Digital System Design

- FSMD Design: complex datapaths + complex control
- Controller Design Specified as ASM Chart Implemented with HDL
- So far, Datapath Design Accomplished in ad hoc Manner
- Use Behavioral Synthesis as more Formal Approach for Datapath Design
  - Resource Estimation
  - Resource Scheduling

Datapath Design

- Faced with problems of :
  1. **Constraints**: minimum clock frequency, maximum number of clock cycles, target device, resource limits (don't have an infinite number of logic cells available)
  2. **Execution unit architecture and number of resources**: fast adder? Slow adder? Pipelined or non-pipelined multiplier? SRAM versus registers? How many do I need based on constraints?
  3. **Scheduling** : what happens during each clock cycle?

Constraints

- Two *Constraints* that can be placed on a digital system design are clock period and clock cycle constraints
- A *Clock period constraint* will define the clock frequency.
  - Will affect the architecture of your execution units (fast adder versus slow adder, pipelined execution unit versus non-pipelined execution unit)
- A *Clock cycle constraint* limits the available number of clock cycles to perform operation - throughput
- Total computation time: (clock period×clock cycles)
- Other constraints: Power, device type, Input/Output

Resource Estimation

- Given constraints, would like a lower bound estimate on the number of resources needed
- Resource types: Registers, Execution units (adders, multipliers, etc)
- Lets do resource estimation for the equation below:

$$Y = a0 * x + a1 * x@1 + a2 * x@2 + a3 * x@3$$

FIR Filter Example

$$Y = a0 * x + a1 * x@1 + a2 * x@2 + a3 * x@3$$

The equation above is an equation for a 4-Tap Finite Impulse Response digital filter.

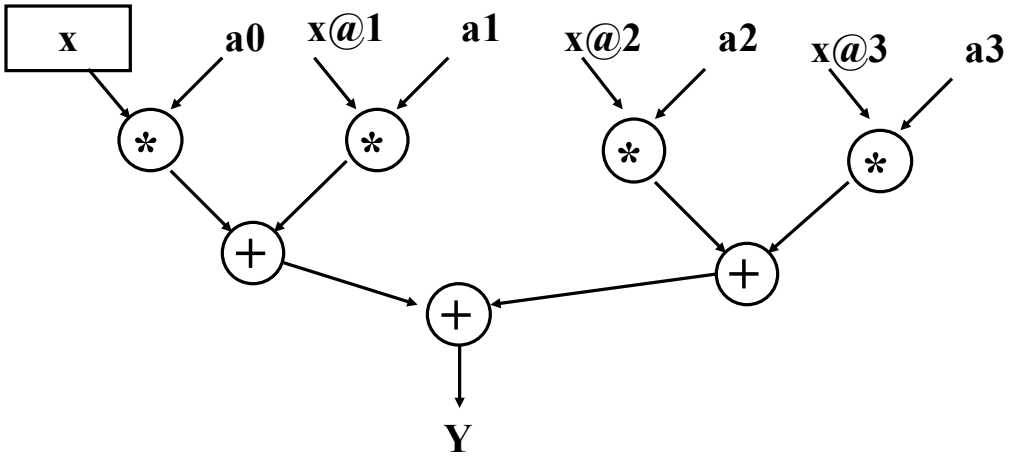
Each *sample period* a new value for **X** is input to the system. A sample period is measured in clock cycles, and the number of clock cycles per sample period will be an external constraint.

x is the value for current sample period.
x@1 is the value for one sample period back.
x@2 is the value for two sample periods back.
x@3 is the value for three sample periods back.

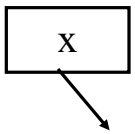
a0, a1,a2,a3 are the filter coefficients. Changing these coefficients change the filter function; assumed to be preloaded.

Dataflow Graph

We need a method of visualizing the data dependencies and operations to be performed. One method of doing this is the *dataflow* graph.

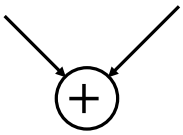


Operations in a Dataflow graph



An input operation. Inputs are assumed registered. An input operation requires 1 clock cycle.

