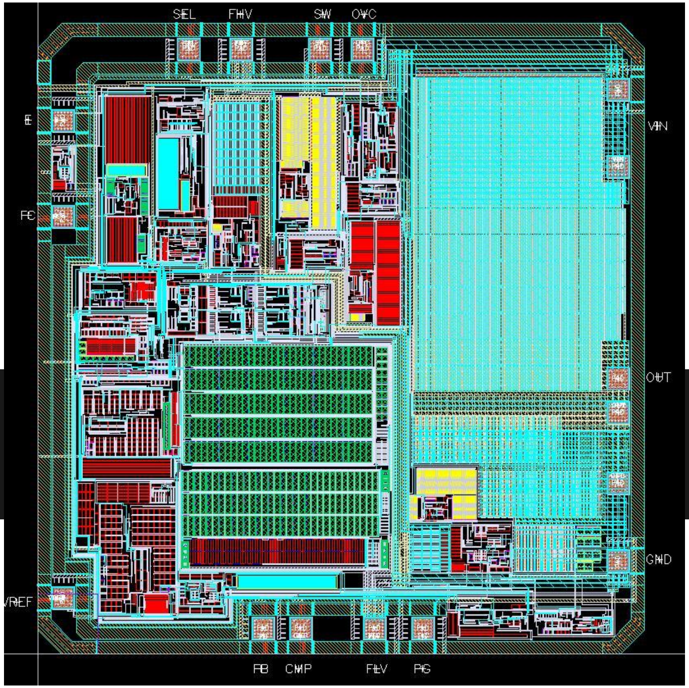
SUPPLEMENTAL

LIFE BEYOND VERILOG

ECE4514

DIGITAL DESIGN 2



Oh, Verilog

- Now that you’ve had a some time to get used to Verilog-2001 you’ve probably noticed a few flaws.

- What do you dislike about Verilog?

Personality Problems

- It’s not always clear what will/won’t synthesize
  - While loops, divide/modulo operators, implicit state machines, complex for loops, etc.

- Sometimes it would be nice if you could just say what you intended, instead of having to imply it.
  - Make this always block combinational
  - Make this infer a flip-flop / make this infer a latch
  - Make this case statement parallel**
  - Make this case statement have priority**

** Without relying on “hidden” synthesis pragmas!

Personality Problems

- It’s easy to get into trouble if you’re not careful
  - Inferring latches
  - Mixing blocking & non-blocking statements
  - Mixing synchronous & asynchronous triggers in sensitivity lists
- Some things are just misleading
  - I can call something a “reg” but it’s not necessarily a register?

SystemVerilog – The ‘new’ Verilog

- New IEEE Standard introduced in 2005
- “Successor” to Verilog – merged into Verilog-2009

- Mostly the same syntax/semantics
- Backwards compatible with most Verilog code

- A lot of new features for Verification (testbenches)
- Tweaks to existing Verilog constructs to make them easier to use
- New datatypes

SystemVerilog – Datatypes [1]

- New variable types: LOGIC, BIT

- Logic can be used
  - As left side in behavioral block
  - As left side in continuous assignment
  - As output of a structural module

- No more worrying about declaring wire vs reg!
- Less confusing – the name doesn't trick people into thinking it is always a register, like reg does.
- bit – Like logic, but cannot have x or z values

SystemVerilog – Datatypes [2]

- TYPEDEF – C-style defined data types

```
typedef pixel logic[23:0];
pixel a_pixel;
pixel[1920:1][1080:1] my_display;
```

Why not just do this with a macro?

- Typedefs do not have the global scope problem of macros.
- Typedefs can be evaluated during Syntax Check rather than during Elaboration, so they are easier to debug.

