How to Design Complex Digital Systems

Course "Mantras"

1. One clock, one edge, flip-flops only
2. Design BEFORE coding
3. Behavior implies function
4. Clearly separate control and datapath

Steps in High Level Design

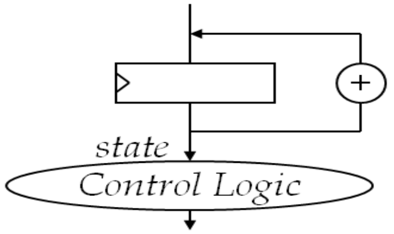
- Determine MicroOperations to be performed on datapath units
  - e.g. adds, subtracts, multiplies, memory references, etc.
- Design datapath units to perform these operations efficiently
  - Design to RTL level
  - Note later sections on efficiency
- Identify control points
  - Control lines
  - Status lines
- Determine reset/start/stop/transition actions
  - Especially global reset strategy

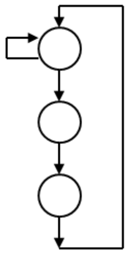
Steps in High Level Design (cont'd)

- Determine control sequence
  - Generally MicroOp sequence required to perform overall task
  - Gives sequence of control events and status line responses
- Determine control strategy
  - Mix of FSMs and/or counters
- Verify before coding

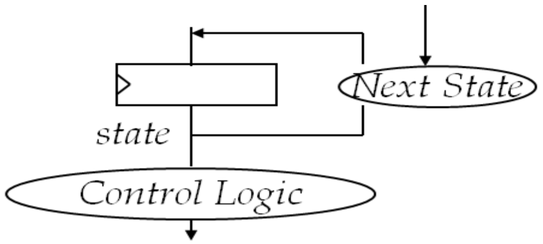
Control Strategies

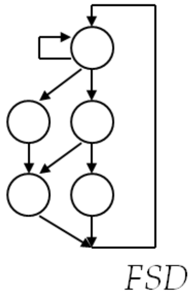
- Counters:
  - Takes machine through a linear sequence of states with few decisions along the way





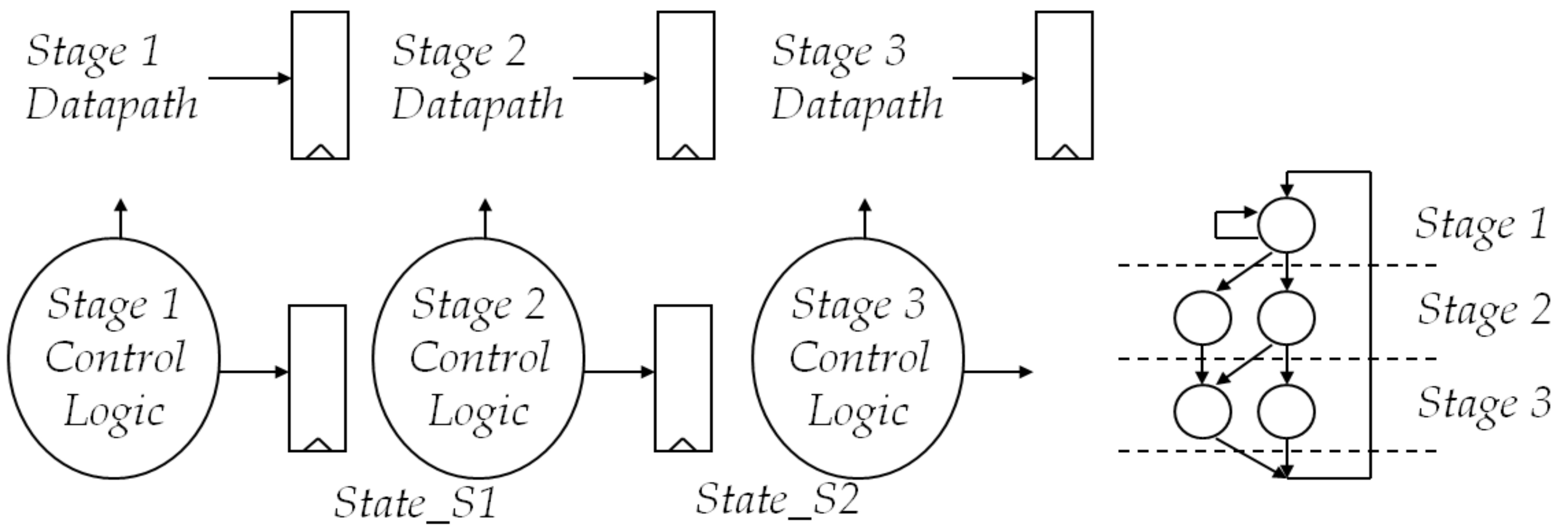
- FSMs
  - Permits branches in control decision chain