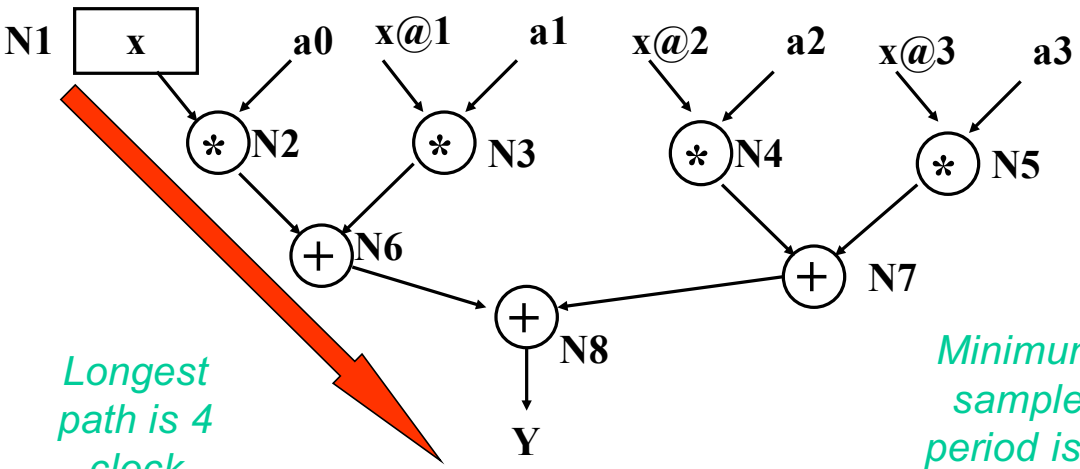
An output operation. Outputs are assumed to not be registered because they will be registered by the following datapath they produce output for.



An execution unit operation. Based on clock period constraints, execution units can be ***chained*** (a multiplier output directly feeding an adder input without an intervening register) or ***non-chained*** (all inputs/outputs of execution units are registered).

Minimum Required Clock Cycles

Assume that clock period constraint does not allow execution unit chaining (registers are between execution units). Minimum # of clock cycles will be longest path through the datapath.



Longest path is 4 clock cycles

Minimum sample period is 4 clocks.

Resource Estimation

Given a clock cycle constraint (sample period), can estimate minimum number of needed resources.

Assume the minimum sample period of 4 clocks.

Minimum resource estimation is:

operations/ # of clocks

Minimum Resource estimation:

multipliers = # multiplies/ # clocks = 4/4 = 1

adders = # additions/ #clocks = 3 /4 = 1

Minimum resource estimation is 1 multiplier, 1 adder. Register estimation is tougher. Need to store **x@1, x@2, x@3, a0, a1, a2, a3.** Need at least 7 registers.

Resource Scheduling

Scheduling is the mapping operations onto execution units. A scheduling table lists clock cycles versus resources. Register Scheduling is addressed later.

Cycle Start	Adder	Multiplier	IO
#1	idle	Reg??←x@3*a3 (N5)	Input X
#2	idle	Reg??←x@2*a2 (N4)	
#3	N7 op (N5+N4)	Reg?? ←x@1*a1 (N3)	
#4	idle	Reg?? ←x*a0 (N2)	

Scheduling Failed

The scheduling failed. Not possible to schedule the adder operations represented by nodes N6 and N8 in the 4 clock cycle budget.

The minimum resource estimation is a *lower bound*; it may not be possible to find a schedule to fit it.

If scheduling fails, there are two options:

a. Increase resources, keep same # of clocks
b. Increase # of clocks, keep same number of resources

For minimum sample period, determine which resource to add.

The bottleneck is the multiplier. Lets add another multiplier.

Resource Scheduling (2nd try)

Resource: Cycle Start	Adder	Mult A	Mult B	IO
#1	idle	x@3*a3 (N5)	x@2*a2 (N4)	Input X
#2	N7 op (N5+N4)	x@1*a1 (N3)	x*a0 (N2)	
#3	N6 op (N3+N2)	idle	idle	
#4	N8 op (N7+N6)	idle	idle	

Scheduling is Successful

Register Allocation

At this point, need to allocate registers to save temporary results. At beginning of operation, we know that we need to have the values **a0, a1, a2, a3, x@3, x@2, x@1** stored. So we need at least 7 registers.

The registers holding **a0-a3** will not change value during the computation, so we will not consider them in our scheduling.

Assume at Start: **RA = x@3, RB=x@2, RC=x@1**

Register Scheduling (Clock #1)

Regs: **RA = x@3, RB=x@2, RC=x@1**

Clock 1:

Input X??? Where to put this? For now, use new register RegD.

