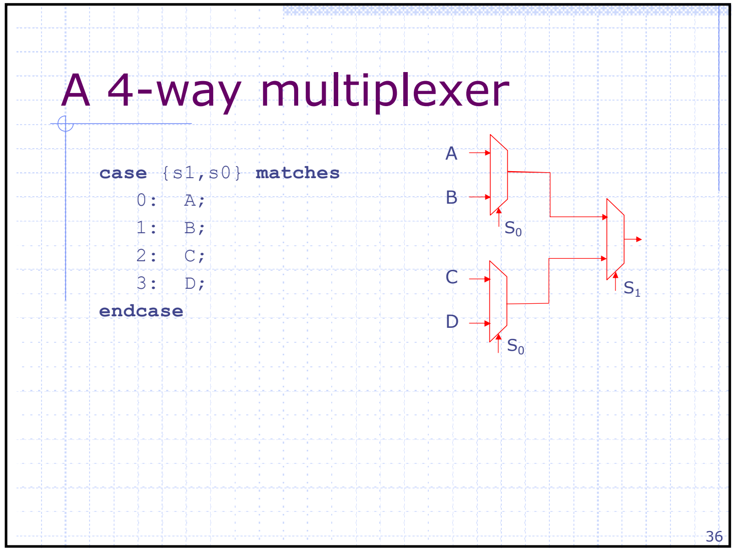
A 2-way multiplexer

```
(s==0)?A:B
```

Gate-level implementation

We will use conditional expressions which will be synthesized using muxes

35



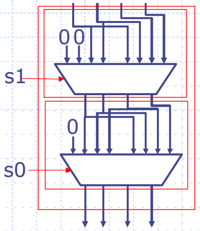
A 4-way multiplexer

```
case {s1,s0} matches
    0:  A;
    1:  B;
    2:  C;
    3:  D;
endcase
```

36

Logical right shift by *n*

- Shift *n* can be broken down in log *n* steps of fixed-length shifts of size 1, 2, 4, …
  - Shift 3 can be performed by doing a shift 2 and shift 1
- We need a mux to omit a particular size shift
- Shift circuit can be expressed as log *n* nested conditional expressions



You will write a Blusepec program to produce a variable size shifter in Lab 1

A Degression on Types

- Suppose we have a variable c whose values can represent three different colors
  - We can declare the type of c to be Bit#(2) and say that 00 represents Red, 01 Blue and 10 Green
- A better way is to create a new type called `Color` as follows:

```
typedef enum {Red, Blue, Green}
Color deriving(Bits, Eq);
```

Types prevent us from mixing raw bits and bits that represent color

The compiler will automatically assign some bit representation to the three colors and also provide a function to test if two colors are equal. If you do not use "deriving" then you will have to specify the representation and equality

Enumerated types

```
typedef enum {Red, Blue, Green}
Color deriving(Bits, Eq);

typedef enum {Eq, Neq, Le, Lt, Ge, Gt, AT, NT}
BrFunc deriving(Bits, Eq);

typedef enum {Add, Sub, And, Or, Xor, Nor, Slt, Sltu,
LShift, RShift, Sra}
AluFunc deriving(Bits, Eq);
```

Each enumerated type defines a new type

Combinational ALU

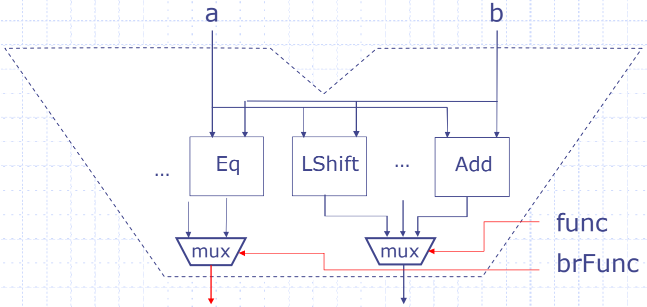
```
function Data alu(Data a, Data b, AluFunc func);
  Data res = case(func)
     Add   : (a + b);
     Sub   : (a - b);
     And   : (a & b);
     Or    : (a | b);
     Xor   : (a ^ b);
     Nor   : ~(a | b);
     Slt   : zeroExtend( pack( signedLT(a, b) ) );
     Sltu  : zeroExtend( pack( a < b ) );
     LShift: (a << b[4:0]);
     RShift: (a >> b[4:0]);
     Sra   : signedShiftRight(a, b[4:0]);
  endcase;
  return res;
endfunction
```

Given an implementation of the primitive operations like `addN, Shift, etc.` the ALU can be implemented simply by introducing a mux controlled by `op` to select the appropriate circuit

Comparison operators

```
function Bool aluBr(Data a, Data b, BrFunc brFunc);
    Bool brTaken = case(brFunc)
      Eq  : (a == b);
      Neq : (a != b);
      Le  : signedLE(a, 0);
      Lt  : signedLT(a, 0);
      Ge  : signedGE(a, 0);
      Gt  : signedGT(a, 0);
      AT  : True;
      NT  : False;
    endcase;
    return brTaken;
  endfunction
```

ALU including Comparison operators

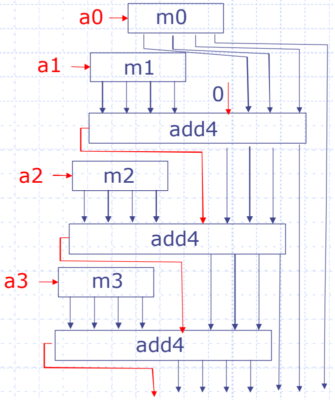


Complex combinational circuits

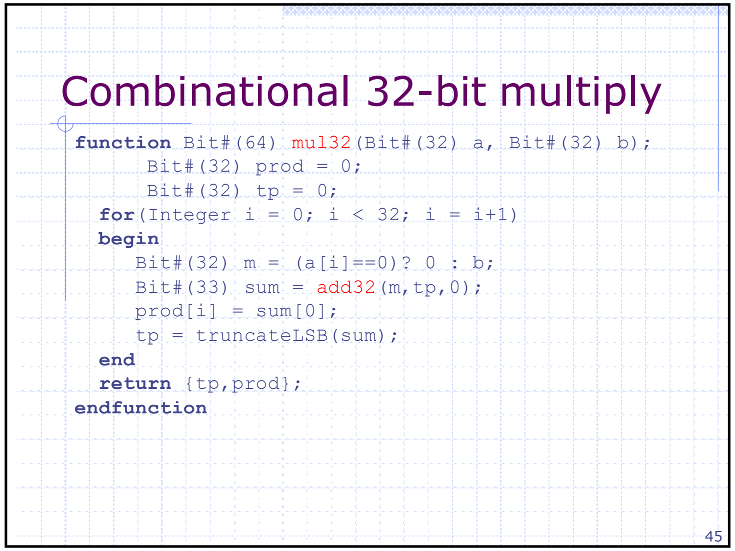
Multiplication

Multiplication by repeated addition

```
b Multiplicand   1101  (13)
a Muliplier  *   1011  (11)
                 1101
             +  1101
             + 0000
             + 1101
           10001111  (143)
```



```
mi = (a[i]==0)? 0 : b;
```

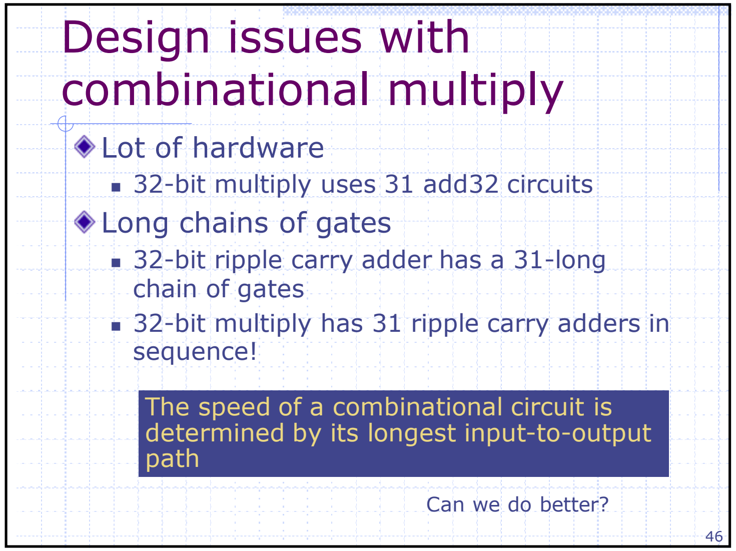


Combinational 32-bit multiply

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
      Bit#(32) prod = 0;
      Bit#(32) tp = 0;
  for(Integer i = 0; i < 32; i = i+1)
  begin
     Bit#(32) m = (a[i]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod[i] = sum[0];
     tp = truncateLSB(sum);
  end
  return {tp,prod};
endfunction
```

45

45



Design issues with combinational multiply

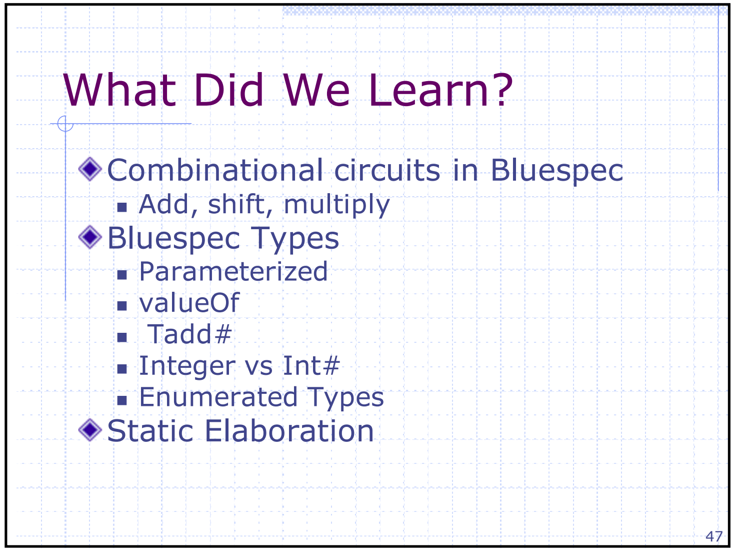
- Lot of hardware
  - 32-bit multiply uses 31 add32 circuits
- Long chains of gates
  - 32-bit ripple carry adder has a 31-long chain of gates
  - 32-bit multiply has 31 ripple carry adders in sequence!

The speed of a combinational circuit is determined by its longest input-to-output path

Can we do better?

46

46

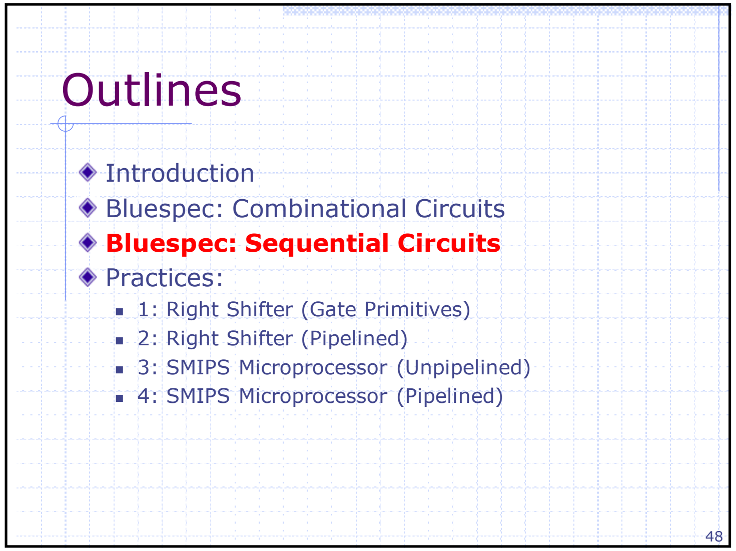


What Did We Learn?

- Combinational circuits in Bluespec
  - Add, shift, multiply
- Bluespec Types
  - Parameterized
  - valueOf
  - Tadd#
  - Integer vs Int#
  - Enumerated Types
- Static Elaboration

47

47

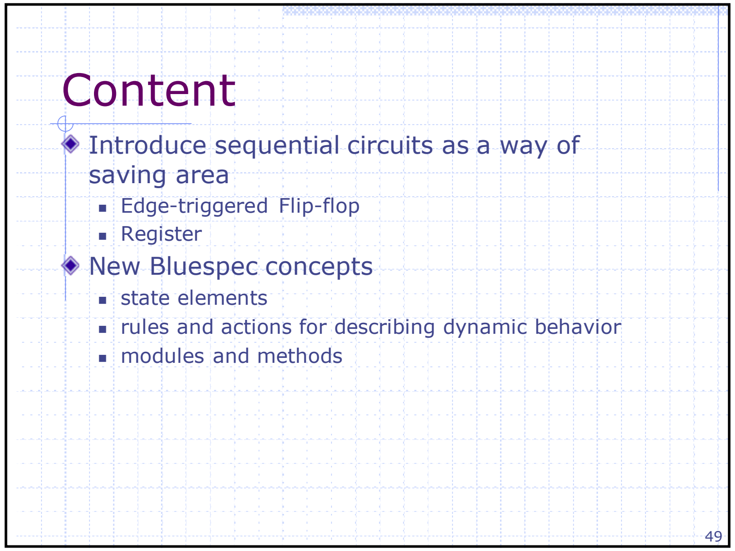


Outlines

- Introduction
- Bluespec: Combinational Circuits
- **Bluespec: Sequential Circuits**
- Practices:
  - 1: Right Shifter (Gate Primitives)
  - 2: Right Shifter (Pipelined)
  - 3: SMIPS Microprocessor (Unpipelined)
  - 4: SMIPS Microprocessor (Pipelined)

48

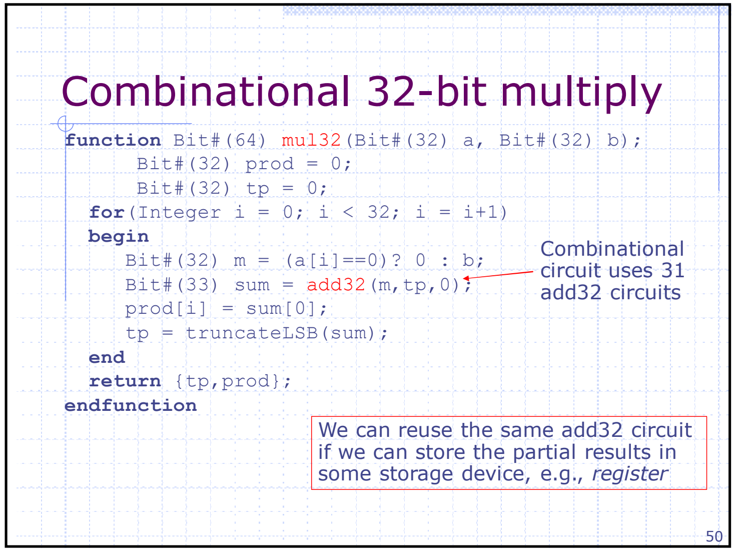
48



Content

- Introduce sequential circuits as a way of saving area
  - Edge-triggered Flip-flop
  - Register
- New Bluespec concepts
  - state elements
  - rules and actions for describing dynamic behavior
  - modules and methods

49



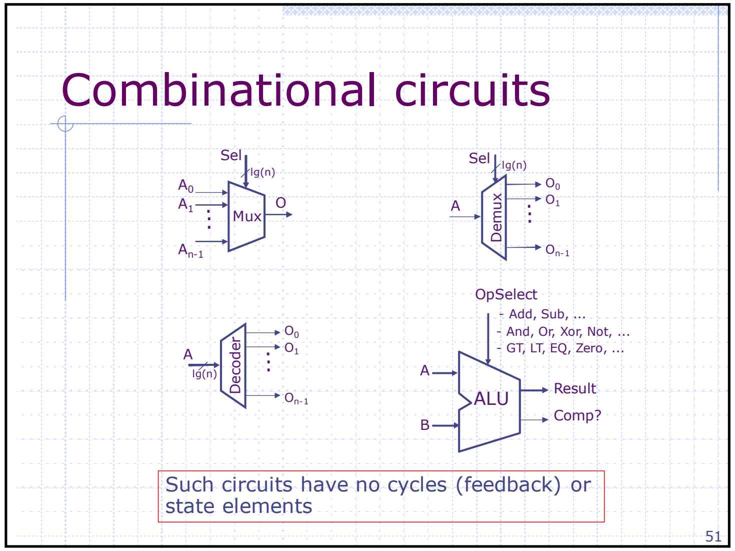
Combinational 32-bit multiply

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
      Bit#(32) prod = 0;
      Bit#(32) tp = 0;
  for(Integer i = 0; i < 32; i = i+1)
  begin
     Bit#(32) m = (a[i]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod[i] = sum[0];
     tp = truncateLSB(sum);
  end
  return {tp,prod};
endfunction
```