… Control Strategies

- Pipeline Control
  - In a sense, an "unrolled" FSM – each stage does one step (or one of several parallel steps) in an FSM; state information communicated between stages



Reset

- Reset is a global signal that the designer can not modify
- It is generally asserted on power up or a "hard" reset
- It is used to start the machine in a "known" state
- Thus it must be distributed to
  - All FSMs
  - Selected counters
  - Selected status registers

Achieving Efficiency

- High level tradeoffs:
  - Parallelism
  - Pipelining
  - Optimizing the critical resource
    - E.g. Memory bandwidth
  - Keep resources busy
    - If a resource is idle can it be shared?
    - Goal : Everything is used every clock cycle
- Algorithmic Optimizations
  - E.g. Algorithms that avoid DRAM accesses
    - e.g. Compress table onto SRAM
  - Exploiting common algorithms in Computer Science
    - e.g. Boyer-Moore for string matching
    - e.g. Hash tables for matches
    - e.g. Shift instead of *2 /2

Mid-level efficiency

- Think hardware (area and delay):
  - Avoid large FSMs
  - Count the large units (* + memories, etc.)
  - Avoid high-fanout signals
  - Avoid priority logic
  - Structure arithmetic for speed
    - E.g. CLA instead of ripple carry
- Exploit existing Intellectual Property

Design Ware

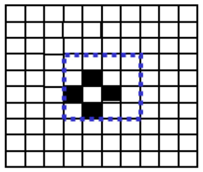
- Synopsys, and others, provide libraries of carefully optimized design blocks for you to use -- called `Design Ware'
  - Libraries include: Arithmetic, Advanced Math, DSP, Control, Sequential, and Fault Tolerant
  - For +,-,*, >=, <=, >, and <, design ware is automatically used
  - More complex cells must be inferred via a procedure call. e.g. cosine:

Design Ware

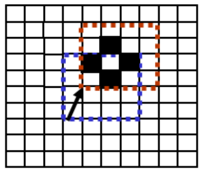
```
module trigger (angle, cos_out);
parameter wordlength1 = 8, wordlength2 = 8;
input [wordlength-1:0] angle;
output [wordlength-1:0] cos_out;
// passes the widths to the cos function
parameter angle_width = wordlength1, cos_width = wordlength2;
‘include
“/afs/bp/dist/synopsys_syn/dw/sim_ver/DW02_cos_function.inc”
wire [wordlength2-1:0] cos_out;
// infer DW02_cos
assign cos_out = cos(angle);
Endmodule
```

Example

- Motion Estimator
- Task:
- Detect blocks of video data in successive frames that are related only via a translation
  - Digital Video is captured as blocks of 16x16 pixels
  - Want to determine if block has moved largely unchanged
    - If true can transmit motion vector rather than block
    - Permits high level of compression
  - Example (4x4 block)



Reference Block in Frame 1



"Draw block" with motion vector (1,2) in frame 2

Search Algorithm

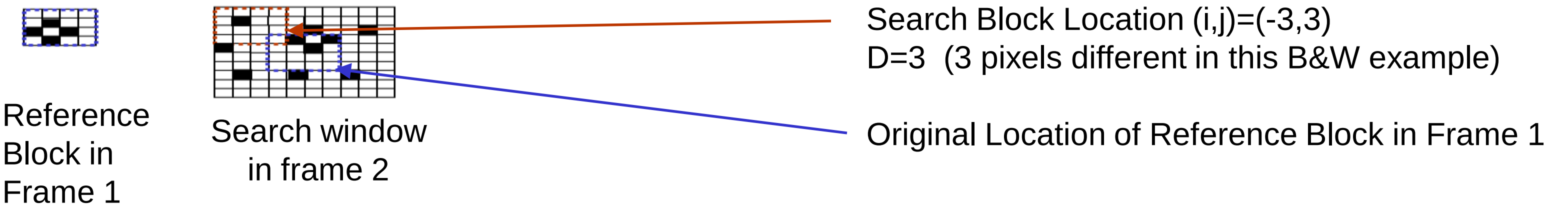
Describe for 16x16 reference block:

1. Move a window the size of the reference block over search space in the second frame
2. For each window location (i,j) determine the distortion vector

$$D(i,j) = \sum_{m=0}^{15} \sum_{n=0}^{15} | r_{m,n} - S_{m+i,N+j} |$$

3. Maintain the best distortion and appropriate motion vector produced so far.

- For Example (4x4 block):


Search Block Location (i,j)=(-3,3)
D=3 (3 pixels different in this B&W example)
Original Location of Reference Block in Frame 1
Reference Block in Frame 1
Search window in frame 2