Input **x**: **RD ← x**
x@3*a3 (N5): **RA ← RA*a3** (don't need **x@3** after this, destroy **RA**)
x@2*a2 (N4): **?? ← RB*a2** (will need **x@2** next time, can't destroy **RB**)

Add another register

x@2*a2 (N4): **RE ← RB*a2** (will need **x@2** next time, can't destroy **RB**)

Scheduling this operations forced us to add two additional registers: **RD, RE**

Next, perform register scheduling for Clock #2

Register Scheduling (Clock #2)

Regs: **RA = N5, RB=x@2, RC=x@1, RD=x, RE=N4**

Clock 2:

N4 + N5 (N7): **RA ← RE+RA** (destroy RA, don't need N5 anymore)
x@1*a1 (N3): **?? ← RC*a1** (will need x@1 next time, can't destroy RC)

Look for a free register. Don't need **RE (N4)** after this clock cycle, use it.

x@1*a1 (N3): **RE ← RC*a1** (store result in **RE**)
x*a0 (N2): **?? ← RD*a0** (will need "**x**" next time, can't destroy **RD**)

Any free registers? NO. Add another register.

x*a0 (N2): **RF ← RD*a0**

Scheduling these operations forced us to add one more register: **RF**

Next, perform register scheduling for Clock #3

Register Scheduling (Clock #3, Clock #4)

Regs: **RA = N7, RB=x@2, RC=x@1, RD=x, RE=N3, RF=N2**

Clock 3:

N6 op (N3+N2): **RE ← RE + RF** (destroy **RE**, don't need **N3** anymore)

Regs: **RA = N7, RB=x@2, RC=x@1, RD=x, RE=N6, RF=N2**

Clock 4:

N8 op (N7+N6): **Y ← RA + RE** (output is unregistered)

Must consider initial conditions for next sample period:
RA = x@3, RB=x@2, RC=x@1

x@1 ← x	**RC ← RD**	Note that **x** in this sample period becomes **x@1**
x@2 ← x@1	**RB ← RC**	for the next sample period, **x@1** becomes **x@2**,
x@3 ← x@2	**RA ← RB**	etc...

Final Datapath Requirements

- For sample period = 4 clocks:
  - 2 Multipliers, 1 adder
  - 10 registers (**RA-RF**, plus 4 registers for **a0,a1,a2,a3**)
- Is this the best hardware allocation?
  - Maybe not, if we try harder may be able to reduce the number of registers
- Lets go with this and develop the datapath diagram

Datapath Unit Sources & Destinations

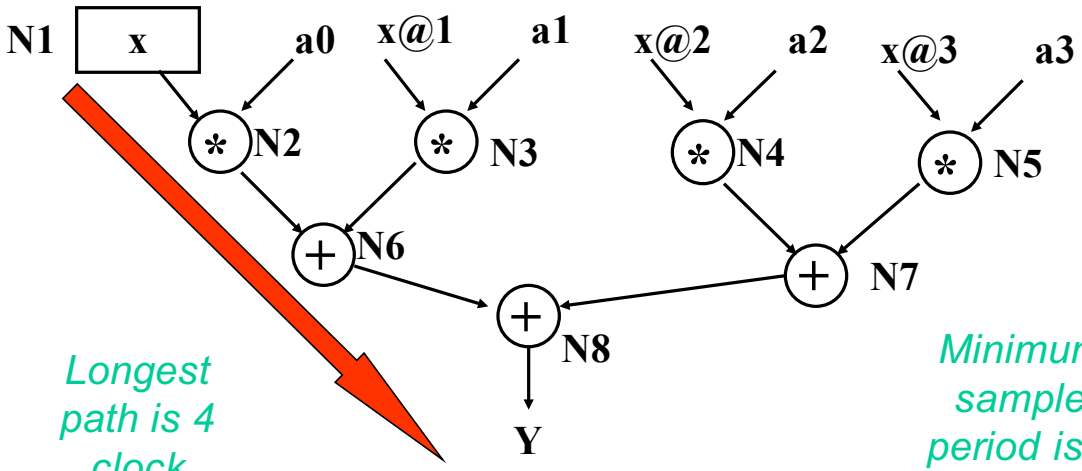
Mult A: Left sources: **RA, RC** Right sources: **a3, a1**
Mult B: Left sources: **RB, RD** Right sources: **a2, a0**
Adder: Left sources: **RE, RA** Right sources: **RA, RF, RE**