Combinational circuit uses 31 add32 circuits

We can reuse the same add32 circuit if we can store the partial results in some storage device, e.g., *register*

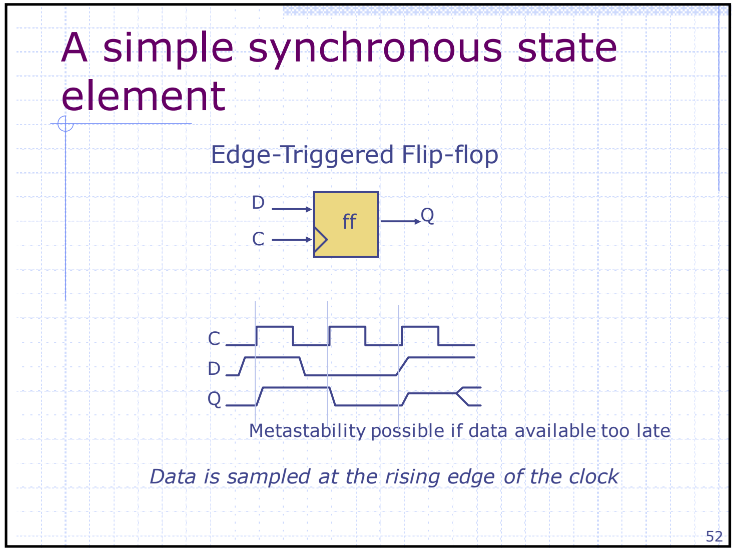
50



Combinational circuits

Such circuits have no cycles (feedback) or state elements

51



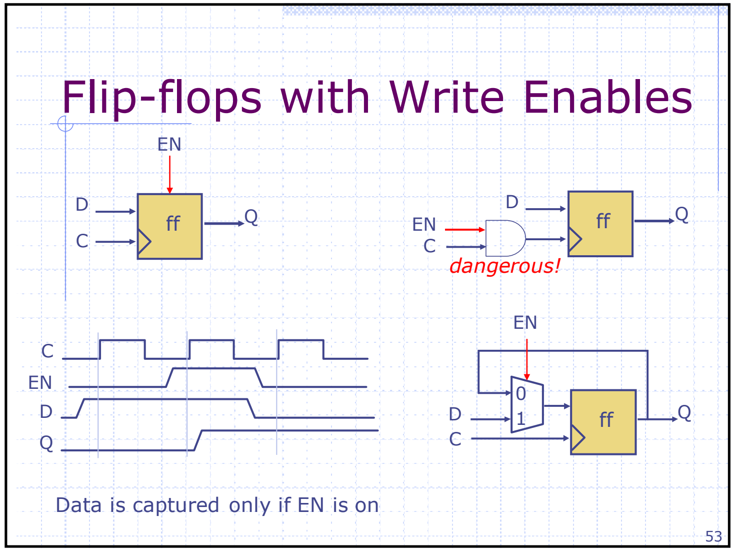
A simple synchronous state element

Edge-Triggered Flip-flop

Metastability possible if data available too late

Data is sampled at the rising edge of the clock

52

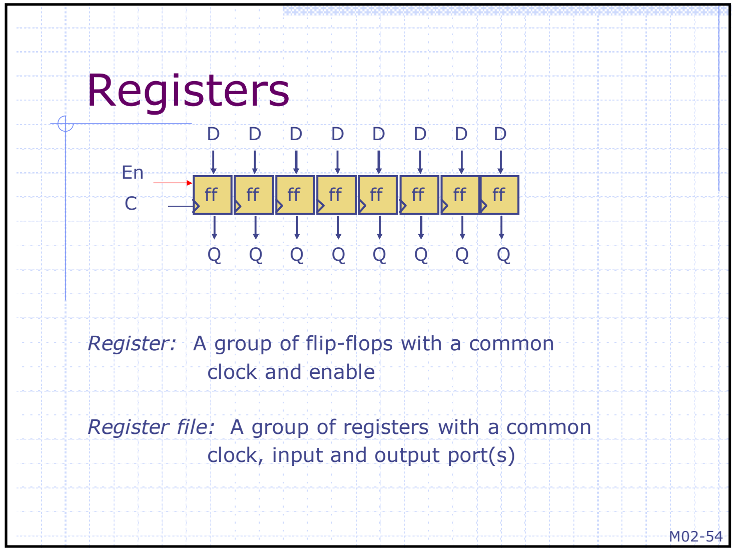


Flip-flops with Write Enables

Data is captured only if EN is on

53

53



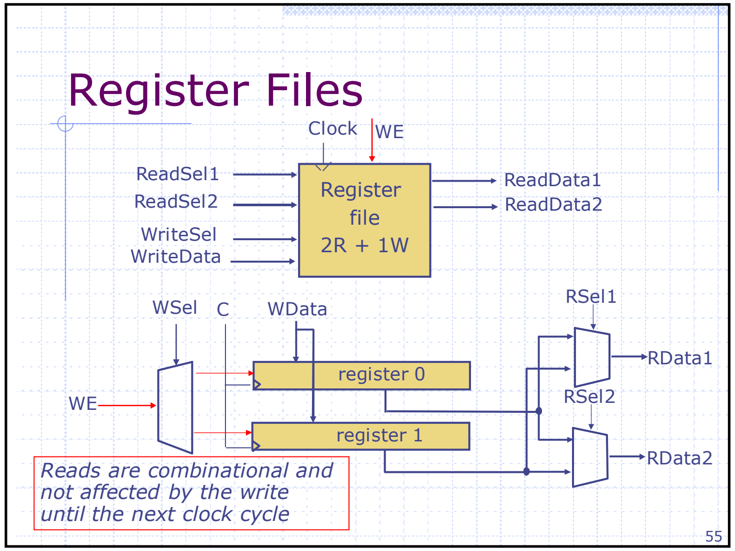
Registers

Register: A group of flip-flops with a common clock and enable

Register file: A group of registers with a common clock, input and output port(s)

M02-54

54

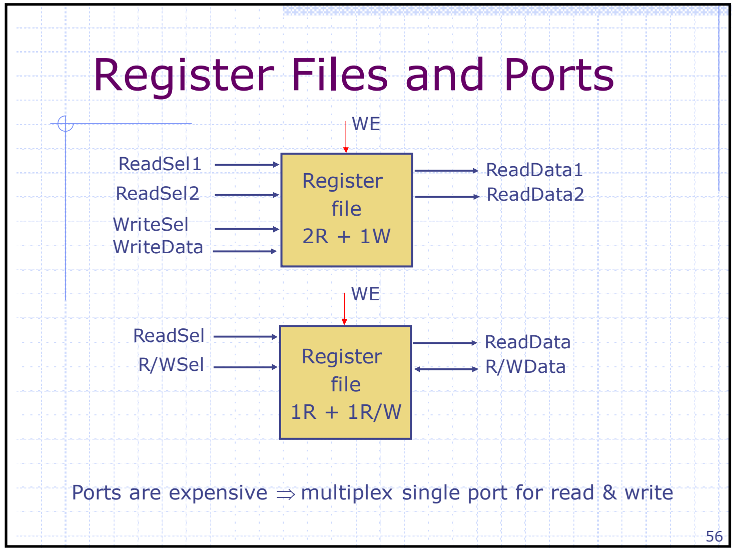


Register Files

Reads are combinational and not affected by the write until the next clock cycle

55

55

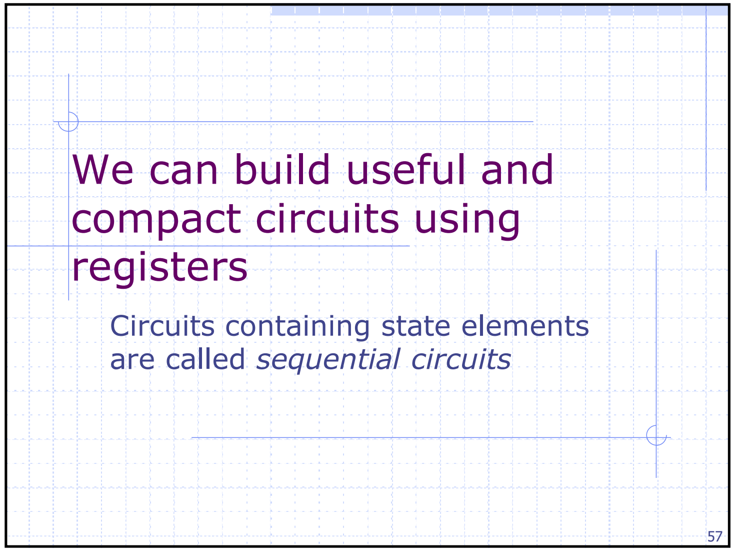


Register Files and Ports

Ports are expensive ⇒ multiplex single port for read & write

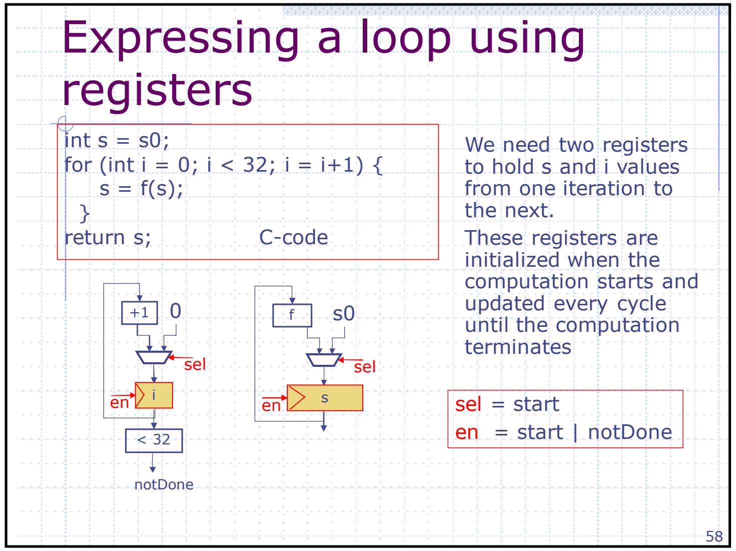
56

56



We can build useful and compact circuits using registers

Circuits containing state elements are called *sequential circuits*



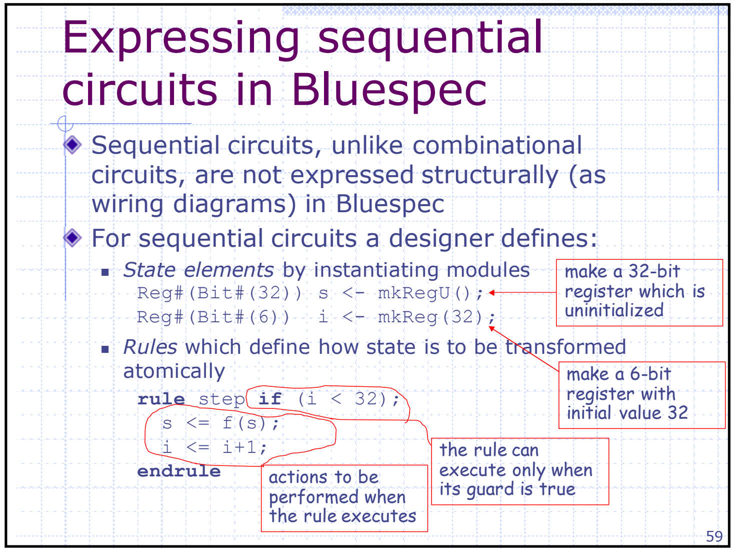
Expressing a loop using registers

```
int s = s0;
for (int i = 0; i < 32; i = i+1) {
    s = f(s);
 }
return s;                C-code
```

We need two registers to hold s and i values from one iteration to the next.