SystemVerilog – Datatypes [3]

- ENUM – Enumerated types

```
typedef enum logic[2:0] {RESET, COMPUTE1, COMPUTE2,
SEND, ERROR} states_t;

states_t current_state, next_state;
…
    if(current_state == COMPUTE1)
        next_state = COMPUTE2;
…
```

SystemVerilog – Datatypes [4]

- Structures & Unions

```
typedef struct packed {
        logic [2:0] packet_type;
        logic [3:0] dest_addr;
        logic [47:0] data_payload;
} router_packet;

case(my_packet.packet_type)
        DATA3:  if(my_packet.dest_addr == MY_ADDR)
                        …
                …
```

SystemVerilog – Always Constructs [1]

- Always Blocks – can specify what you’re trying to do!

- Combinational Block:

```
always_comb begin //Can omit sensitivity list!
    a = b;      //Tool adds it like @(*)
    c = ~d;
end
```

SystemVerilog – Always Constructs [2]

```
always_ff@(posedge clk) //still need sensitivity list
    state <= next_state; //for flip-flops. Why?

always_latch
    if(en)
        d_out <= d_in;
```

This can't "force" the synthesis tool to synthesize flip flops if you describe latch-like behavior

However, it can warn you that what you described didn't match what you said you wanted!

SystemVerilog – Control Constructs[1]

- Case and If/Else have priority by default in Verilog.
- To match this behavior in synthesis, we need a cascade of muxes.
- Designers commonly use synopsys parallel_case to force the synthesizer to make a single mux instead.

- Synthesizer pragmas give different information to the synthesis tool than to the simulator.
- This is fundamentally bad. We want synthesis tool and simulator to have the same information!

SystemVerilog – Control Constructs[2]

- *unique* keyword modifier

```
unique case (sel)
        CASE1: …
        CASE2: …
        CASE3: …
endcase
```

- unique tells the synthesizer *and simulator* that one, and only one, case will be selected – ignore priority so you can synthesis a single parallel mux!
- Also works with if: unique if(…) …

SystemVerilog – Control Constructs[3]

- *priority* keyword modifier

```
priority case (sel)
    CASE1: …
    CASE2: …
endcase
```

- priority tells the synthesizer *and simulator* that at least one of the cases will always match. If this doesn't happen in simulation it will warn you.
- Also works with if: priority if(…) …
- Easy way to avoid accidental latches!

SystemVerilog – Interfaces

- Can use multi-dimensional arrays in I/O

```
module(input [1:0] a[9:0][3:0], output b);
```

- Can define interfaces separately from modules
  - Allow an interface to be re-used in multiple modules

```
interface intf;
    logic a, b;
    modport in  (input a, output b);
    modport out (input b, output a);
endinterface
```

SystemVerilog – Verification

- Most of SystemVerilog's new features are in non-synthesizable constructs for testbenches
- SystemVerilog is sometimes referred to as an "HVL" – Hardware Verification Language.

- New verification features include
  - Dynamic & associative arrays, Classes
  - FIFOs, semaphores, mailboxes
  - Assertions and time sequencing
  - Built-in support for computing test coverage
  - Extended support for easy generation of random test vectors

SystemVerilog – Current State

- SystemVerilog is already widely used in industry for designing testbenches
- So why, you ask, did we learn Verilog-2001 instead of SystemVerilog/Verilog-2009?
- Unfortunately, many companies are using not yet adopting SystemVerilog for synthesis