RA src: **MultA, Adder, RB**
RB src: **RC**
RC src: **RD**
RD src: **X**
RE src: **Adder, Mult A, Mult B**
RF src: **Multiplier B**

a0-a3 registers loaded from external databus **X**

Minimum Required Clock Cycles

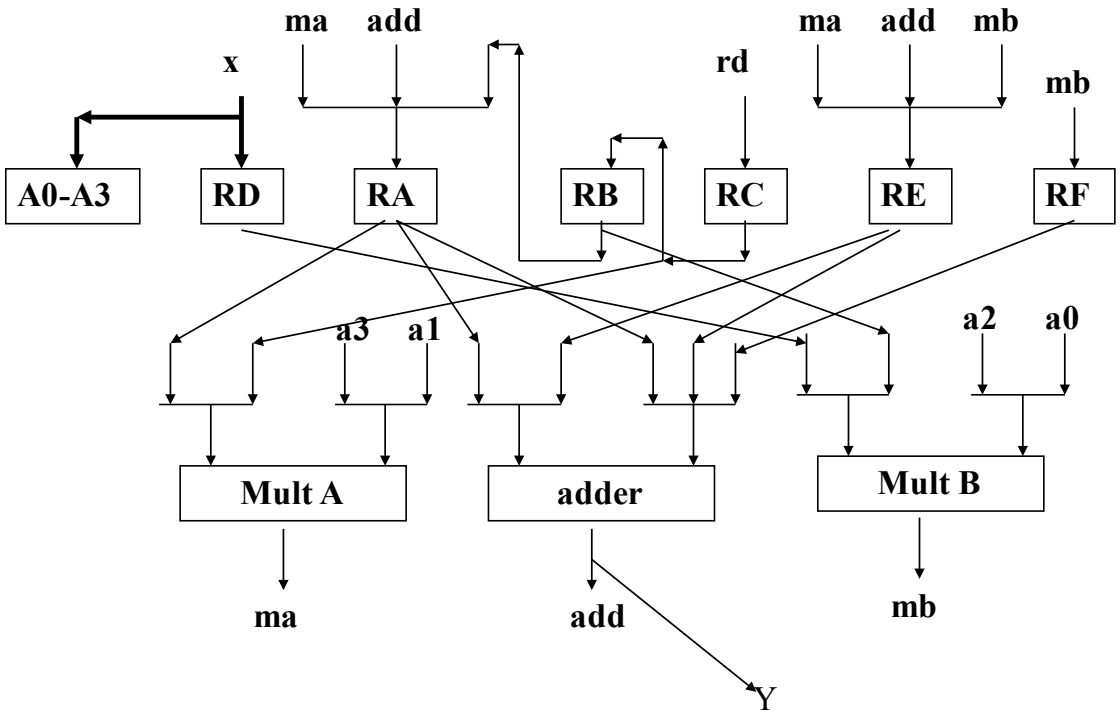
Assume that clock period constraint does not allow execution unit chaining (registers are between execution units). Minimum # of clock cycles will be longest path through the datapath.



Longest path is 4 clock cycles

Minimum sample period is 4 clocks.

Datapath



Comments

- Saving on Execution units can lead to lots of wiring (in FPGA routing delay) and muxes because of the amount of execution unit sharing that is required
- Could probably have reduced some of the mux requirements by more careful assignment of temporary values to registers
- This datapath would require a controller with four states; each state corresponding to a clock cycle.
  - Output of FSM would be mux select lines, register load lines
  - May need extra states if handshaking control (input_rdy, output_rdy) is required.