These registers are initialized when the computation starts and updated every cycle until the computation terminates

sel = start
en = start | notDone



Expressing sequential circuits in Bluespec

- Sequential circuits, unlike combinational circuits, are not expressed structurally (as wiring diagrams) in Bluespec
- For sequential circuits a designer defines:
  - *State elements* by instantiating modules

```
Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);
```

make a 32-bit register which is uninitialized

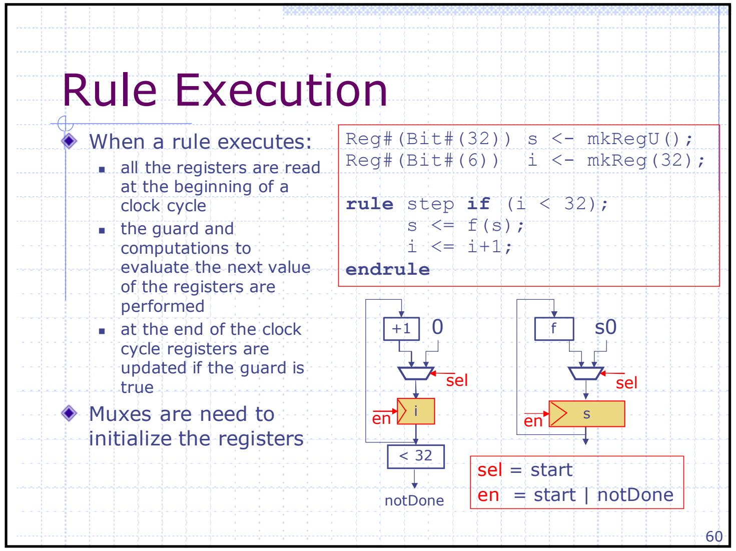
make a 6-bit register with initial value 32

  - *Rules* which define how state is to be transformed atomically

```
rule step if (i < 32);
  s <= f(s);
  i <= i+1;
endrule
```

actions to be performed when the rule executes

the rule can execute only when its guard is true



Rule Execution

- When a rule executes:
  - all the registers are read at the beginning of a clock cycle
  - the guard and computations to evaluate the next value of the registers are performed
  - at the end of the clock cycle registers are updated if the guard is true
- Muxes are need to initialize the registers

```
Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);

rule step if (i < 32);
    s <= f(s);
    i <= i+1;
endrule
```

sel = start
en = start | notDone

Multiply using registers

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
      Bit#(32) prod = 0;
      Bit#(32) tp = 0;
  for(Integer i = 0; i < 32; i = i+1)
  begin
     Bit#(32) m = (a[i]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod[i] = sum[0];
     tp = truncateLSB(sum);
  end
  return {tp,prod};
endfunction
```

Combinational version

Need registers to hold a, b, tp, prod and i

Update the registers every cycle until we are done

Sequential multiply

```
      Reg#(Bit#(32)) a <- mkRegU();
      Reg#(Bit#(32)) b <- mkRegU();
      Reg#(Bit#(32)) prod <-mkRegU();
      Reg#(Bit#(32)) tp <- mkRegU();
      Reg#(Bit#(6))  i <- mkReg(32);

rule mulStep if (i < 32);
   Bit#(32) m = (a[i]==0)? 0 : b;
   Bit#(33) sum = add32(m,tp,0);
   prod[i] <= sum[0];
   tp <= sum[32:1];
   i <= i+1;
endrule
```

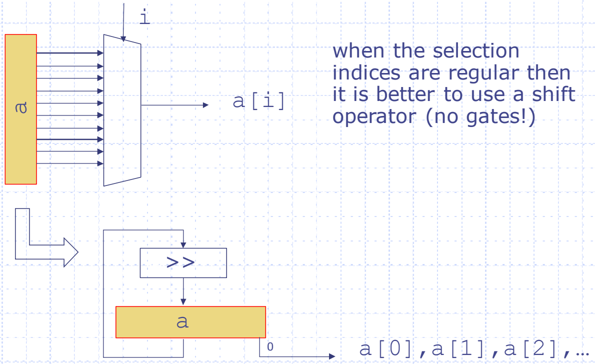
state elements

a rule to describe the dynamic behavior

similar to the loop body in the combinational version

The rule won't fire until i is set to value smaller than 32

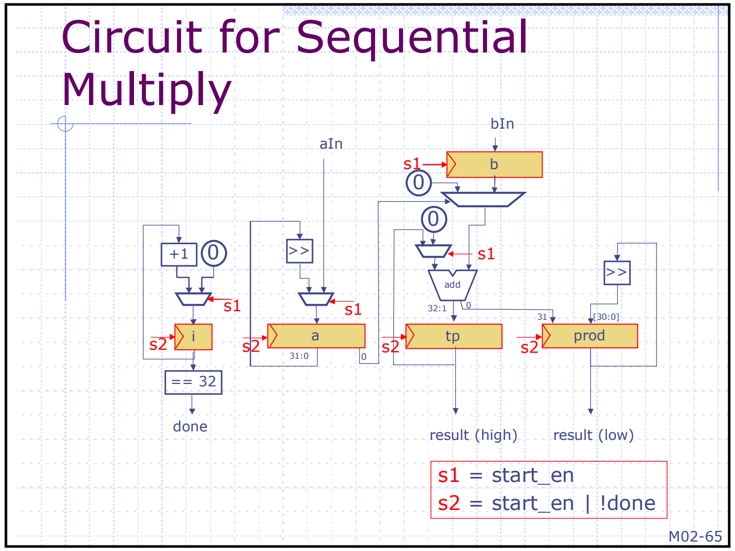
Dynamic selection requires a mux



when the selection indices are regular then it is better to use a shift operator (no gates!)

Replacing repeated selections by shifts

```
      Reg#(Bit#(32)) a <- mkRegU();
      Reg#(Bit#(32)) b <- mkRegU();
      Reg#(Bit#(32)) prod <-mkRegU();
      Reg#(Bit#(32)) tp <- mkRegU();
      Reg#(Bit#(6))  i <- mkReg(32);

rule mulStep if (i < 32);
   Bit#(32) m = (a[0]==0)? 0 : b;
   a <= (a >> 1);
   Bit#(33) sum = add32(m,tp,0);
   prod <= {sum[0], (prod >> 1)[30:0]};
   tp <= sum[32:1];
   i <= i+1;
endrule
```



Circuit for Sequential Multiply

s1 = start_en
s2 = start_en | !done

M02-65

65

Circuit analysis

- Number of add32 circuits has been reduced from 31 to one, though some registers and muxes have been added
- The longest combinational path has been reduced from 31 serial add32's to one add32 plus a few muxes
- The sequential circuit will take 31 clock cycles to compute an answer

66

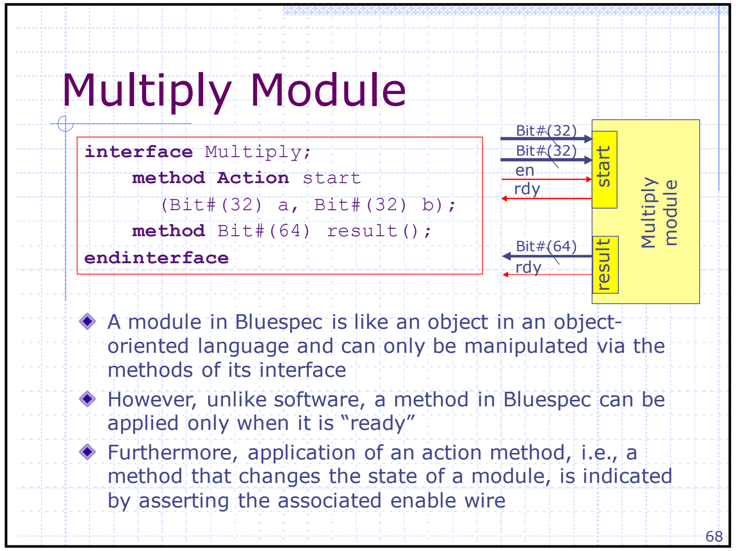
66

Modules

We often package sequential circuits into modules to hide the details

67

67



Multiply Module

```
interface Multiply;
    method Action start
      (Bit#(32) a, Bit#(32) b);
    method Bit#(64) result();
endinterface
```

- A module in Bluespec is like an object in an object-oriented language and can only be manipulated via the methods of its interface
- However, unlike software, a method in Bluespec can be applied only when it is "ready"
- Furthermore, application of an action method, i.e., a method that changes the state of a module, is indicated by asserting the associated enable wire

68

68

Multiply Module

```
module mkMultiply32 (Multiply);
      Reg#(Bit#(32)) a <- mkRegU();
      Reg#(Bit#(32)) b <- mkRegU();
      Reg#(Bit#(32)) prod <-mkRegU();
      Reg#(Bit#(32)) tp <- mkRegU();
      Reg#(Bit#(6))  i <- mkReg(32);
  rule mulStep if (i != 32);
     Bit#(32) m = (a[0]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod <= {sum[0], (prod >> 1)[30:0]};
     tp <= truncateLSB(sum); a <= a >> 1; i <= i+1;
  endrule
  method Action start(Bit#(32) aIn, Bit#(32) bIn)
                                            if (i == 32);
     a <= aIn; b <= bIn; i <= 0; tp <= 0; prod <= 0;
  endmethod
  method Bit#(64) result() if (i == 32);
     return {tp,prod};
  endmethod  endmodule
```

State

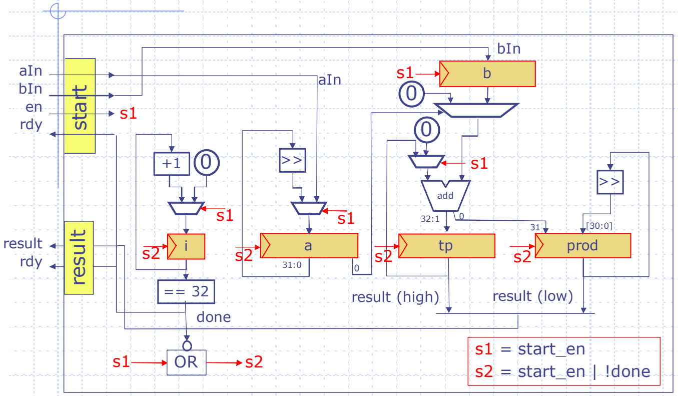
Internal behavior

External interface

method guards

69

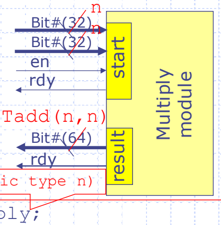
Module: Method Interface



s1 = start_en
s2 = start_en | !done

70

Polymorphic Multiply Module



n could be
32, 13, …

implicit conditions

# (Numeric type n)

```
interface Multiply;
    method Action start (Bit#(32) a, Bit#(32) b);
    method Bit#(64) result();
endinterface
```

n n TAdd#(n,n)

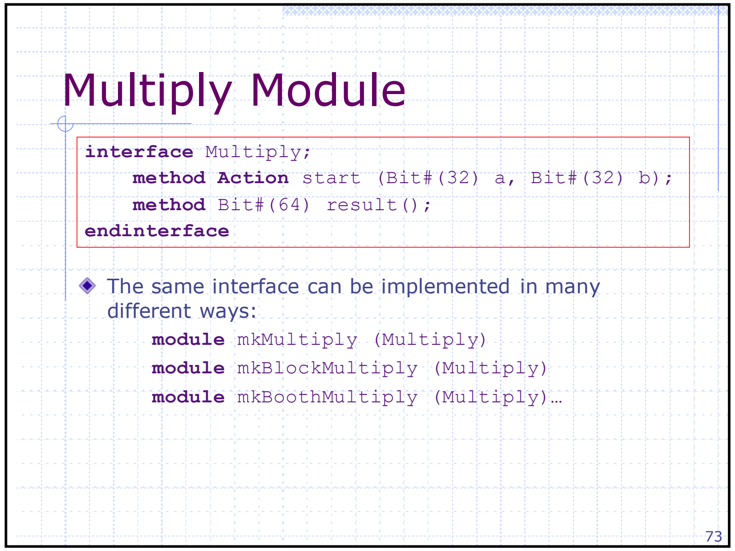
- The module can easily be made polymorphic

71

Sequential n-bit multiply

```
module mkMultiplyN (MultiplyN#(n));
      Reg#(Bit#(n)) a <- mkRegU();
      Reg#(Bit#(n)) b <- mkRegU();
      Reg#(Bit#(n)) prod <-mkRegU();
      Reg#(Bit#(n)) tp <- mkRegU();
     let nv = fromInteger(valueOf(n));
     Reg#(Bit#(TAdd#(TLog#(n),1)))  i <- mkReg(nv);
  rule mulStep if (i != nv);
     Bit#(n) m = (a[0]==0)? 0 : b;
     Bit#(Tadd#(n,1)) sum = addN(m,tp,0);
     prod <= {sum[0], (prod >> 1)[(nv-2):0]};
     tp <= truncateLSB(sum); a <= a >> 1; i <= i+1;
  endrule
  method Action start(Bit#(n) aIn, Bit#(n) bIn) if (i == nv);
     a <= aIn; b <= bIn; i <= 0; tp <= 0; prod <= 0;
  endmethod
  method Bit#(Tadd#(n,n)) result() if (i == nv);
     return {tp,prod};
  endmethod  endmodule
```

72

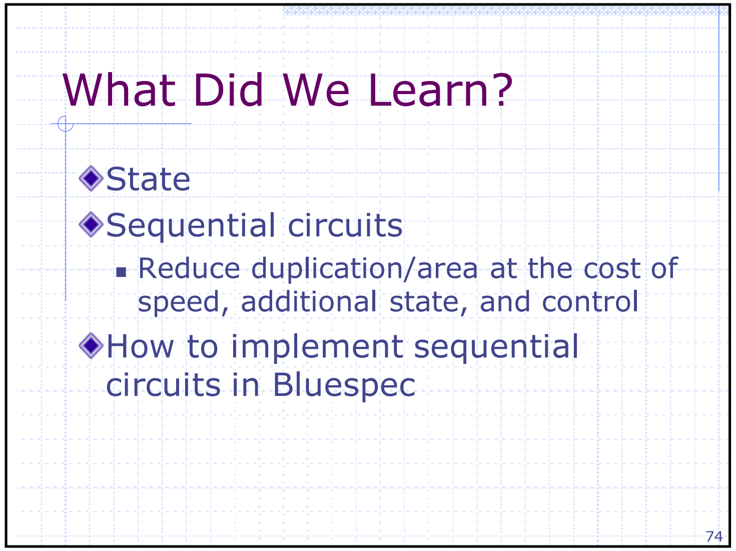


Multiply Module

