6.S078 - Computer Architecture:
A Constructive Approach

Combinational circuits

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Course Staff

Instructors

Arvind
arvind@csail.mit.edu

Joel Emer
emer@csail.mit.edu

Li-Shiuan Peh
peh@csail.mit.edu

Teaching Assistants

Abhinav Agarwal
abhiag@csail.mit.edu

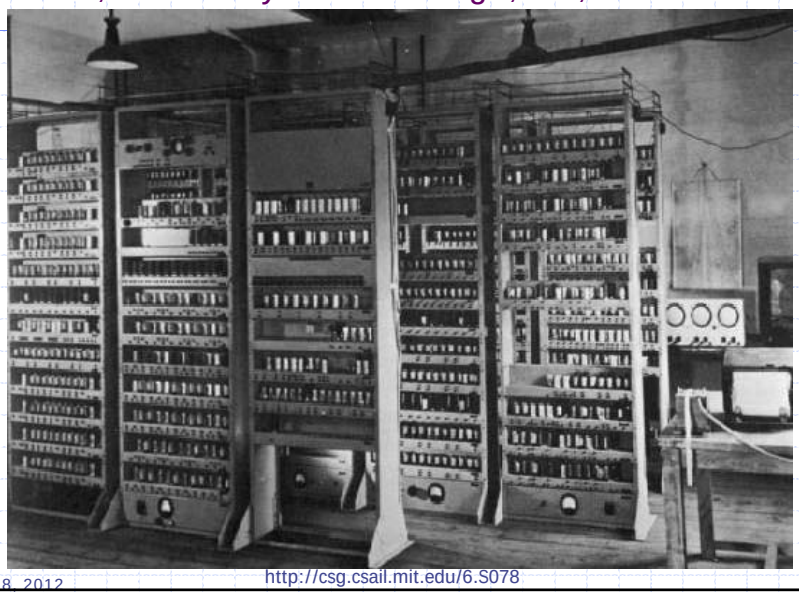
Myron King
mdk@csail.mit.edu

Administration

Sally Lee
sally@csail.mit.edu

Computing Devices Then…

EDSAC, University of Cambridge, UK, 1949



Computing Devices Now

Dramatic progress in terms of size, speed, cost, reliability

Computer architecture is about designing machines to meet some power, performance, cost and size constraints

The basics: How does a program execute on hardware

- A C program to add two arrays:

```
void vvadd( int n, int a[],
            int b[], int c[] )
  { int i;
    for ( i = 0; i < n; i++ )
      c[i] = a[i] + b[i];
  }
```

- The hardware must know, for example,
  - How to add and compare two numbers
  - Must have a place to keep the program and data
  - Must know how to fetch instructions and data
  - Must know how to sequence instructions:
    - Fetch a[i], Fetch b[i], add, store results in c[i], increment i, …

Computer Architecture is learning about how programs execute and designing hardware to execute them efficiently

Instruction Set Architecture (ISA)

- Computer architecture is the discipline of *designing and implementing* interfaces through which hardware and software interact
- This interface is often referred to as the Instruction Set Architecture (ISA)
  - Examples: Intel's IA-32, ARM, ARM-Thumb, PowerPC
  - In this class we will use SMIPS, a subset of MIPS ISA
- Implementations are deeply affected by the technology issues; we will assume a simple and abstract model of technology based on Silicon

Computer Architecture

A method of constructing machines: Machine descriptions which can be simulated in software and synthesized into hardware

Quantitative evaluation: To what extent designs meet various design criteria

Testing and verification: Does the machine do what it is supposed to do

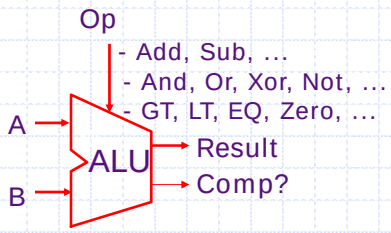
The goals of this subject

- Learn a constructive approach to studying computer architecture
- Learn a new method of describing architectures where there is less emphasis on figures/diagrams and more emphasis on executable descriptions
  - Each architecture and each part of it would be defined as executable code in Bluespec
- Learn about test benches, including designing your own
- Learn about quantitative evaluation of your designs

By the end-of-the-term you will design six or more different computers of increasing complexity and performance, and you will quantitatively evaluate the performance of your C programs on these machines

All the designs you do in this course can be implemented on FPGAs or realized as ASICs without significant additional effort. However, lack of time won't permit us to explore this aspect.

Arithmetic-Logic Unit (ALU)



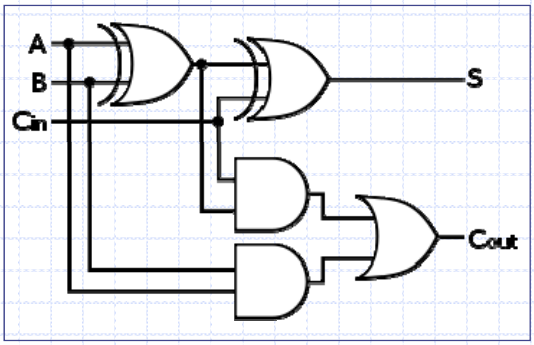
ALU performs all the arithmetic and logical functions

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

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



Full Adder: A one-bit adder

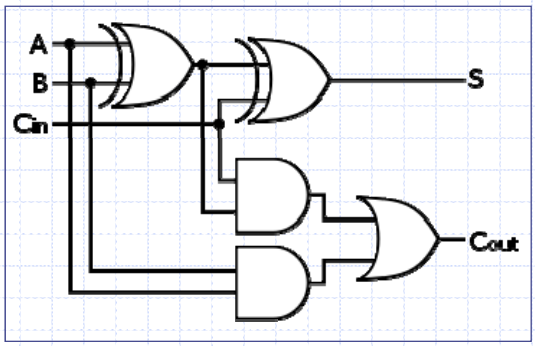
corrected

```
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` declaration says that a is one bit wide

`{c_out,s}` represents bit concatenation

How big is `{c_out,s}`?



Types

- Every expression and variable in a Bluespec program has a type; sometimes it is specified explicitly and sometimes it is deduced by the compiler
- 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)
```

- Thus we say an expression has a type or belongs to a type

The type of each expression is unique

Type declaration versus deduction

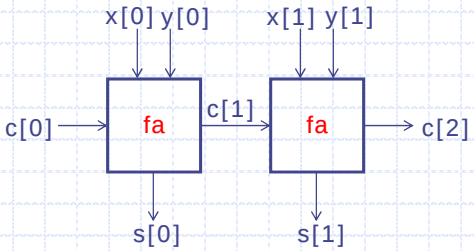
- The programmer writes down types of some expressions in a program and the compiler deduces the types of the rest of expressions
- If the type deduction cannot be performed or the type declarations are inconsistent then the compiler complains

```
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
```


2-bit Ripple-Carry Adder

```
function Bit#(3) add(Bit#(2) x, Bit#(2) y, Bit#(1) c0);
   Bit#(2) s = 0;
   Bit#(3) c; 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
```

fa is like a blackbox, its internals are not visible to the user. fa can be used as long as we understand its type signature



"let" syntax

- The "let" syntax: 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]);`

Parameterized types:

- A type declaration itself can be parameterized – the 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), …`

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; 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
// 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);
```


valueOf(w) versus w

- Each expression has a type and a value and these come from two entirely disjoint worlds
- n in Bit#(n) 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

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 space 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; 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
```

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
```


Static Elaboration phase

- When Bluespec program are compiled, first type checking is done and then the compiler gets rid of many different 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];
```