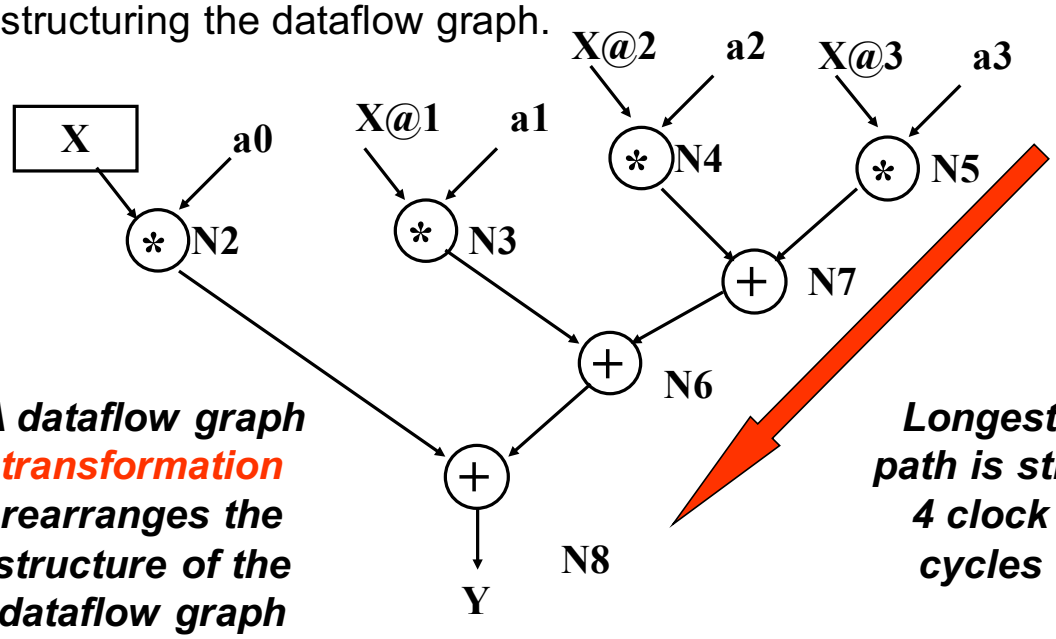
Reschedule with an Extra Clock Cycle

Lets increase sample period from 4 to 5 to try to reduce the number of required resources in the datapath

Resource: Cycle Start	Adder	Multiplier	IO
#1	idle	Reg??←x@3*a3 (N5)	Input X
#2	idle	Reg??←x@2*a2 (N4)	
#3	N7 op (N5+N4)	Reg?? ←x@1*a1 (N3)	
#4	idle	Reg?? ←x*a0 (N2)	
#5	N6 op (N2 + N3)	idle	

Scheduling Still Failed

Did not schedule Node 8 (N8). There should be a way in which we can make better use of the adder. Try restructuring the dataflow graph.



A dataflow graph transformation rearranges the structure of the dataflow graph

Longest path is still 4 clock cycles

Try again with Sample Period = 5

Resource: Cycle Start	Adder	Multiplier	IO
#1	idle	Reg??←x@3*a3 (N5)	Input X
#2	idle	Reg??←x@2*a2 (N4)	
#3	N7 op (N5+N4)	Reg?? ←x@1*a1 (N3)	
#4	N6 op (N3+N7)	Reg?? ←x*a0 (N2)	
#5	N8 op (N2 + N6)	idle	

Scheduling succeeds with new dataflow graph

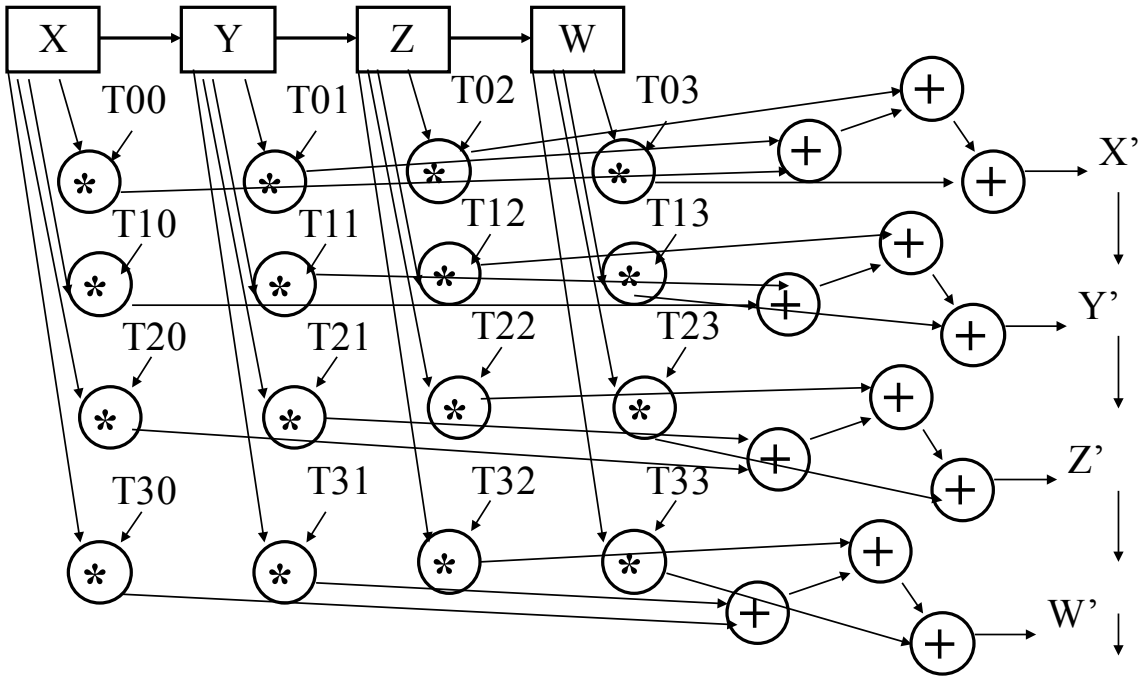
Flowgraph for Matrix Multiply

$$\begin{pmatrix} T00 & T01 & T02 & T03 \\ T10 & T11 & T12 & T13 \\ T20 & T21 & T22 & T23 \\ T30 & T31 & T32 & T33 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} = \begin{pmatrix} X' \\ Y' \\ Z' \\ W' \end{pmatrix}$$

$$X' = X*T00 + Y*T01 + Z*T02 + W*T03$$
$$Y' = X*T10 + Y*T11 + Z*T12 + W*T13$$
$$Z' = X*T20 + Y*T21 + Z*T22 + W*T23$$
$$W' = X*T30 + Y*T31 + Z*T32 + W*T33$$