```
interface Multiply;
    method Action start (Bit#(32) a, Bit#(32) b);
    method Bit#(64) result();
endinterface
```

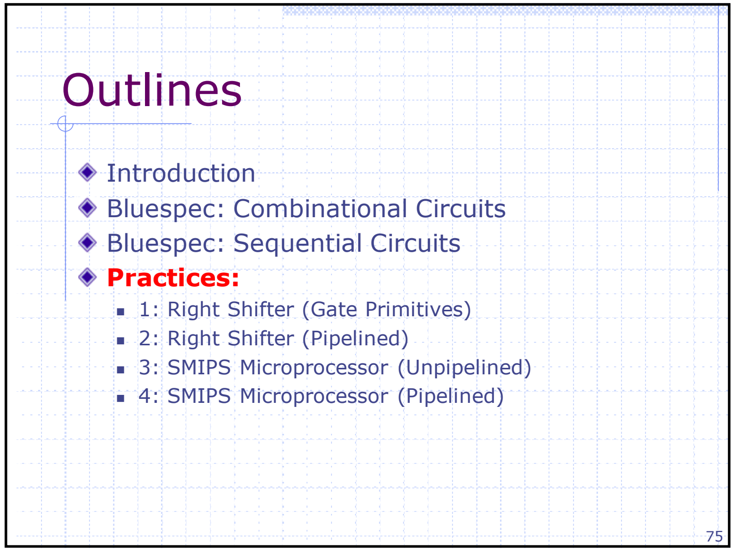
- The same interface can be implemented in many different ways:

```
    module mkMultiply (Multiply)
    module mkBlockMultiply (Multiply)
    module mkBoothMultiply (Multiply)…
```



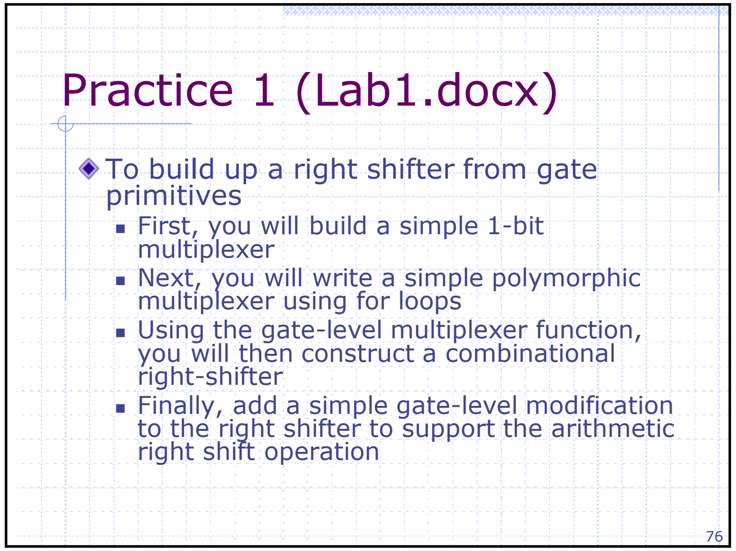
What Did We Learn?

- State
- Sequential circuits
  - Reduce duplication/area at the cost of speed, additional state, and control
- How to implement sequential circuits in Bluespec



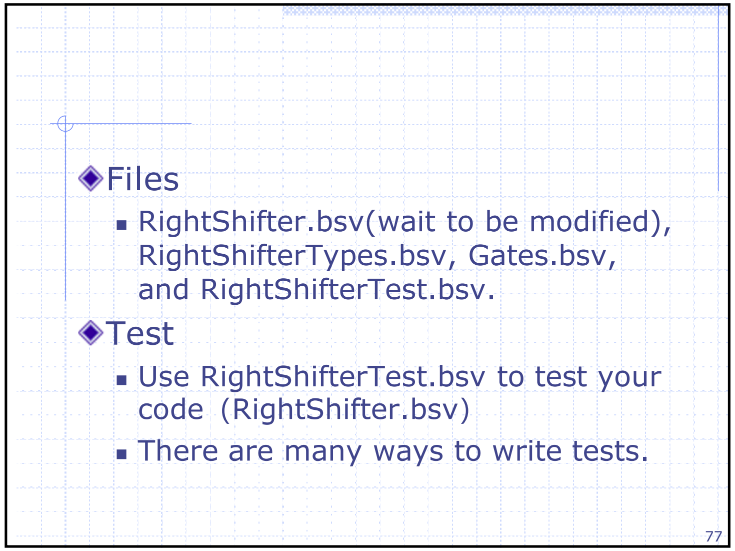
Outlines

- Introduction
- Bluespec: Combinational Circuits
- Bluespec: Sequential Circuits
- **Practices:**
  - 1: Right Shifter (Gate Primitives)
  - 2: Right Shifter (Pipelined)
  - 3: SMIPS Microprocessor (Unpipelined)
  - 4: SMIPS Microprocessor (Pipelined)



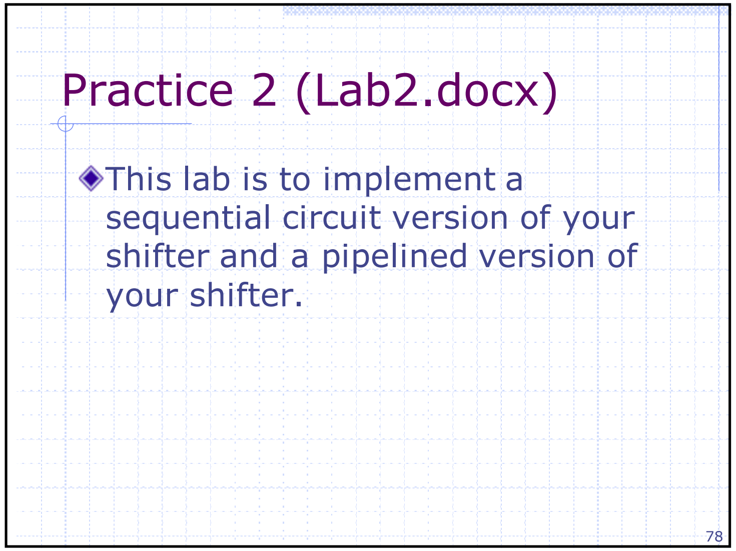
Practice 1 (Lab1.docx)

- To build up a right shifter from gate primitives
  - First, you will build a simple 1-bit multiplexer
  - Next, you will write a simple polymorphic multiplexer using for loops
  - Using the gate-level multiplexer function, you will then construct a combinational right-shifter
  - Finally, add a simple gate-level modification to the right shifter to support the arithmetic right shift operation



- Files
  - RightShifter.bsv(wait to be modified), RightShifterTypes.bsv, Gates.bsv, and RightShifterTest.bsv.
- Test
  - Use RightShifterTest.bsv to test your code (RightShifter.bsv)
  - There are many ways to write tests.

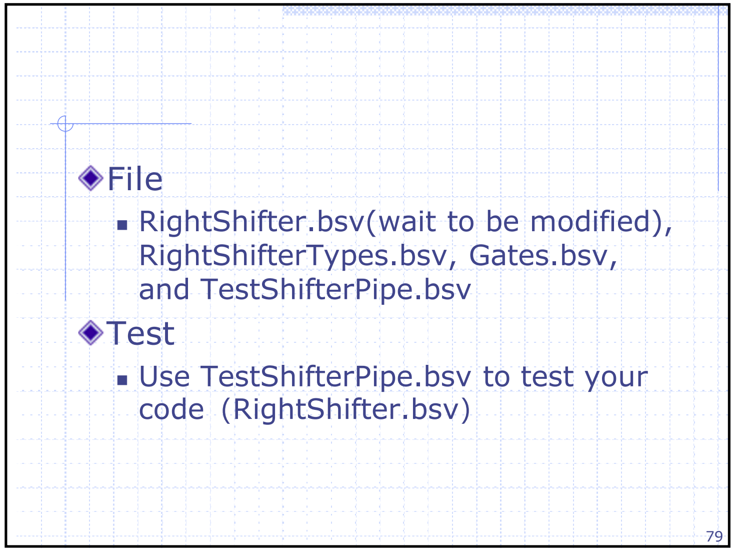
77



Practice 2 (Lab2.docx)

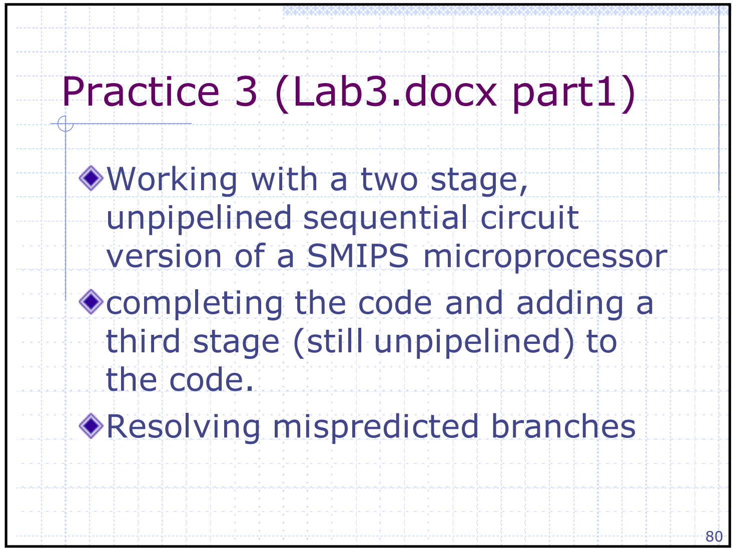
- This lab is to implement a sequential circuit version of your shifter and a pipelined version of your shifter.

78



- File
  - RightShifter.bsv(wait to be modified), RightShifterTypes.bsv, Gates.bsv, and TestShifterPipe.bsv
- Test
  - Use TestShifterPipe.bsv to test your code (RightShifter.bsv)

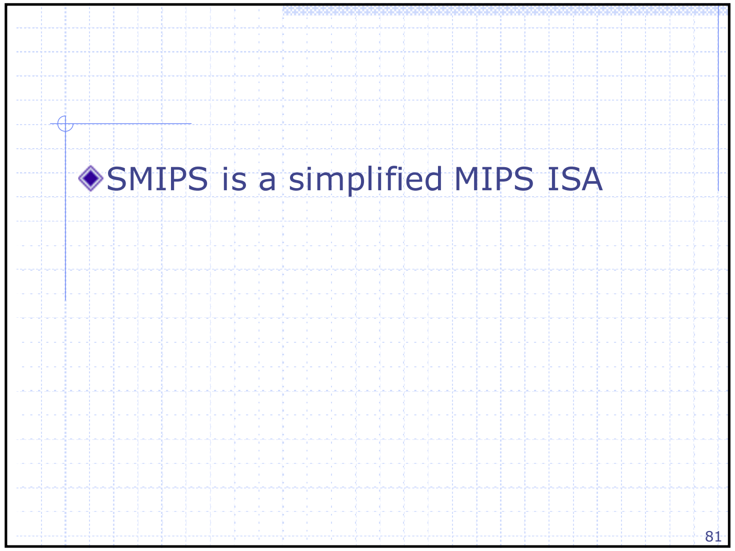
79



Practice 3 (Lab3.docx part1)

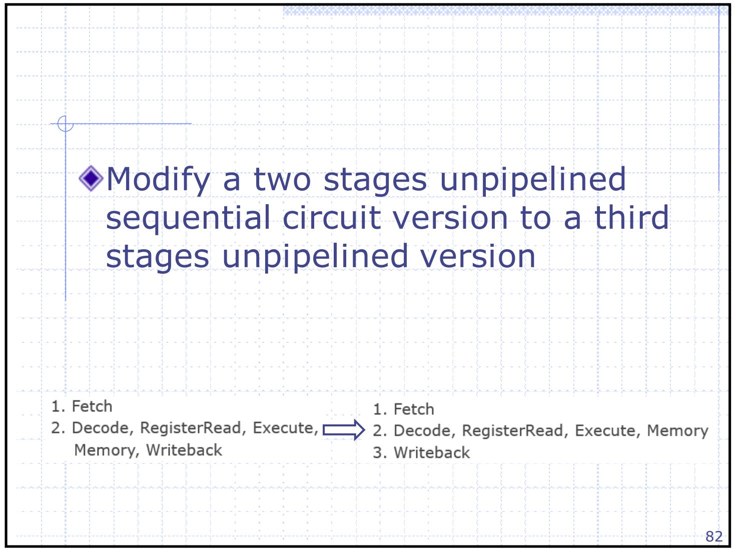
- Working with a two stage, unpipelined sequential circuit version of a SMIPS microprocessor
- completing the code and adding a third stage (still unpipelined) to the code.
- Resolving mispredicted branches

80



- SMIPS is a simplified MIPS ISA

81



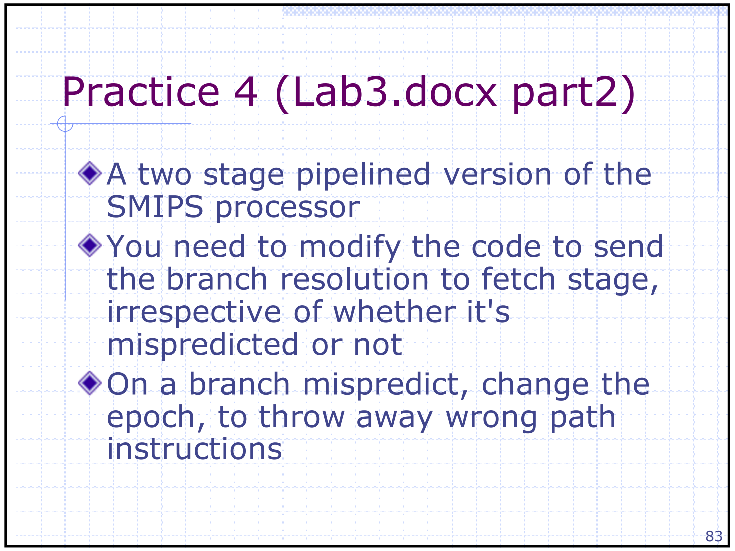
- Modify a two stages unpipelined sequential circuit version to a third stages unpipelined version

1. Fetch
2. Decode, RegisterRead, Execute, Memory, Writeback

⇒

1. Fetch
2. Decode, RegisterRead, Execute, Memory
3. Writeback

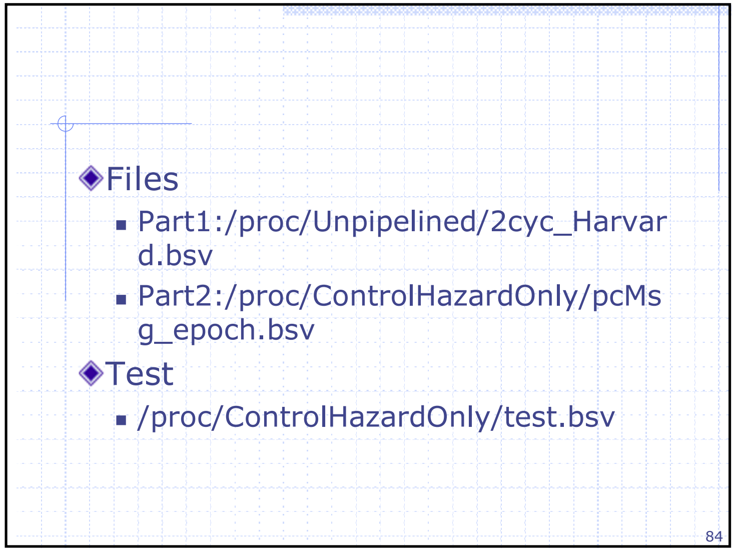
82



Practice 4 (Lab3.docx part2)

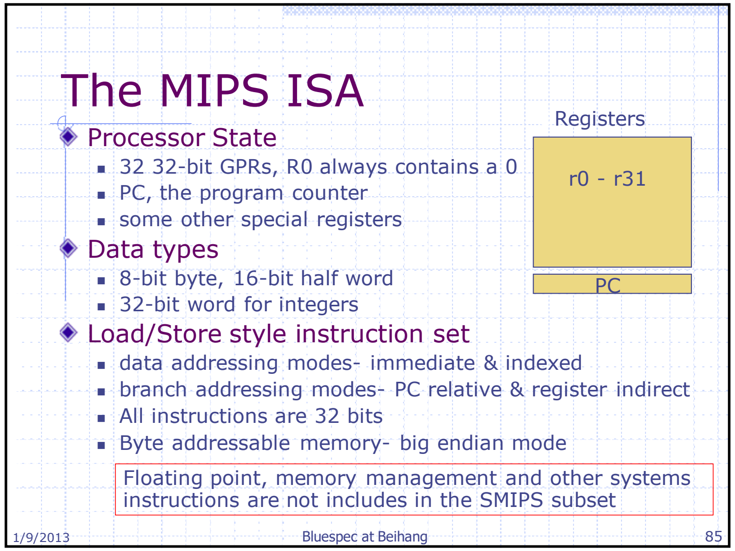
- A two stage pipelined version of the SMIPS processor
- You need to modify the code to send the branch resolution to fetch stage, irrespective of whether it's mispredicted or not
- On a branch mispredict, change the epoch, to throw away wrong path instructions

83



- Files
  - Part1:/proc/Unpipelined/2cyc_Harvard.bsv
  - Part2:/proc/ControlHazardOnly/pcMsg_epoch.bsv
- Test
  - /proc/ControlHazardOnly/test.bsv

84



The MIPS ISA

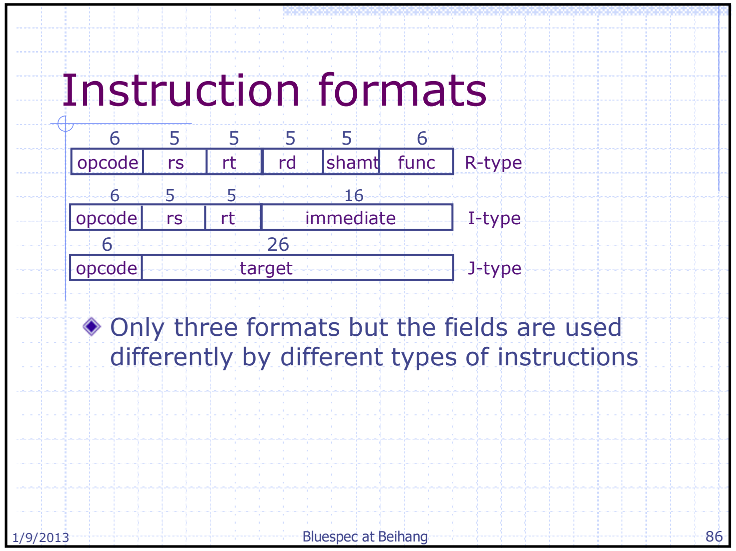
- Processor State
  - 32 32-bit GPRs, R0 always contains a 0
  - PC, the program counter
  - some other special registers
- Data types
  - 8-bit byte, 16-bit half word
  - 32-bit word for integers
- Load/Store style instruction set
  - data addressing modes- immediate & indexed
  - branch addressing modes- PC relative & register indirect
  - All instructions are 32 bits
  - Byte addressable memory- big endian mode

Registers

r0 - r31

PC

Floating point, memory management and other systems instructions are not includes in the SMIPS subset



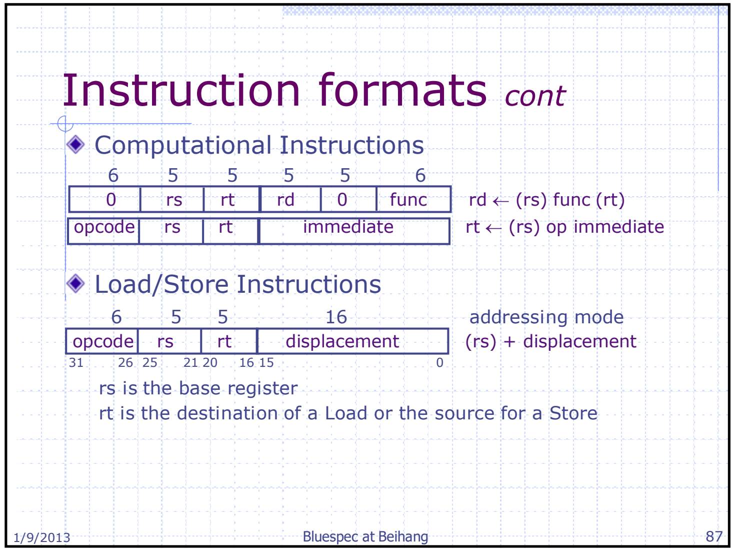
Instruction formats

6	5	5	5	5	6	
opcode	rs	rt	rd	shamt	func	R-type

6	5	5	16	
opcode	rs	rt	immediate	I-type

6	26	
opcode	target	J-type

- Only three formats but the fields are used differently by different types of instructions



Instruction formats *cont*

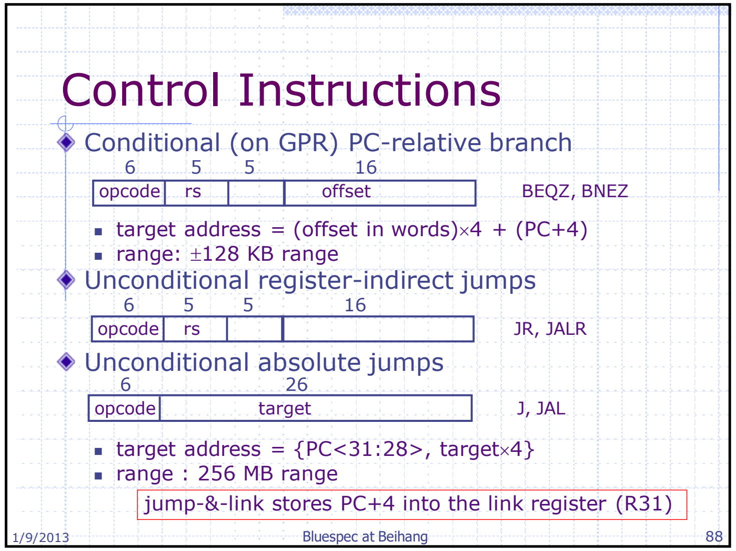
- Computational Instructions

6	5	5	5	5	6	
0	rs	rt	rd	0	func	rd ← (rs) func (rt)
opcode	rs	rt	immediate			rt ← (rs) op immediate

- Load/Store Instructions

6	5	5	16	addressing mode
opcode	rs	rt	displacement	(rs) + displacement
31 26	25 21	20 16	15 0	

rs is the base register
rt is the destination of a Load or the source for a Store



Control Instructions

- Conditional (on GPR) PC-relative branch

6	5	5	16	
opcode	rs		offset	BEQZ, BNEZ

  - target address = (offset in words)×4 + (PC+4)
  - range: ±128 KB range
- Unconditional register-indirect jumps

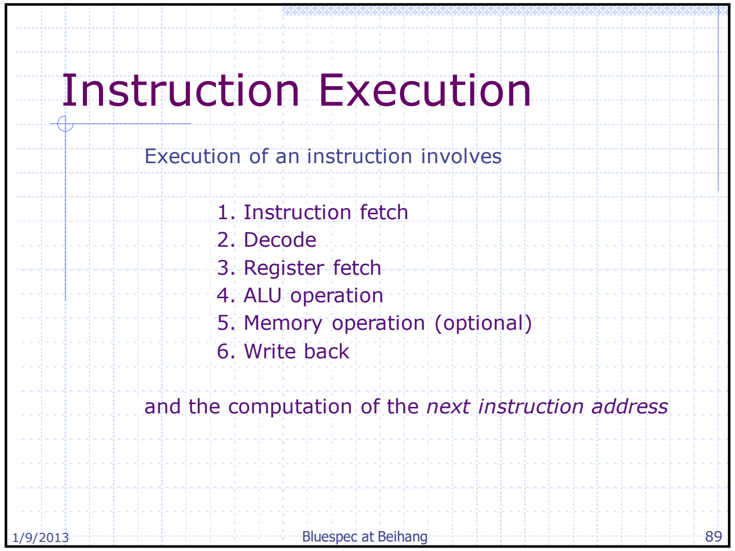
6	5	5	16	
opcode	rs			JR, JALR

- Unconditional absolute jumps

6	26	
opcode	target	J, JAL

  - target address = {PC<31:28>, target×4}
  - range : 256 MB range

jump-&-link stores PC+4 into the link register (R31)

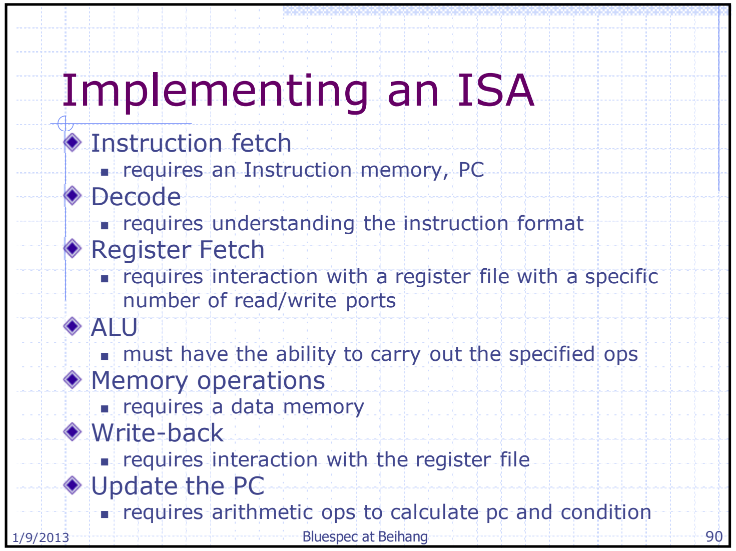


Instruction Execution

Execution of an instruction involves

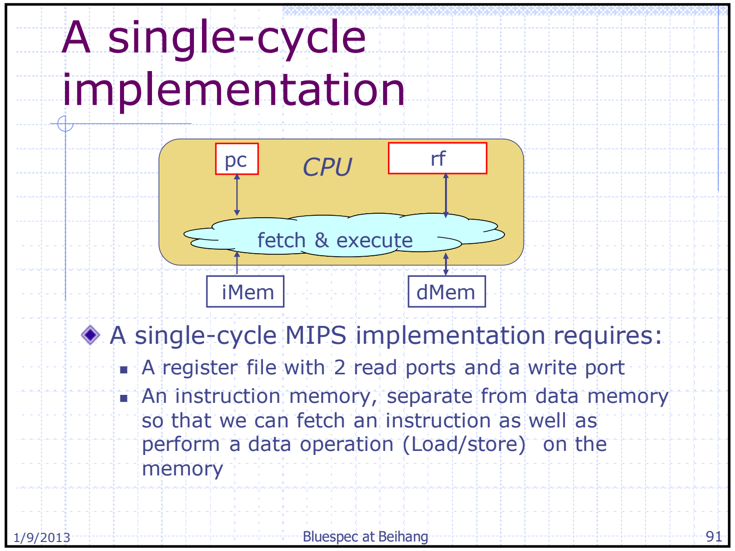
1. Instruction fetch
2. Decode
3. Register fetch
4. ALU operation
5. Memory operation (optional)
6. Write back

and the computation of the *next instruction address*



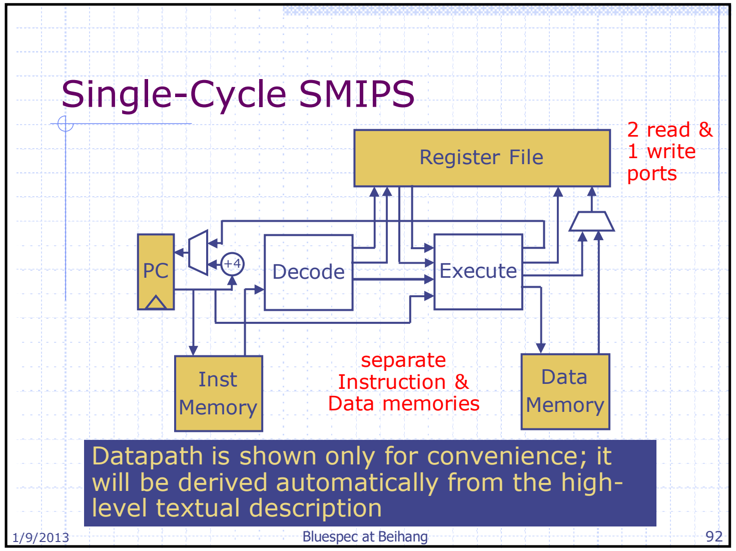
Implementing an ISA

- Instruction fetch
  - requires an Instruction memory, PC
- Decode
  - requires understanding the instruction format
- Register Fetch
  - requires interaction with a register file with a specific number of read/write ports
- ALU
  - must have the ability to carry out the specified ops
- Memory operations
  - requires a data memory
- Write-back
  - requires interaction with the register file
- Update the PC
  - requires arithmetic ops to calculate pc and condition



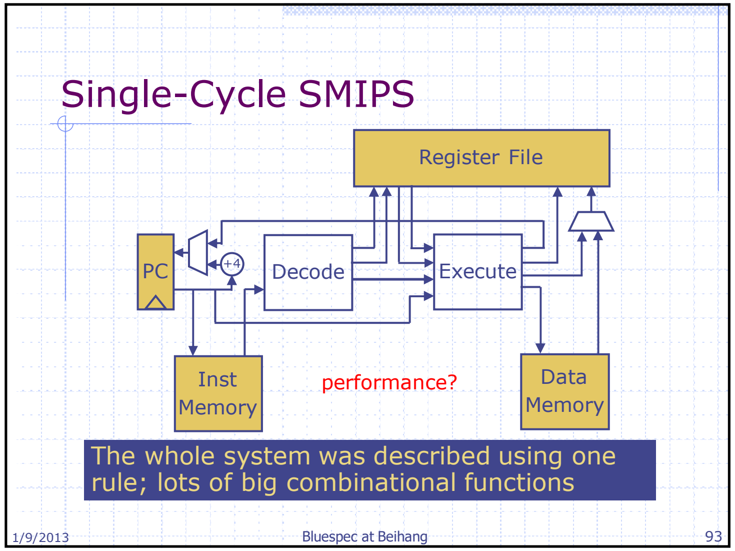
A single-cycle implementation

- A single-cycle MIPS implementation requires:
  - A register file with 2 read ports and a write port
  - An instruction memory, separate from data memory so that we can fetch an instruction as well as perform a data operation (Load/store) on the memory



Single-Cycle SMIPS

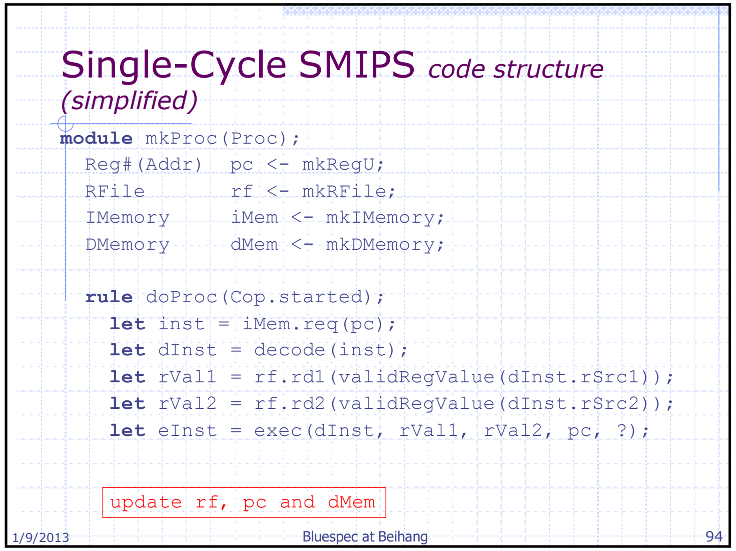
Datapath is shown only for convenience; it will be derived automatically from the high-level textual description



Single-Cycle SMIPS

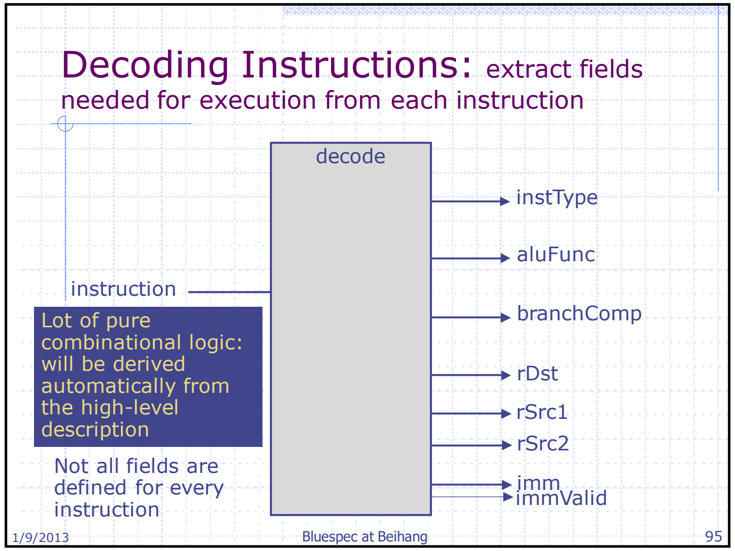
performance?

The whole system was described using one rule; lots of big combinational functions



Single-Cycle SMIPS *code structure (simplified)*

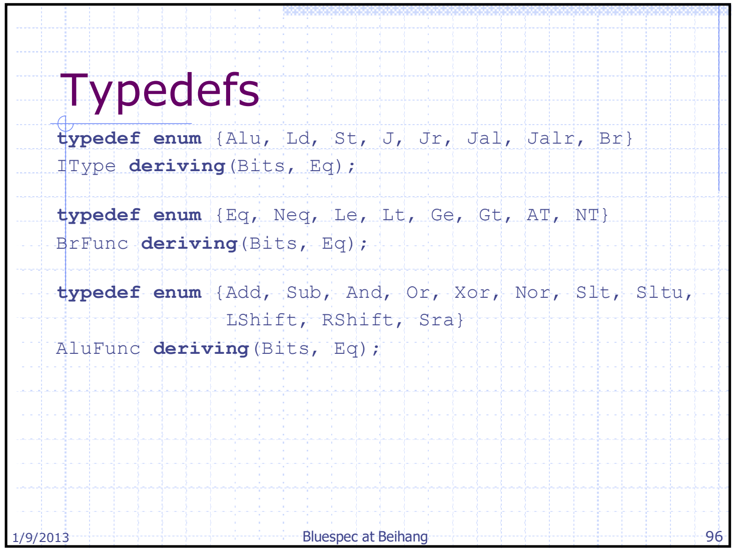
```
module mkProc(Proc);
  Reg#(Addr)   pc <- mkRegU;
  RFile        rf <- mkRFile;
  IMemory      iMem <- mkIMemory;
  DMemory      dMem <- mkDMemory;