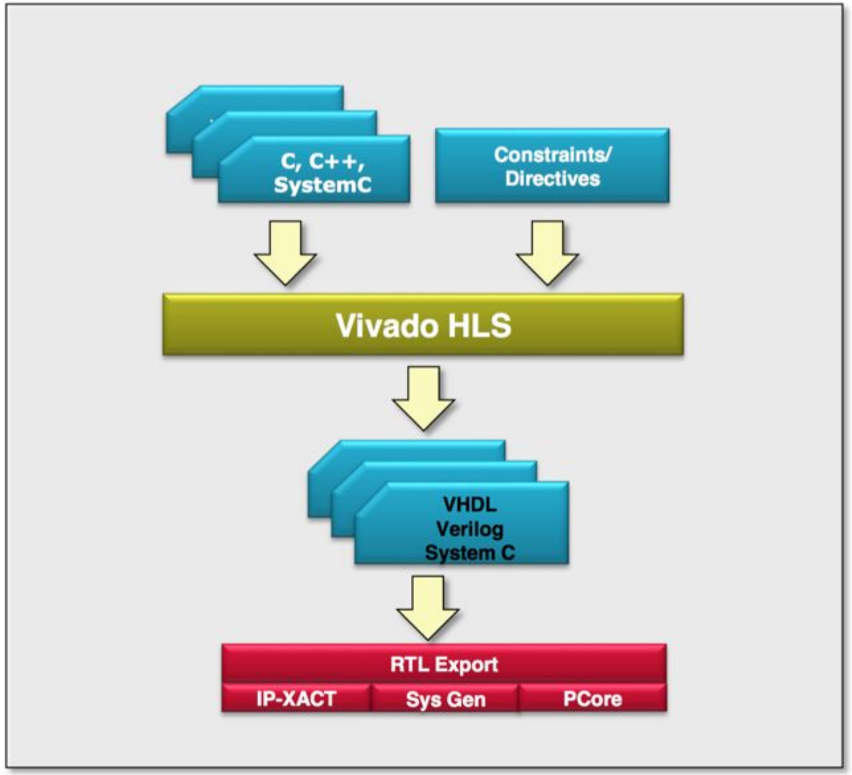
Beyond Verilog

High-Level Language Synthesis

High-Level Synthesis: HLS

- **High-Level Synthesis**
  - Creates an RTL implementation from C level source code
  - Extracts control and dataflow from the source code
  - Implements the design based on defaults and user applied directives

- **Many implementation are possible from the same source description**
  - Smaller designs, faster designs, optimal designs
  - Enables design exploration



Design Exploration with Directives

One body of code:
Many hardware outcomes

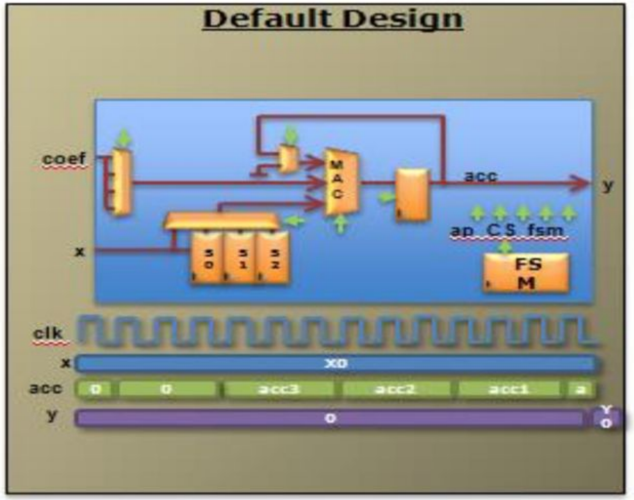
```
...
loop: for (i=3;i>=0;i--) {
  if (i==0) {
    acc+=x*c[0];
    shift_reg[0]=x;
  } else {
    shift_reg[i]=shift_reg[i-1];
    acc+=shift_reg[i]*c[i];
  }
}
....
```

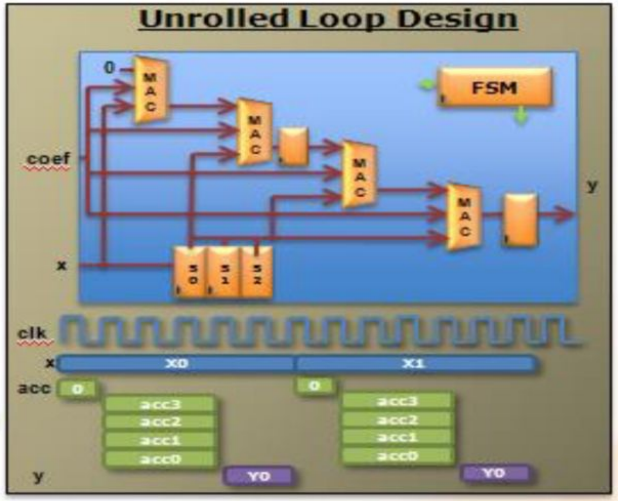
Before we get into details, let's look under the hood ….