IO Constraint: Single input bus, single output bus

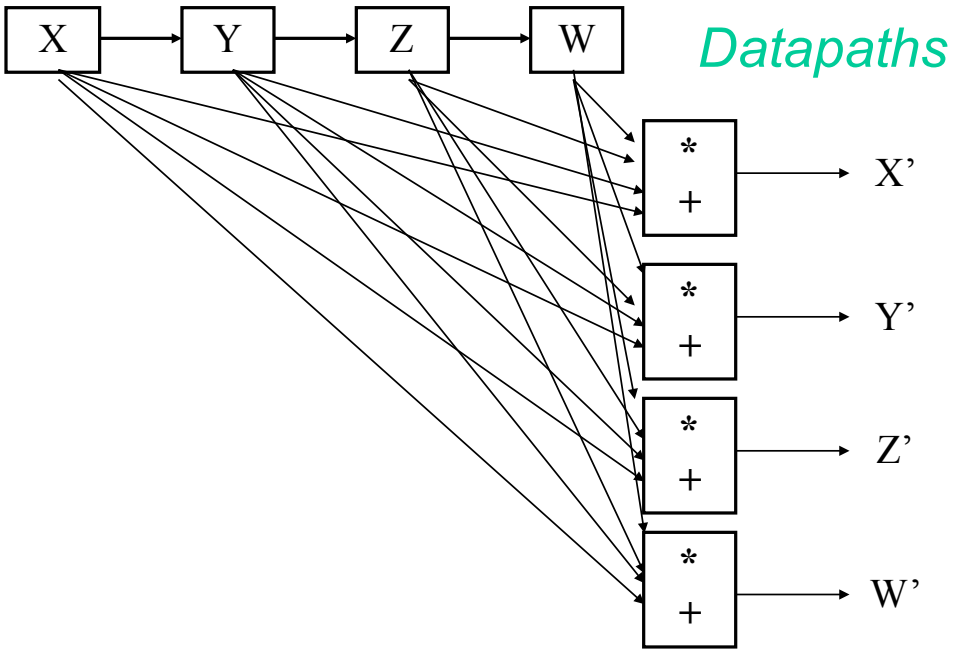
Flowgraph for Matrix Multiply (cont)



Comments on MM Flowgraph

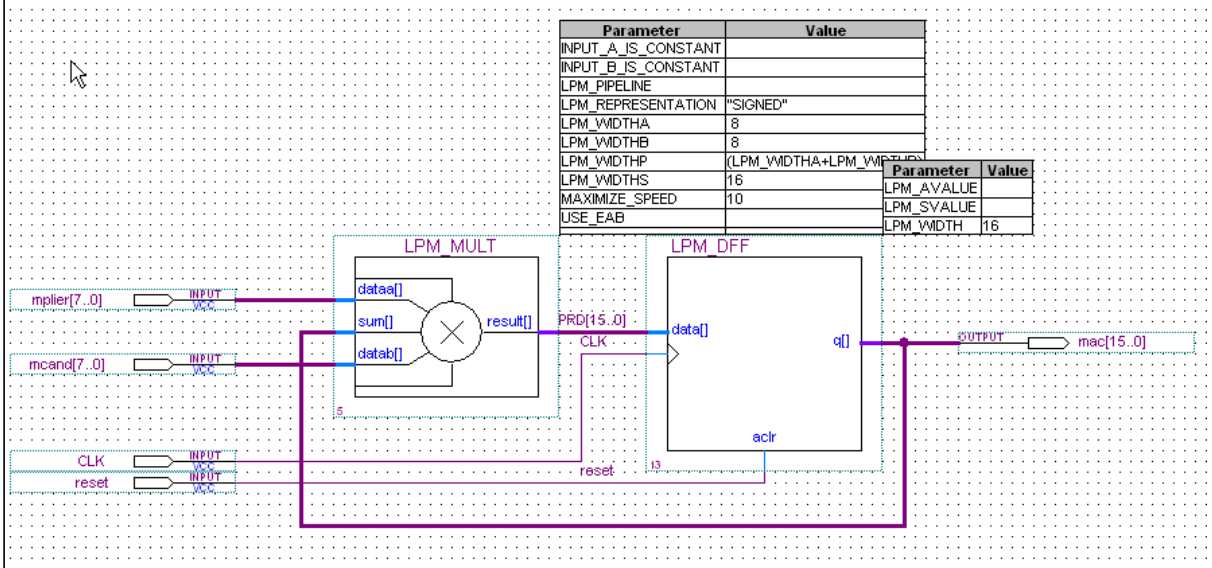
- The main thing to notice about the graph is that you don't have to wait until you have X,Y,Z,W before you begin operations
  - Once you have X, you can do four multiply operations
- Another thing to note is the symmetry and parallelism available
  - You could have four parallel datapaths, each one containing a multiplier and an adder, and produce X', Y', Z', W' from these four datapaths