  rule doProc(Cop.started);
    let inst = iMem.req(pc);
    let dInst = decode(inst);
    let rVal1 = rf.rd1(validRegValue(dInst.rSrc1));
    let rVal2 = rf.rd2(validRegValue(dInst.rSrc2));
    let eInst = exec(dInst, rVal1, rVal2, pc, ?);

    update rf, pc and dMem
```



Decoding Instructions: extract fields needed for execution from each instruction

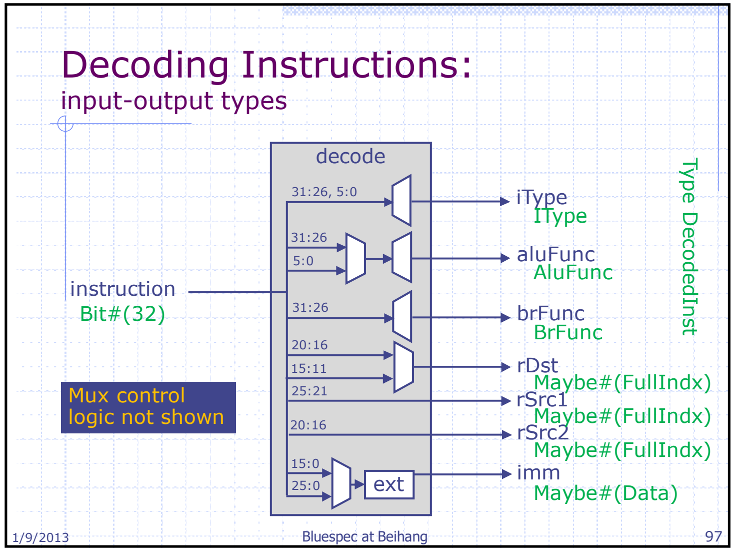
Lot of pure combinational logic: will be derived automatically from the high-level description

Not all fields are defined for every instruction



Typedefs

```
typedef enum {Alu, Ld, St, J, Jr, Jal, Jalr, Br}
IType deriving(Bits, Eq);

typedef enum {Eq, Neq, Le, Lt, Ge, Gt, AT, NT}
BrFunc deriving(Bits, Eq);

typedef enum {Add, Sub, And, Or, Xor, Nor, Slt, Sltu,
              LShift, RShift, Sra}
AluFunc deriving(Bits, Eq);
```

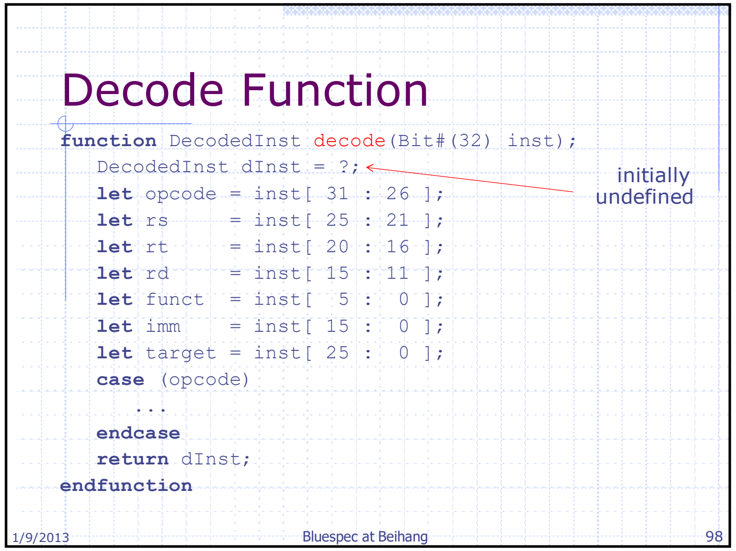


Decoding Instructions:

input-output types

Mux control logic not shown

97

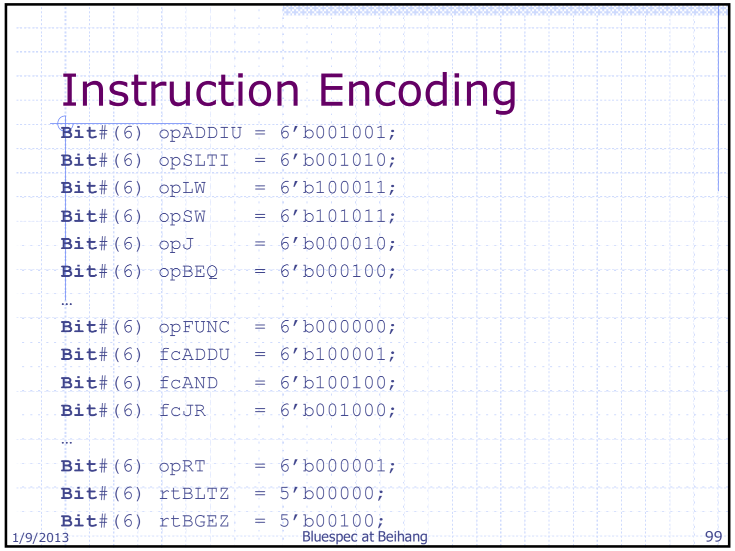


Decode Function

```
function DecodedInst decode(Bit#(32) inst);
   DecodedInst dInst = ?;
   let opcode = inst[ 31 : 26 ];
   let rs     = inst[ 25 : 21 ];
   let rt     = inst[ 20 : 16 ];
   let rd     = inst[ 15 : 11 ];
   let funct  = inst[  5 :  0 ];
   let imm    = inst[ 15 :  0 ];
   let target = inst[ 25 :  0 ];
   case (opcode)
      ...
   endcase
   return dInst;
endfunction
```

initially undefined

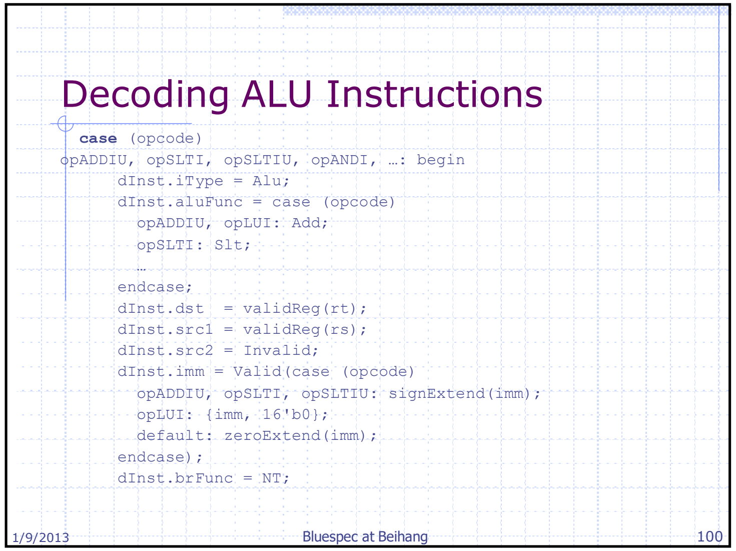
98



Instruction Encoding