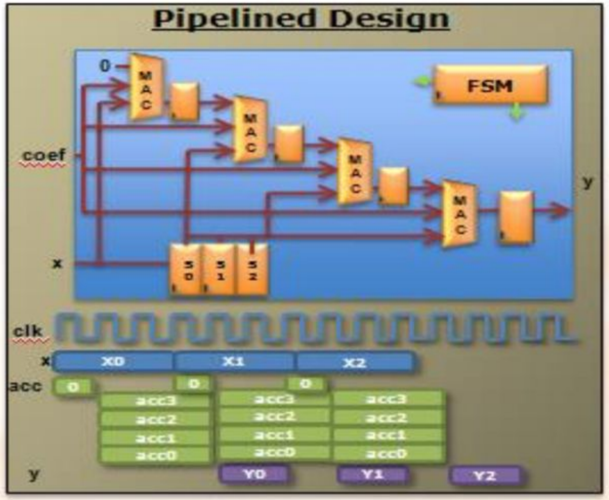
The same hardware is used for each iteration of the loop:
- **Small area**
- **Long latency**
- **Low throughput**

Different hardware is used for each iteration of the loop:
- **Higher area**
- **Short latency**
- **Better throughput**

Different iterations are executed concurrently:
- **Higher area**
- **Short latency**
- **Best throughput**







Introduction to High-Level Synthesis

- **How is hardware extracted from C code?**
  - Control and datapath can be extracted from C code at the top level
  - The same principles used in the example can be applied to sub-functions
    - At some point in the top-level control flow, control is passed to a sub-function
    - Sub-function may be implemented to execute concurrently with the top-level and or other sub-functions
- **How is this control and dataflow turned into a hardware design?**
  - Vivado HLS maps this to hardware through scheduling and binding processes
- **How is my design created?**
  - How functions, loops, arrays and IO ports are mapped?