Parallel Datapaths for MM



Multiply/Accumulate Unit (MAC)

(in QuartusII)



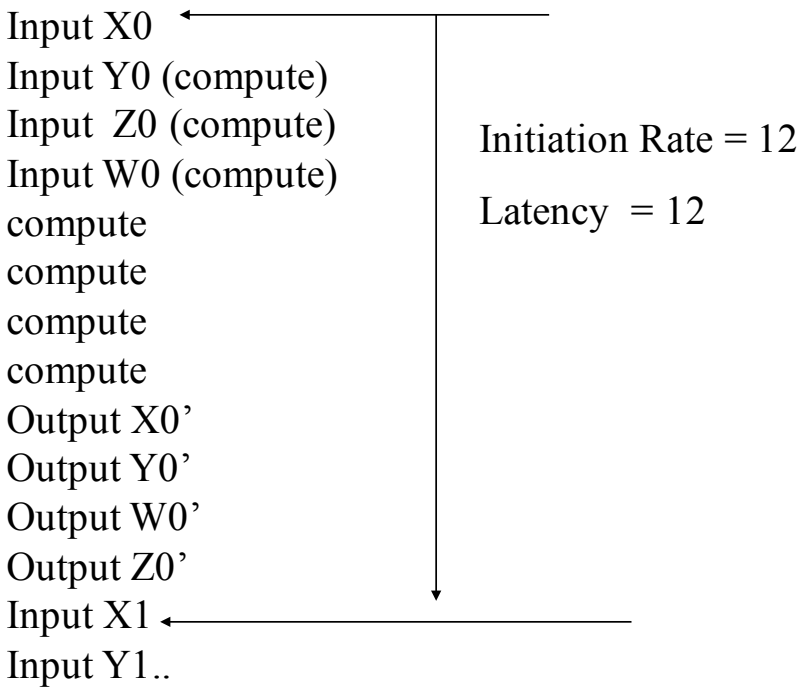
Latency and Initiation Rate

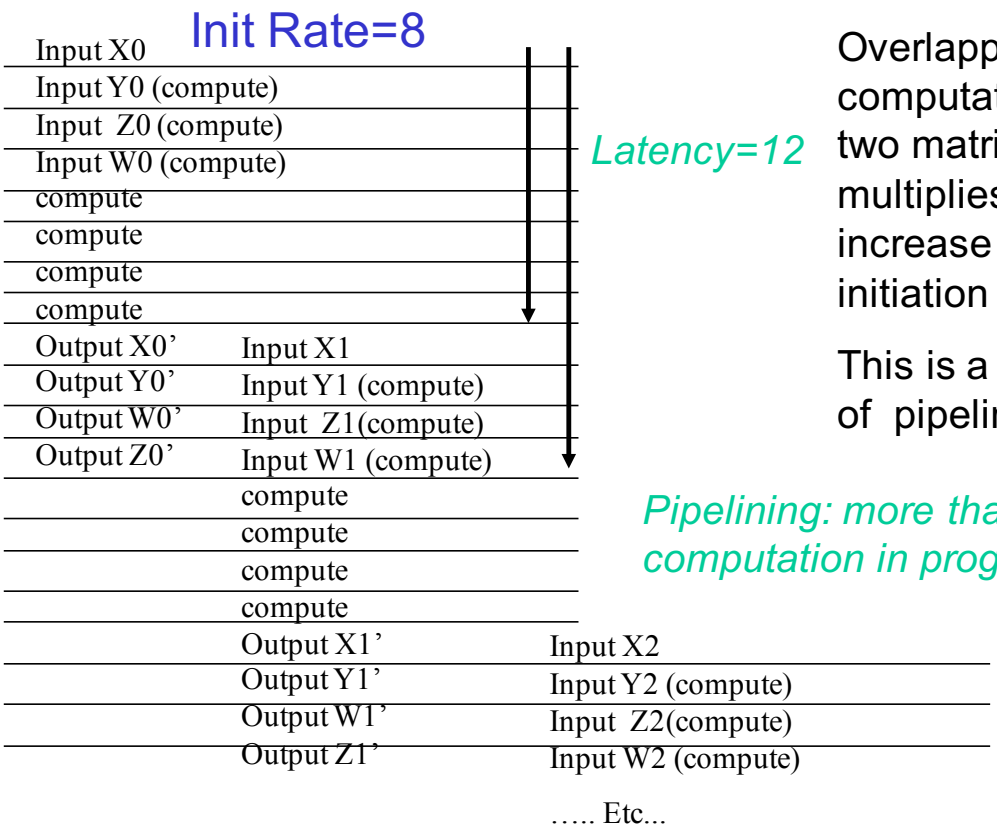
Initiation Rate: Maximum rate at which new values are may be input to the circuit