System Requirements

- System Requirements:
- 16x16 Reference Block
- 31x31 Search Window
- Each stored in one two-read-ported memory
  - In reality one memory per frame
- Grey-scale coded pixels (8 bits/block)
- 4096 reference blocks in a frame
- Conduct search at 15 frames per second
  - (Encoding does not have to be real time)
- Clocks available : 130, 260 MHz
- 0.25 μm CMOS library

Step 1 : System Design

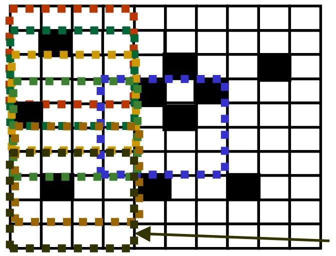
- Elements to thinking:
- Bottom-up design
  - Determine critical bottlenecks (paths & other bottlenecks)
- Top-down design
  - Determine use of pipelining and parallelism to meet performance constraints

$$D = D + | r_{mn} - S_{m+i,N+j} |$$

- <u>Critical Bottlenecks:</u>
- Elemental Arithmetic Operation (add-accumulate):
  - Design, synthesize ➔ Can operate at 260 MHz with some timing margin left over
- Memories:
  - Single access per clock cycle

… System Design

- Top Down Design
- Number of add-accumulates per clock cycle:
  - 4096 blocks per 1/15 of a second
  - (31-15)x(31-15)=256 searches/block
  - 16x16=256 add-accumulates per search
  - ➔ 4096*15*256*256 = 4.027E9 add-accumulates/second
  - At 260 MHz ➔ At least 16 adders in parallel (4027/260=15.5)
- Searches/block [(4x4) on (10x10) example]:

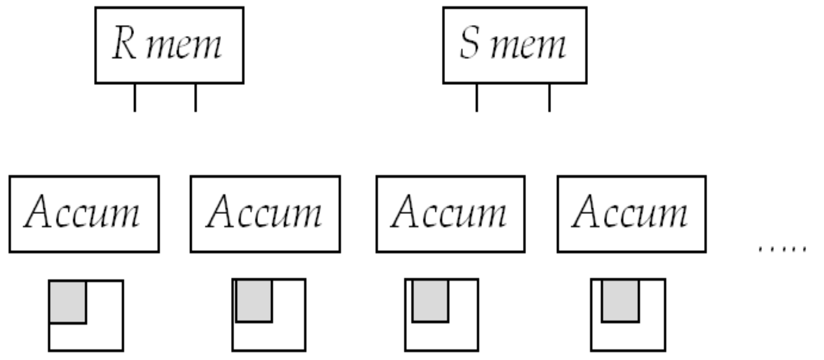


Search (-3,-3)
Search (-2,-3)
Search (-1,-3)
Search (0,-3)
Search(1,-3)
Search(2,-3)
Search(3,-3)

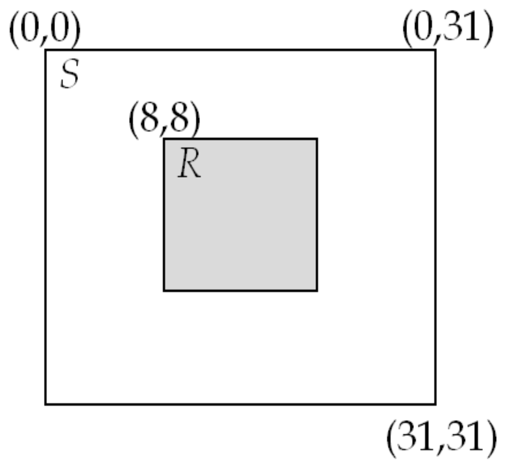
7 searches per column
7 searches per row
(10-4)x(10-4) total searches