HLS: Control Extraction

Code

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i]*c[i];
    }
  }
  *y=acc;
}
```

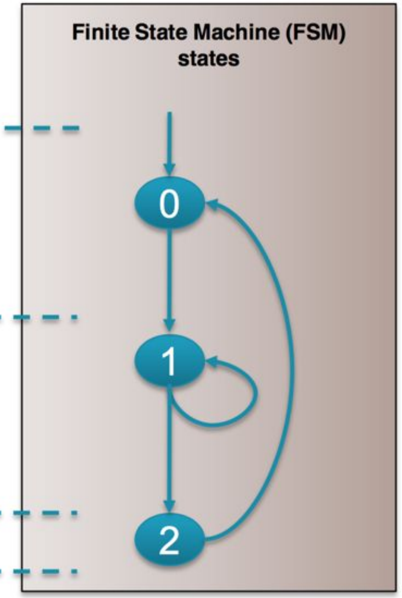
Function Start

For-Loop Start

For-Loop End

Function End

Control Behavior

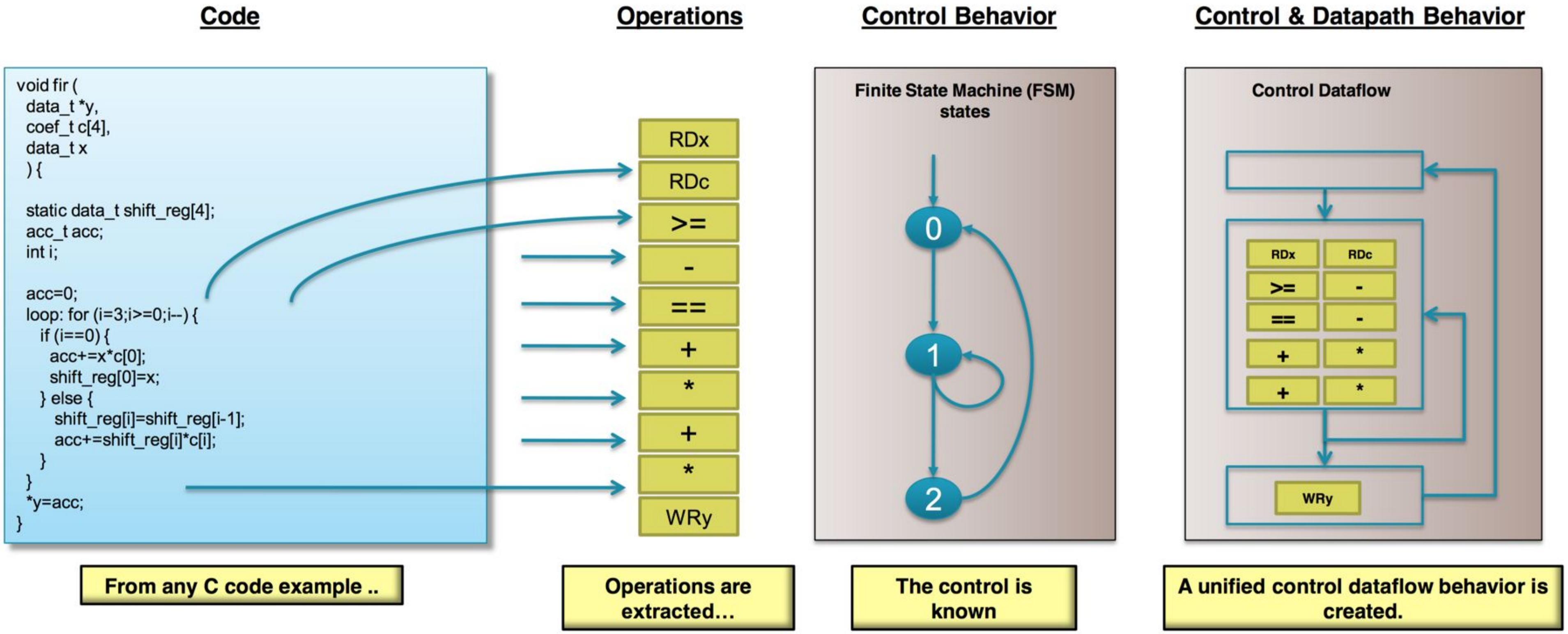


From any C code example ..

The loops in the C code correlated to states of behavior

This behavior is extracted into a hardware state machine

HLS: Control & Datapath Extraction


Code
Operations
Control Behavior
Control & Datapath Behavior
RDx
RDc
>=
-
==
+
*
+
*
WRy
Finite State Machine (FSM) states
0
1
2
Control Dataflow
RDx
RDc
>=
-
==
-
+
*
+
*
WRy
From any C code example ..
Operations are extracted…
The control is known
A unified control dataflow behavior is created.

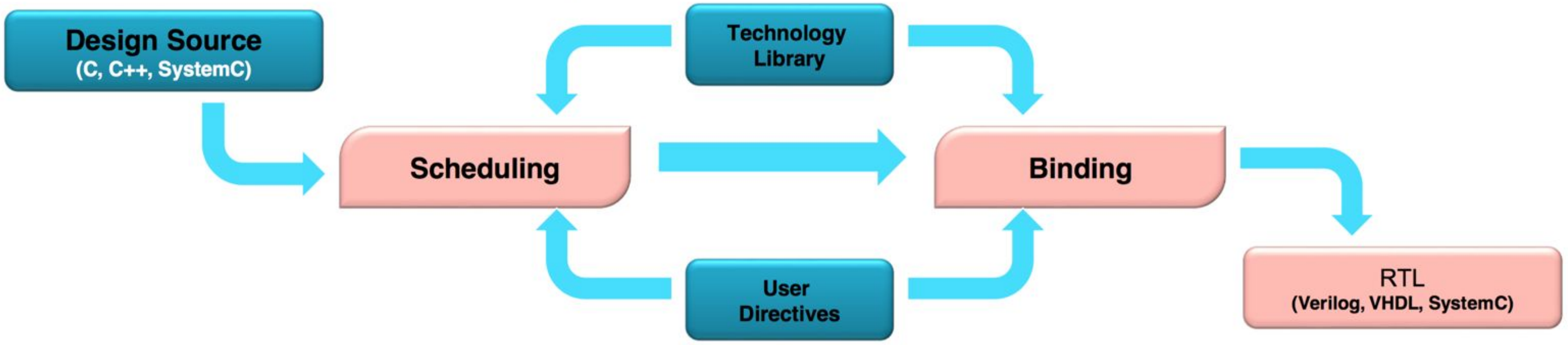