```
Bit#(6) opADDIU = 6’b001001;
Bit#(6) opSLTI  = 6’b001010;
Bit#(6) opLW    = 6’b100011;
Bit#(6) opSW    = 6’b101011;
Bit#(6) opJ     = 6’b000010;
Bit#(6) opBEQ   = 6’b000100;
…
Bit#(6) opFUNC  = 6’b000000;
Bit#(6) fcADDU  = 6’b100001;
Bit#(6) fcAND   = 6’b100100;
Bit#(6) fcJR    = 6’b001000;
…
Bit#(6) opRT    = 6’b000001;
Bit#(6) rtBLTZ  = 5’b00000;
Bit#(6) rtBGEZ  = 5’b00100;
```

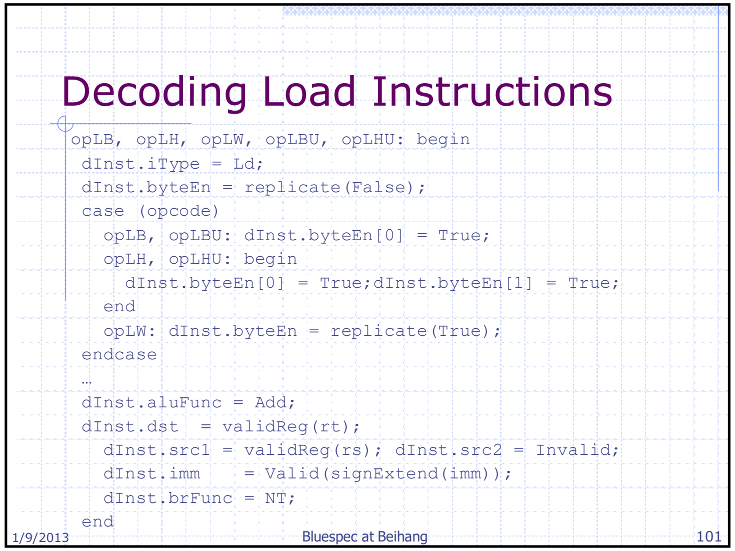
99



Decoding ALU Instructions

```
  case (opcode)
opADDIU, opSLTI, opSLTIU, opANDI, …: begin
      dInst.iType = Alu;
      dInst.aluFunc = case (opcode)
        opADDIU, opLUI: Add;
        opSLTI: Slt;
        …
      endcase;
      dInst.dst  = validReg(rt);
      dInst.src1 = validReg(rs);
      dInst.src2 = Invalid;
      dInst.imm = Valid(case (opcode)
        opADDIU, opSLTI, opSLTIU: signExtend(imm);
        opLUI: {imm, 16'b0};
        default: zeroExtend(imm);
      endcase);
      dInst.brFunc = NT;
```

100

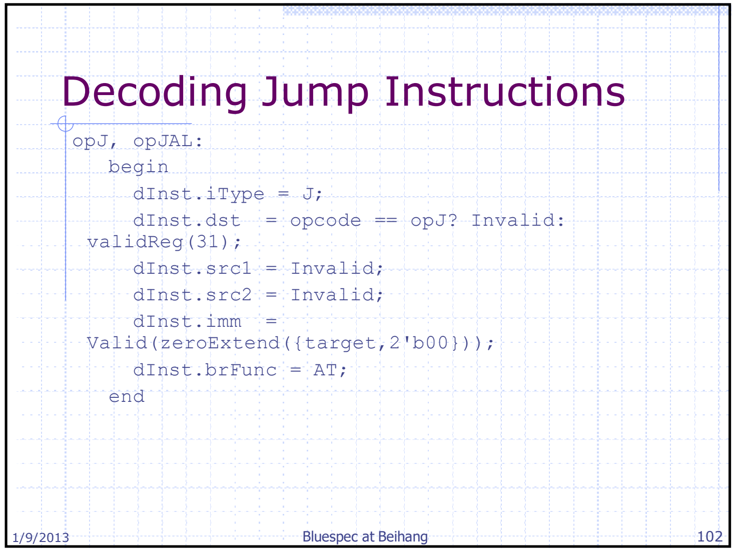


Decoding Load Instructions

```
opLB, opLH, opLW, opLBU, opLHU: begin
 dInst.iType = Ld;
 dInst.byteEn = replicate(False);
 case (opcode)
   opLB, opLBU: dInst.byteEn[0] = True;
   opLH, opLHU: begin
     dInst.byteEn[0] = True;dInst.byteEn[1] = True;
   end
   opLW: dInst.byteEn = replicate(True);
 endcase
 …
 dInst.aluFunc = Add;
 dInst.dst  = validReg(rt);
   dInst.src1 = validReg(rs); dInst.src2 = Invalid;
   dInst.imm    = Valid(signExtend(imm));
   dInst.brFunc = NT;
 end
```

1/9/2013 Bluespec at Beihang 101

101

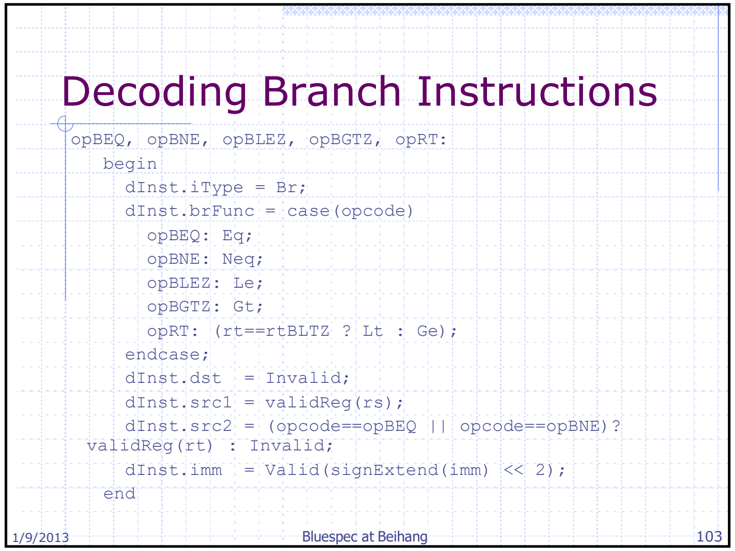


Decoding Jump Instructions

```
opJ, opJAL:
   begin
     dInst.iType = J;
     dInst.dst  = opcode == opJ? Invalid:
 validReg(31);
     dInst.src1 = Invalid;
     dInst.src2 = Invalid;
     dInst.imm  =
 Valid(zeroExtend({target,2'b00}));
     dInst.brFunc = AT;
   end
```

1/9/2013 Bluespec at Beihang 102

102

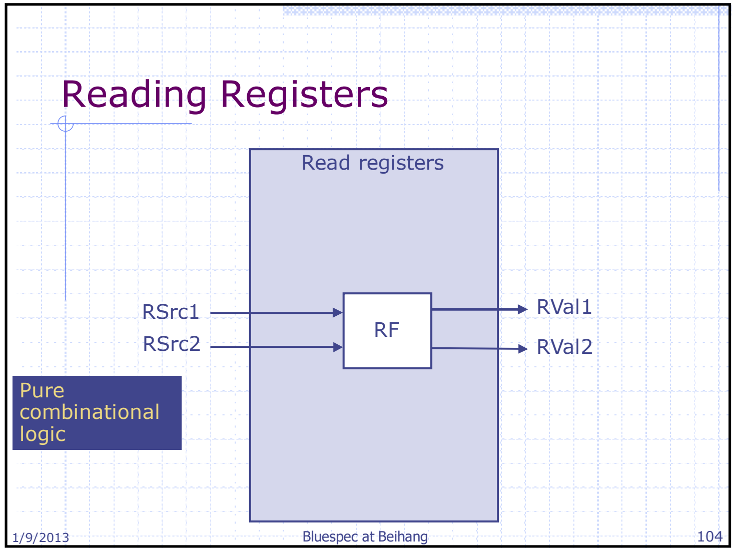


Decoding Branch Instructions

```
opBEQ, opBNE, opBLEZ, opBGTZ, opRT:
   begin
     dInst.iType = Br;
     dInst.brFunc = case(opcode)
       opBEQ: Eq;
       opBNE: Neq;
       opBLEZ: Le;
       opBGTZ: Gt;
       opRT: (rt==rtBLTZ ? Lt : Ge);
     endcase;
     dInst.dst  = Invalid;
     dInst.src1 = validReg(rs);
     dInst.src2 = (opcode==opBEQ || opcode==opBNE)?
  validReg(rt) : Invalid;
     dInst.imm  = Valid(signExtend(imm) << 2);
   end
```

1/9/2013 Bluespec at Beihang 103

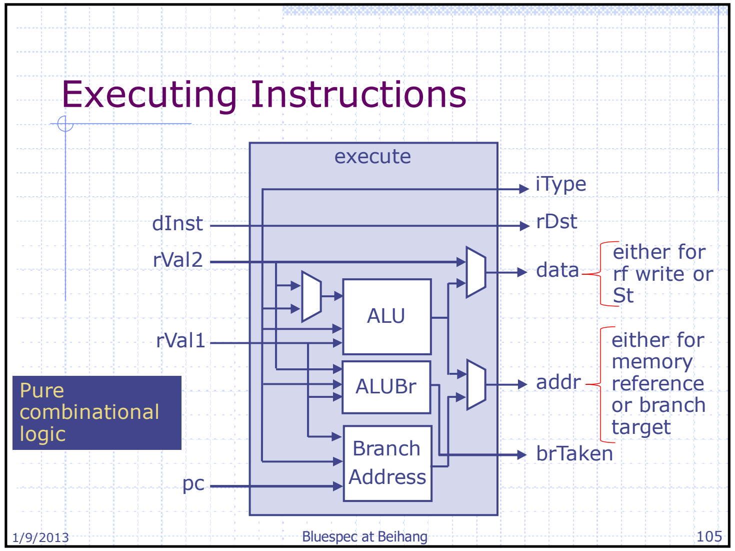
103



Reading Registers

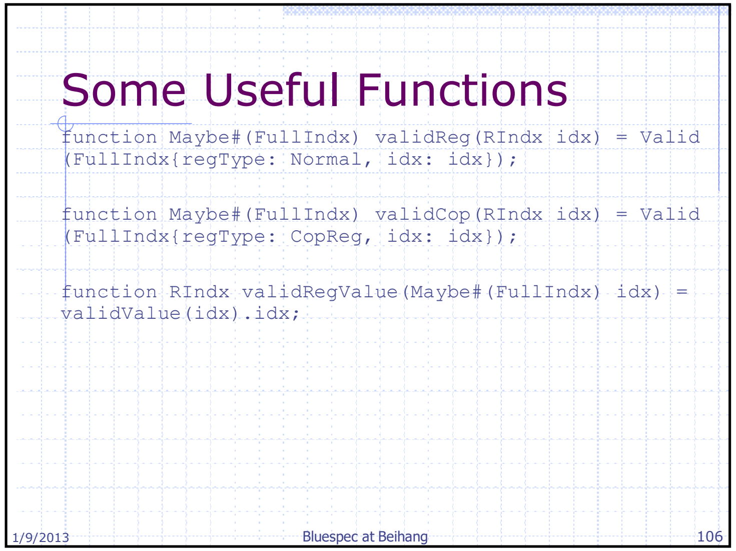
1/9/2013 Bluespec at Beihang 104

104



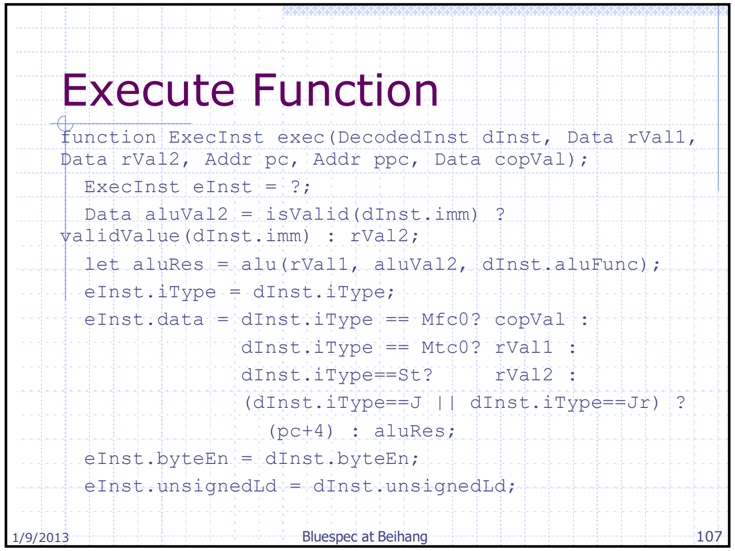
Executing Instructions

Pure combinational logic



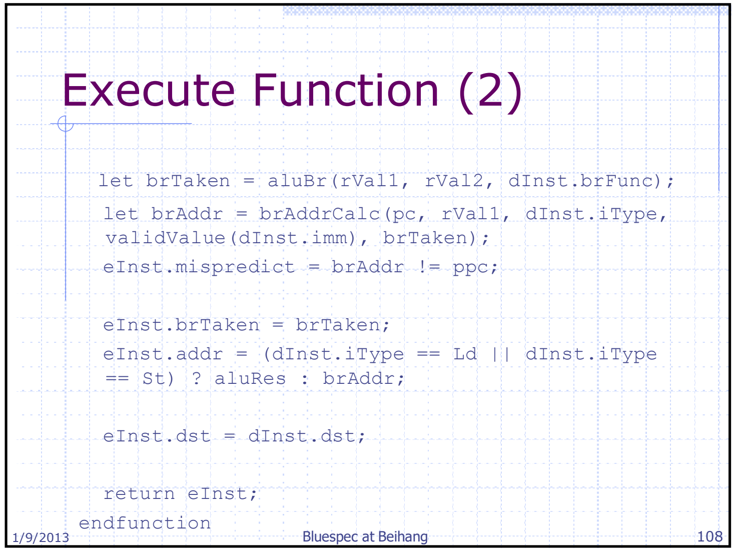
Some Useful Functions