Latency: Number of clocks from input value to COMPLETED output value

For the project, initiation rate will be number of clocks from inputting **A** for one set of $(a_{11}, a_{21}, a_{31}, \ldots)$ to inputting the next **A** for a new set of $(a_{11}, a_{21}, a_{31}, \ldots)$

MM Initiation Rate and Latency



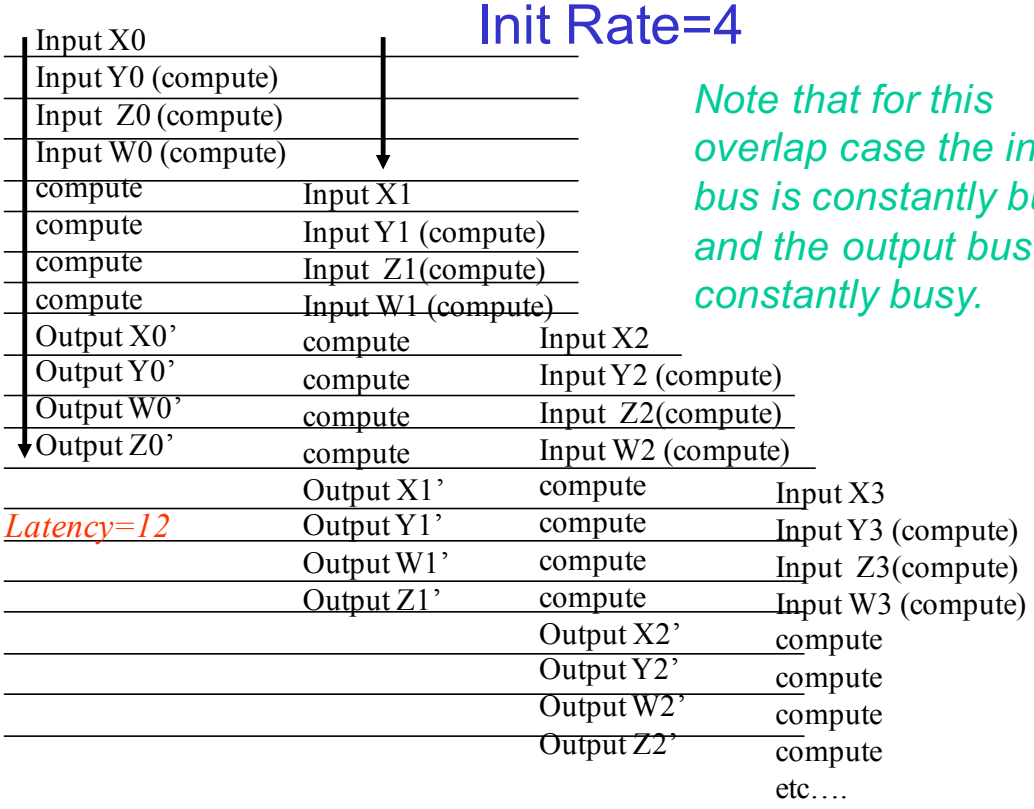


Overlapping computation of two matrix multiplies to increase initiation rate.

This is a form of pipelining!!!

Pipelining: more than one computation in progress.

Init Rate=4



Note that for this overlap case the input bus is constantly busy, and the output bus is constantly busy.

3 "Types" of Pipelining

- Bit Level
  - **Individual Building Blocks (eg. multipliers, etc.)**
- Non-chained Datapaths
  - **Pipelining Between Execution Units (Building Blocks)**
- System Level
  - **Overlapping Computations in the Resource Schedule**