…System Design

- First Attempt
- Assign one search per Accumulator

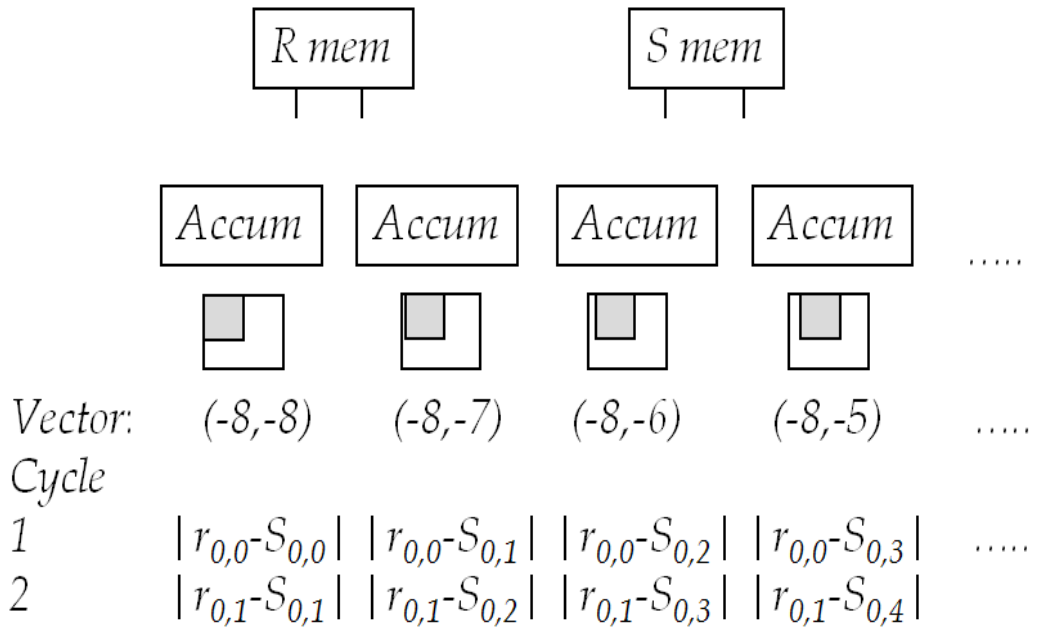


Vector:	(-8,-8)	(-8,-7)	(-8,-6)	(-8,-5)	.....
Cycle					
1	$\lvert r_{0,0}-s_{0,0}\rvert$	$\lvert r_{0,0}-s_{0,1}\rvert$	$\lvert r_{0,0}-s_{0,2}\rvert$	$\lvert r_{0,0}-s_{0,3}\rvert$	.....
2	$\lvert r_{0,1}-s_{0,1}\rvert$	$\lvert r_{0,1}-s_{0,2}\rvert$	$\lvert r_{0,1}-s_{0,3}\rvert$	$\lvert r_{0,1}-s_{0,4}\rvert$	



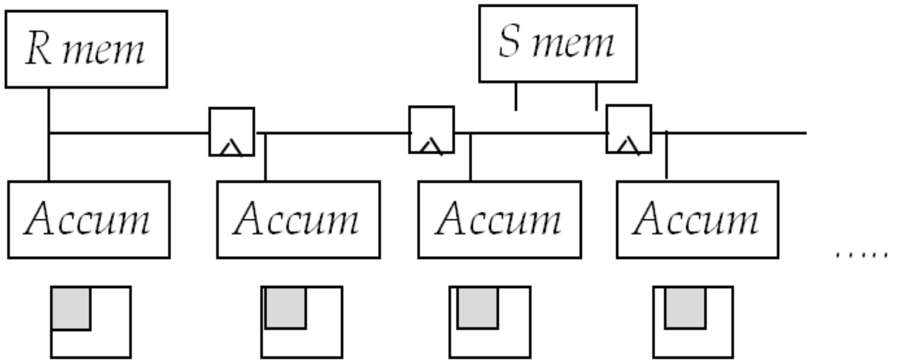
… System Design

- Second Attempt:
- Stagger Startup of Accumulators


R mem
S mem
Accum
Accum
Accum
Accum
.....
Vector: (-8,-8) (-8,-7) (-8,-6) (-8,-5) .....
Cycle
1 $|r_{0,0}-S_{0,0}|$ $|r_{0,0}-S_{0,1}|$ $|r_{0,0}-S_{0,2}|$ $|r_{0,0}-S_{0,3}|$ .....
2 $|r_{0,1}-S_{0,1}|$ $|r_{0,1}-S_{0,2}|$ $|r_{0,1}-S_{0,3}|$ $|r_{0,1}-S_{0,4}|$