```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
       shift_reg[i]=shift_reg[i-1];
       acc+=shift_reg[i]*c[i];
    }
  }
  *y=acc;
}
```

High-Level Synthesis: Scheduling & Binding

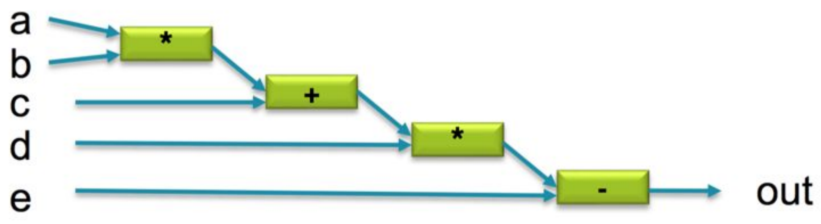
- **Scheduling & Binding**
  - Scheduling and Binding are at the heart of HLS
- **Scheduling determines in which clock cycle an operation will occur**
  - Takes into account the control, dataflow and user directives
  - The allocation of resources can be constrained
- **Binding determines which library cell is used for each operation**
  - Takes into account component delays, user directives

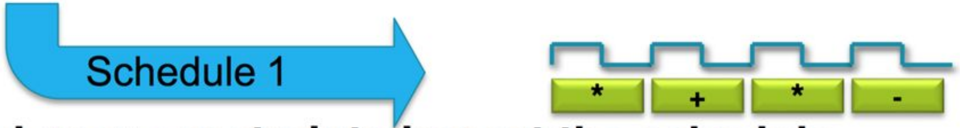


Scheduling

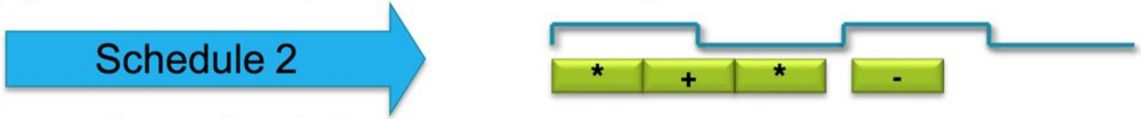
- **The operations in the control flow graph are mapped into clock cycles**