```
function Maybe#(FullIndx) validReg(RIndx idx) = Valid
(FullIndx{regType: Normal, idx: idx});

function Maybe#(FullIndx) validCop(RIndx idx) = Valid
(FullIndx{regType: CopReg, idx: idx});

function RIndx validRegValue(Maybe#(FullIndx) idx) =
validValue(idx).idx;
```



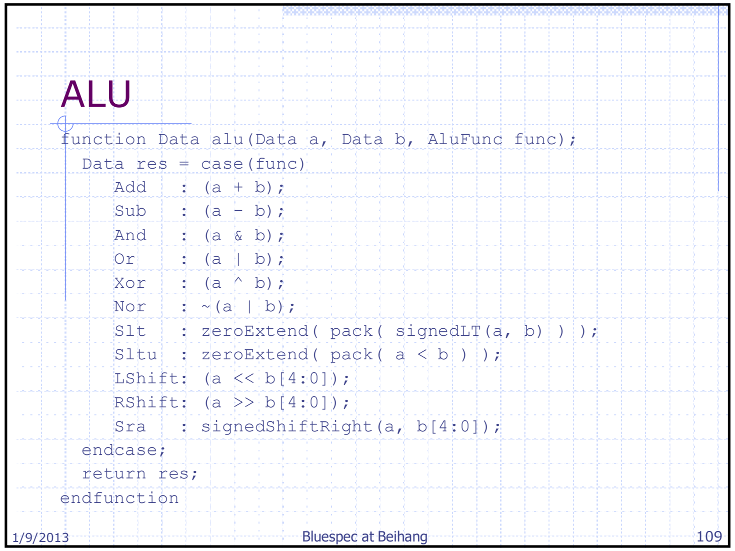
Execute Function

```
function ExecInst exec(DecodedInst dInst, Data rVal1,
Data rVal2, Addr pc, Addr ppc, Data copVal);
  ExecInst eInst = ?;
  Data aluVal2 = isValid(dInst.imm) ?
validValue(dInst.imm) : rVal2;
  let aluRes = alu(rVal1, aluVal2, dInst.aluFunc);
  eInst.iType = dInst.iType;
  eInst.data = dInst.iType == Mfc0? copVal :
               dInst.iType == Mtc0? rVal1 :
               dInst.iType==St?     rVal2 :
               (dInst.iType==J || dInst.iType==Jr) ?
                 (pc+4) : aluRes;
  eInst.byteEn = dInst.byteEn;
  eInst.unsignedLd = dInst.unsignedLd;
```



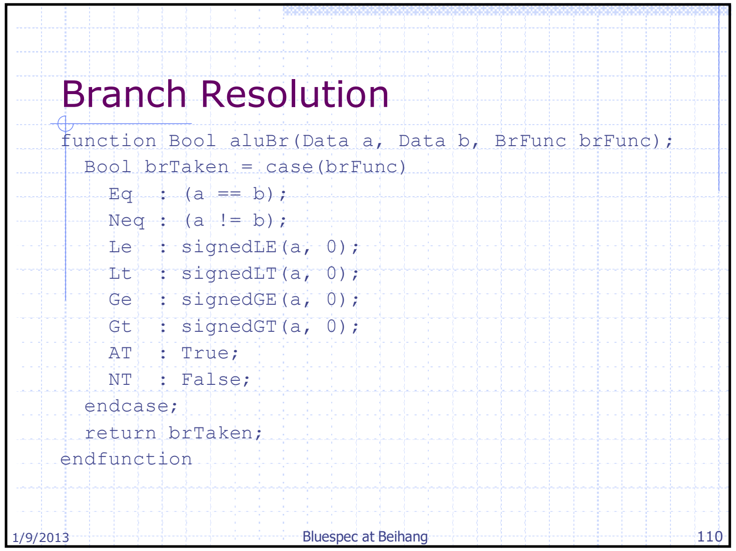
Execute Function (2)

```
   let brTaken = aluBr(rVal1, rVal2, dInst.brFunc);
   let brAddr = brAddrCalc(pc, rVal1, dInst.iType,
   validValue(dInst.imm), brTaken);
   eInst.mispredict = brAddr != ppc;

   eInst.brTaken = brTaken;
   eInst.addr = (dInst.iType == Ld || dInst.iType
   == St) ? aluRes : brAddr;

   eInst.dst = dInst.dst;

   return eInst;
 endfunction
```



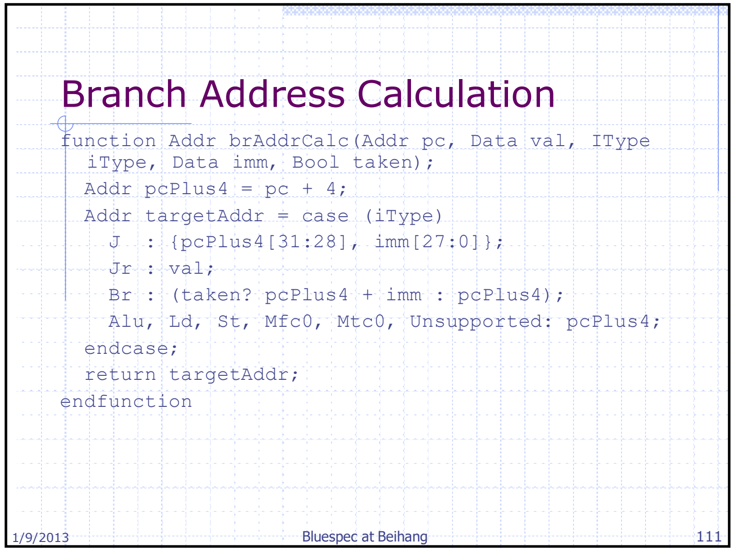
ALU

```
function Data alu(Data a, Data b, AluFunc func);
  Data res = case(func)
     Add   : (a + b);
     Sub   : (a - b);
     And   : (a & b);
     Or    : (a | b);
     Xor   : (a ^ b);
     Nor   : ~(a | b);
     Slt   : zeroExtend( pack( signedLT(a, b) ) );
     Sltu  : zeroExtend( pack( a < b ) );
     LShift: (a << b[4:0]);
     RShift: (a >> b[4:0]);
     Sra   : signedShiftRight(a, b[4:0]);
  endcase;
  return res;
endfunction
```



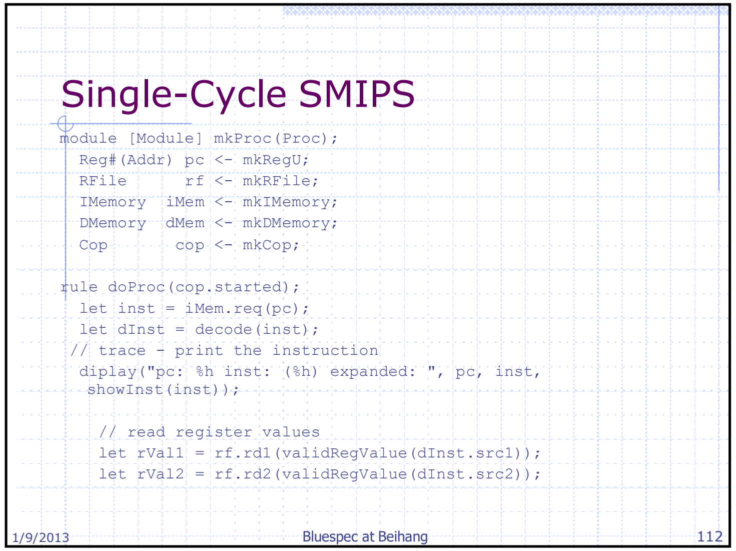
Branch Resolution

```
function Bool aluBr(Data a, Data b, BrFunc brFunc);
  Bool brTaken = case(brFunc)
    Eq  : (a == b);
    Neq : (a != b);
    Le  : signedLE(a, 0);
    Lt  : signedLT(a, 0);
    Ge  : signedGE(a, 0);
    Gt  : signedGT(a, 0);
    AT  : True;
    NT  : False;
  endcase;
  return brTaken;
endfunction
```



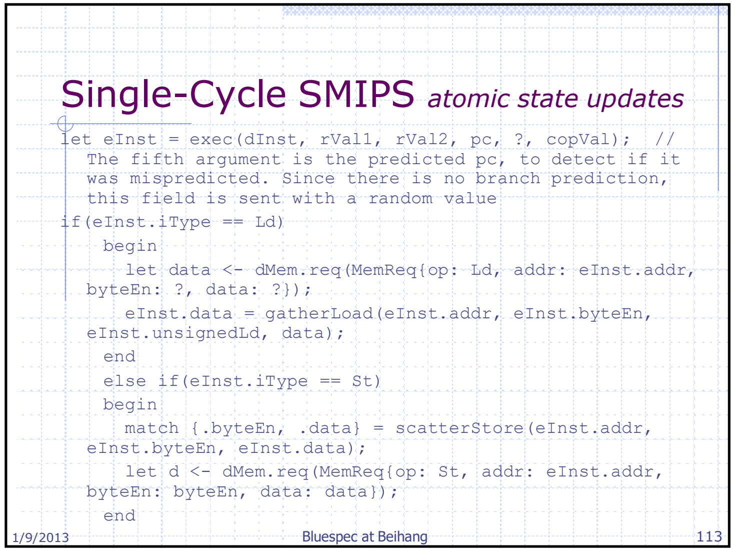
Branch Address Calculation

```
function Addr brAddrCalc(Addr pc, Data val, IType
  iType, Data imm, Bool taken);
  Addr pcPlus4 = pc + 4;
  Addr targetAddr = case (iType)
    J  : {pcPlus4[31:28], imm[27:0]};
    Jr : val;
    Br : (taken? pcPlus4 + imm : pcPlus4);
    Alu, Ld, St, Mfc0, Mtc0, Unsupported: pcPlus4;
  endcase;
  return targetAddr;
endfunction
```



Single-Cycle SMIPS

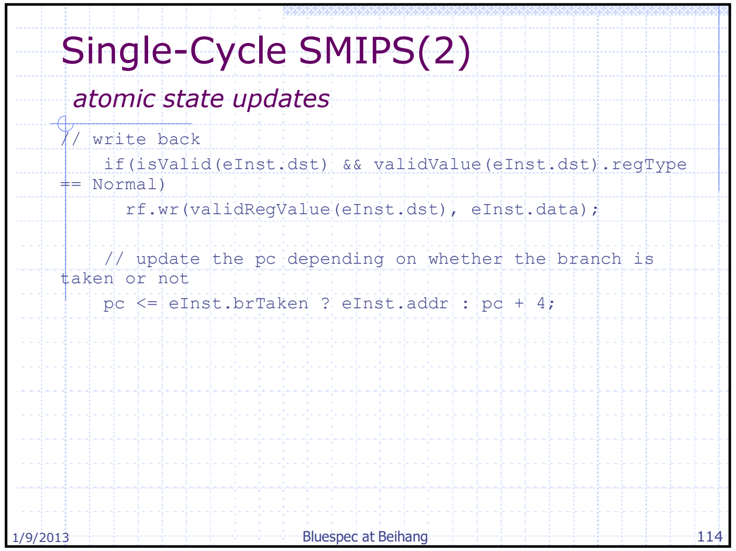
```
module [Module] mkProc(Proc);
  Reg#(Addr) pc <- mkRegU;
  RFile      rf <- mkRFile;
  IMemory  iMem <- mkIMemory;
  DMemory  dMem <- mkDMemory;
  Cop       cop <- mkCop;

rule doProc(cop.started);
  let inst = iMem.req(pc);
  let dInst = decode(inst);
 // trace - print the instruction
  diplay("pc: %h inst: (%h) expanded: ", pc, inst,
   showInst(inst));

    // read register values
    let rVal1 = rf.rd1(validRegValue(dInst.src1));
    let rVal2 = rf.rd2(validRegValue(dInst.src2));
```



Single-Cycle SMIPS *atomic state updates*

```
let eInst = exec(dInst, rVal1, rVal2, pc, ?, copVal);  //
  The fifth argument is the predicted pc, to detect if it
  was mispredicted. Since there is no branch prediction,
  this field is sent with a random value
if(eInst.iType == Ld)
    begin
      let data <- dMem.req(MemReq{op: Ld, addr: eInst.addr,
  byteEn: ?, data: ?});
      eInst.data = gatherLoad(eInst.addr, eInst.byteEn,
  eInst.unsignedLd, data);
    end
    else if(eInst.iType == St)
    begin
      match {.byteEn, .data} = scatterStore(eInst.addr,
  eInst.byteEn, eInst.data);
      let d <- dMem.req(MemReq{op: St, addr: eInst.addr,
  byteEn: byteEn, data: data});
    end
```

113

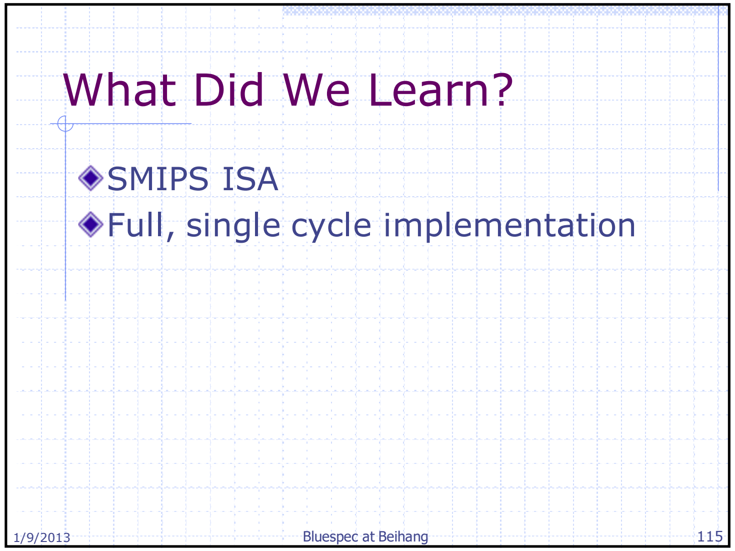


Single-Cycle SMIPS(2)

atomic state updates

```
// write back
    if(isValid(eInst.dst) && validValue(eInst.dst).regType
== Normal)
      rf.wr(validRegValue(eInst.dst), eInst.data);

    // update the pc depending on whether the branch is
taken or not
    pc <= eInst.brTaken ? eInst.addr : pc + 4;
```

114



What Did We Learn?

- SMIPS ISA
- Full, single cycle implementation

115