… System Design

- Final Solution:
- Pipeline R



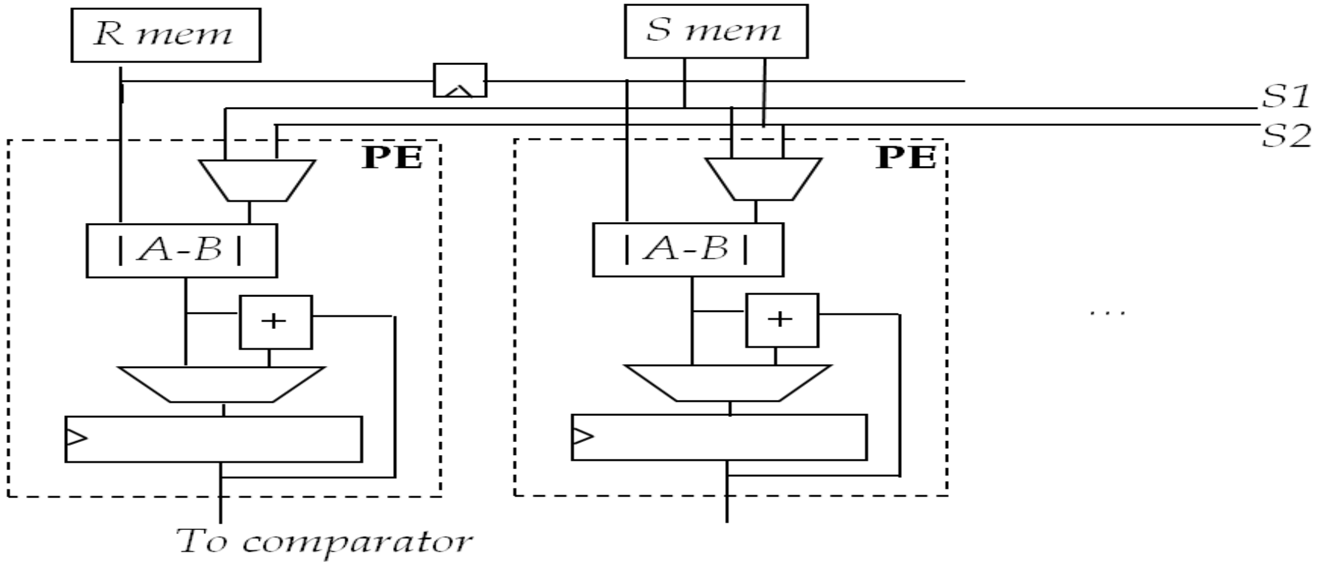
Vector:	(-8,-8)	(-8,-7)	(-8,-6)	(-8,-5)	.....
Cycle					
1	$\lvert r_{0,0}-S_{0,0}\rvert$				.....
2	$\lvert r_{0,1}-S_{0,1}\rvert$	$\lvert r_{0,0}-S_{0,1}\rvert$			.....
3	$\lvert r_{0,2}-S_{0,2}\rvert$	$\lvert r_{0,1}-S_{0,2}\rvert$	$\lvert r_{0,0}-S_{0,2}\rvert$		.....
4	$\lvert r_{0,3}-S_{0,3}\rvert$	$\lvert r_{0,2}-S_{0,3}\rvert$	$\lvert r_{0,1}-S_{0,3}\rvert$	$\lvert r_{0,0}-S_{0,3}\rvert$	.....
...					
15	$\lvert r_{0,15}-S_{0,15}\rvert$	$\lvert r_{0,14}-S_{0,15}\rvert$	$\lvert r_{0,13}-S_{0,15}\rvert$	$\lvert r_{0,12}-S_{0,15}\rvert$	
16	$\lvert r_{1,1}-S_{1,1}\rvert$	$\lvert r_{0,15}-S_{0,16}\rvert$	$\lvert r_{0,14}-S_{0,16}\rvert$	$\lvert r_{0,13}-S_{0,16}\rvert$	

2 S mem ports required $\longrightarrow$ 15

Step 2 : Design Datapath

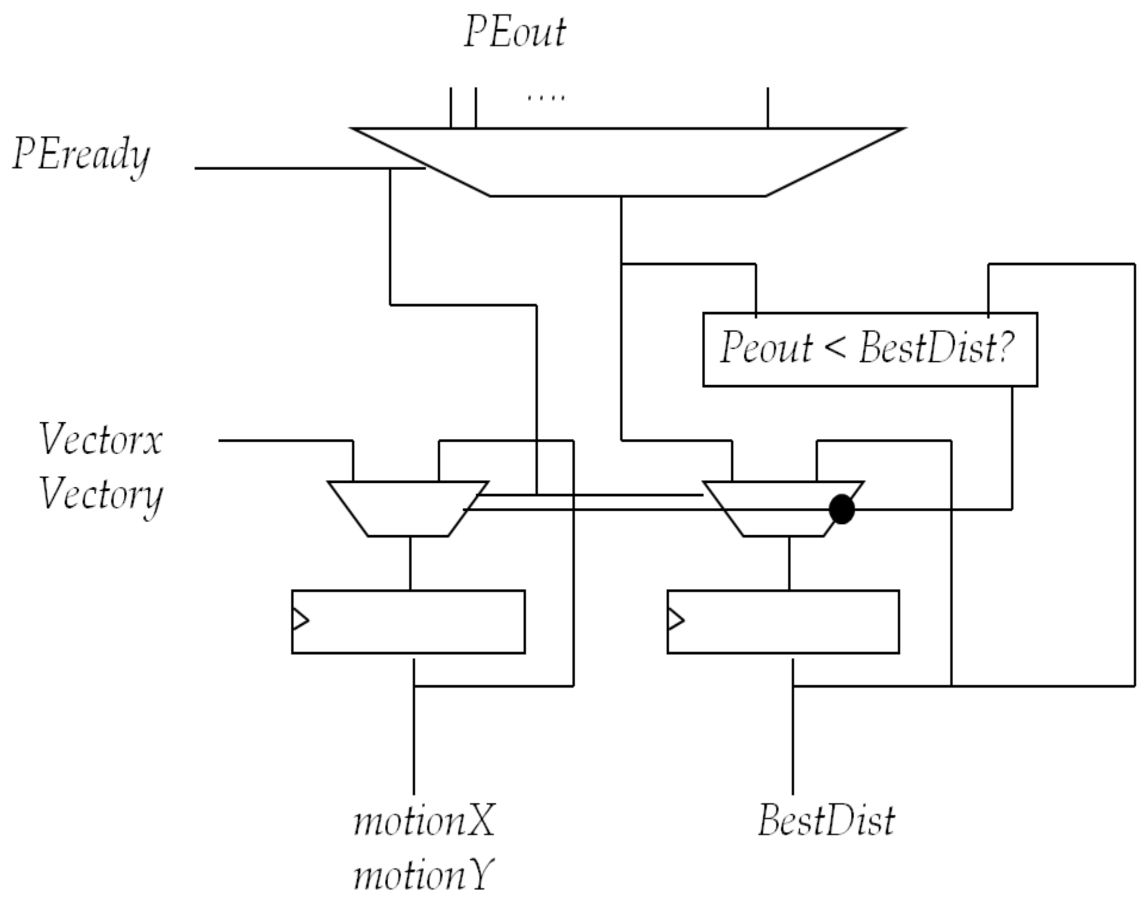
- Datapath Details:
- Detailed hardware required to implement above