```
void foo (
 …
  t1 = a * b;
  t2 = c + t1;
  t3 = d * t2;
  out = t3 – e;
}
```





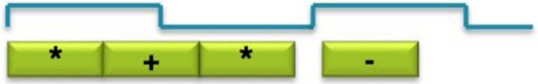
- **The technology and user constraints impact the schedule**
  - A faster technology (or slower clock) may allow more operations to occur in the same clock cycle



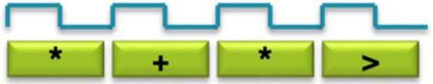
- **The code also impacts the schedule**
  - Code implications and data dependencies must be obeyed

Binding

- **Binding is where operations are mapped to cores from the hardware library**
  - Operators map to cores
- **Binding Decision: to share**
  - Given this schedule:

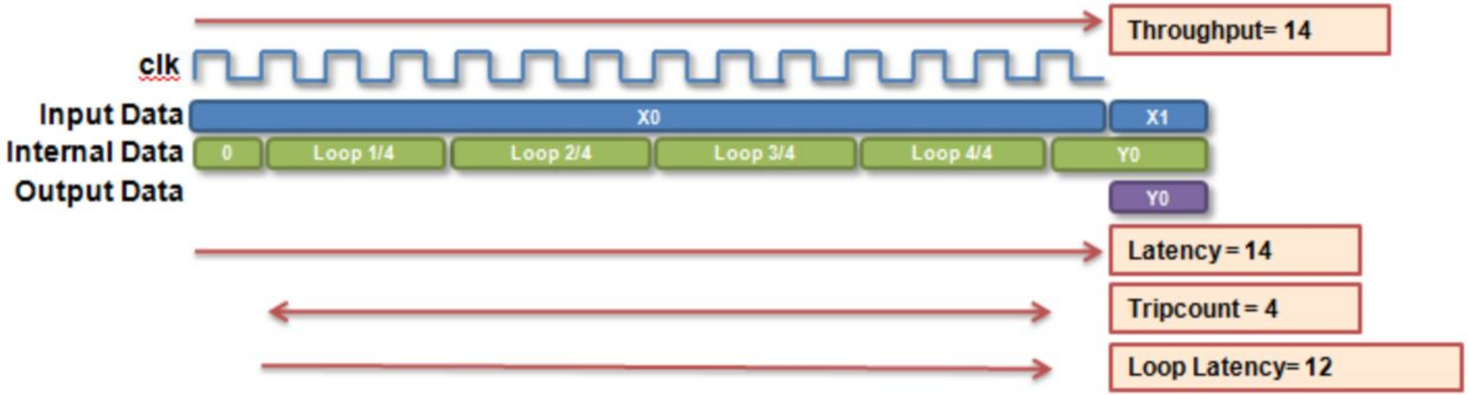


    - Binding must use 2 multipliers, since both are in the same cycle
    - It can decide to use an adder <u>and</u> subtractor <u>or</u> one addsub
- **Binding Decision: or not to share**
  - Given this schedule:



    - Binding may decide to share the multipliers (each is used in a different cycle)
    - Or it may decide the cost of sharing (muxing) would impact timing and it may decide not to share them
    - It may make this same decision in the first example above too

HLS Terminology for measuring in Clock Cycles



Latency	The number of cycles from input to output (final output of an array write)	14 cycles
Throughput	The number of cycle between new input samples (in this example it must wait for all operations to complete before it can read a new input)	14 cycles
Initiation Interval (II)	The number of cycles between new inputs to a pipeline (the same as throughput, but this term is used with pipelines).	Not shown in this example.
Data Rate	The 1/throughout * clock frequency	10ns clock => 7.14 Mhz, ((1/10e9)*14)
Trip count	The number of iterations in a loop	4
Loop Latency	The latency of the entire loop (divide by tripcount to get the latency for each loop iteration)	12 cycles

Undertaking a Project

Planning

- What platform should I use for this project?
  - Standard Cell, Custom ASIC, MPGA
  - FPGA, PLD
  - ROM
  - Software?

- What HDL / tools should I use?
  - Verilog, VHDL, SystemVerilog
  - HLS, OpenCL

Design Entry

- How should I architect this design?
  - Hierarchy
  - Abstraction level (structural, RTL, behavioral)
- How do I describe specific types of logic?
  - Datapaths
  - Control logic
  - State Machines
- How can I design for re-use?
  - Parameterization
  - Generated Instantiation

Functional Test

- Architecting Testbenches
  - File-based testing
  - Self-checking / Automated testbenches

- Designing Test Patterns
  - Exhaustive Testing vs Non-Exhaustive Testing
  - Selecting tests and writing a test plan
  - Identifying corner cases

- Debugging a Design
  - Reading waveforms
  - Identifying problems