PE = Processing Element



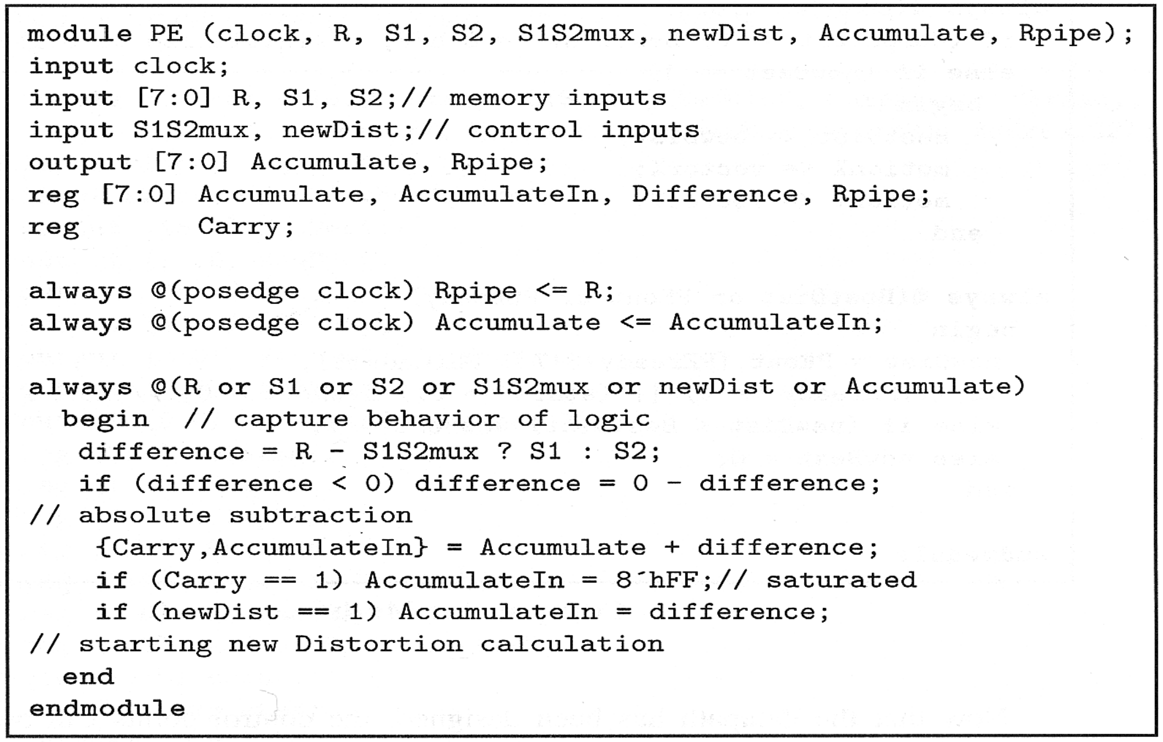
… Datapath

- Comparator:



Coding Datapath

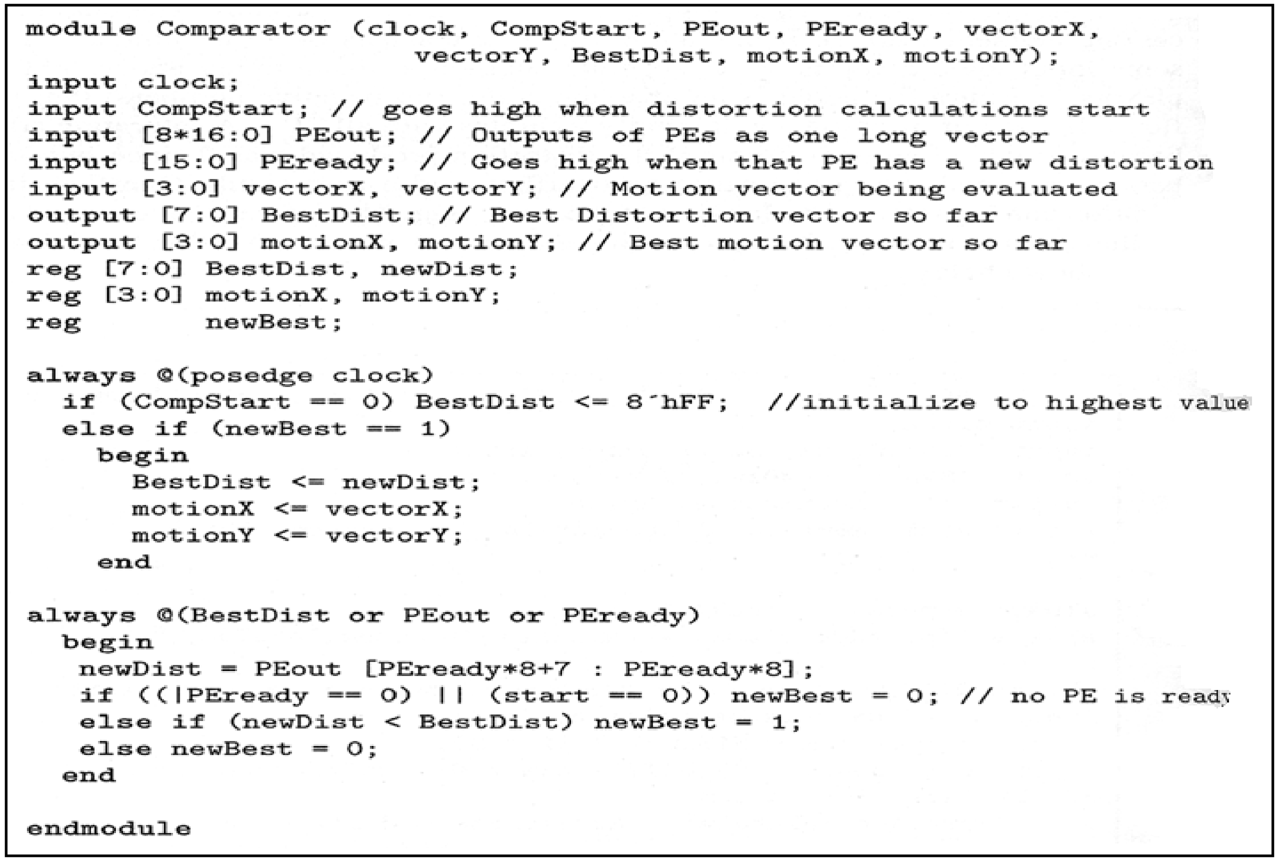
- PE: Note, accumulator cant overflow – saturate at FF



```
module PE (clock, R, S1, S2, S1S2mux, newDist, Accumulate, Rpipe);
input clock;
input [7:0] R, S1, S2;// memory inputs
input S1S2mux, newDist;// control inputs
output [7:0] Accumulate, Rpipe;
reg [7:0] Accumulate, AccumulateIn, Difference, Rpipe;
reg       Carry;

always @(posedge clock) Rpipe <= R;
always @(posedge clock) Accumulate <= AccumulateIn;

always @(R or S1 or S2 or S1S2mux or newDist or Accumulate)
  begin  // capture behavior of logic
   difference = R - S1S2mux ? S1 : S2;
   if (difference < 0) difference = 0 - difference;
// absolute subtraction
    {Carry,AccumulateIn} = Accumulate + difference;
    if (Carry == 1) AccumulateIn = 8´hFF;// saturated
    if (newDist == 1) AccumulateIn = difference;
// starting new Distortion calculation
  end
endmodule
```

Motion Estimator Processing Element (PE).

Datapath



```
module Comparator (clock, CompStart, PEout, PEready, vectorX,
                      vectorY, BestDist, motionX, motionY);
input clock;
input CompStart; // goes high when distortion calculations start
input [8*16:0] PEout; // Outputs of PEs as one long vector
input [15:0] PEready; // Goes high when that PE has a new distortion
input [3:0] vectorX, vectorY; // Motion vector being evaluated
output [7:0] BestDist; // Best Distortion vector so far
output [3:0] motionX, motionY; // Best motion vector so far
reg [7:0] BestDist, newDist;
reg [3:0] motionX, motionY;
reg       newBest;

always @(posedge clock)
  if (CompStart == 0) BestDist <= 8´hFF;  //initialize to highest value
  else if (newBest == 1)
    begin
      BestDist <= newDist;
      motionX <= vectorX;
      motionY <= vectorY;
    end

always @(BestDist or PEout or PEready)
  begin
   newDist = PEout [PEready*8+7 : PEready*8];
   if ((|PEready == 0) || (start == 0)) newBest = 0; // no PE is ready
   else if (newDist < BestDist) newBest = 1;
   else newBest = 0;
  end

endmodule
```

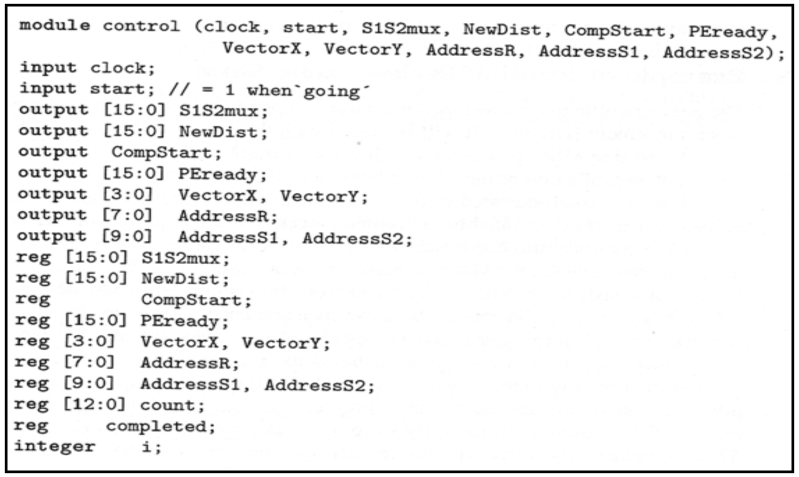
Comparator Module.

Step 3. Identify Control Points

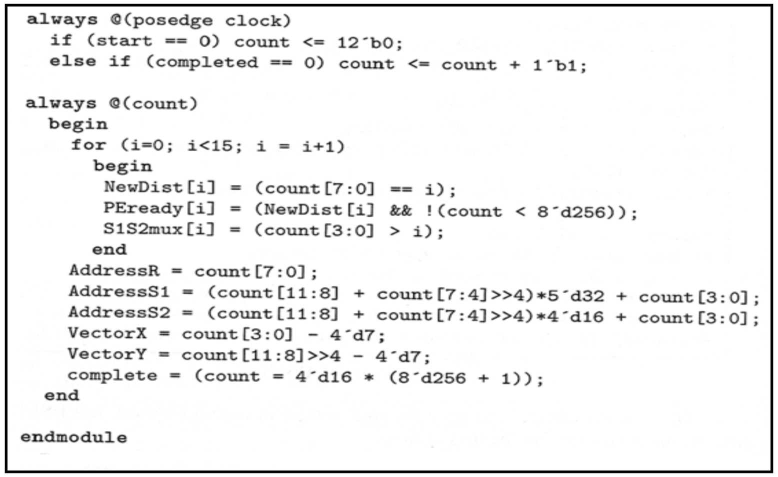
- PE control lines:
- S1S2mux [15:0]; // S1-S2 mux control
- NewDist [15:0] ; // =1 when PE is starting a new distortion calculation
- Comparator control lines:
- CompStart;11 = // when PEs running
- PEready [15:0]; // PEready[I]=1 when PEi has a new distortion vector
- VectorX [3:0] ;
- VectorY [3:0];// Motion vector being evaluated
- Memory control lines:
- Memories organized in row-major format
  - e.g. R(3,2) is stored at location 3*15+2- 1 = 46
- AddressR [7:0]; // address for Reference memory (0,0). ..(15,15)
- AddressS1 [9:0] ; // address for first read port of Search mem
- AddressS2 [9:0] ; // second read port of Search mem (0,0)-(30,30)

Step. 4 Design Controller

- Best Strategy : Counter



```
module control (clock, start, S1S2mux, NewDist, CompStart, PEready,
                VectorX, VectorY, AddressR, AddressS1, AddressS2);
input clock;
input start; // = 1 when`going´
output [15:0] S1S2mux;
output [15:0] NewDist;
output  CompStart;
output [15:0] PEready;
output [3:0]  VectorX, VectorY;
output [7:0]  AddressR;
output [9:0]  AddressS1, AddressS2;
reg [15:0] S1S2mux;
reg [15:0] NewDist;
reg        CompStart;
reg [15:0] PEready;
reg [3:0]  VectorX, VectorY;
reg [7:0]  AddressR;
reg [9:0]  AddressS1, AddressS2;
reg [12:0] count;
reg     completed;
integer    i;
```

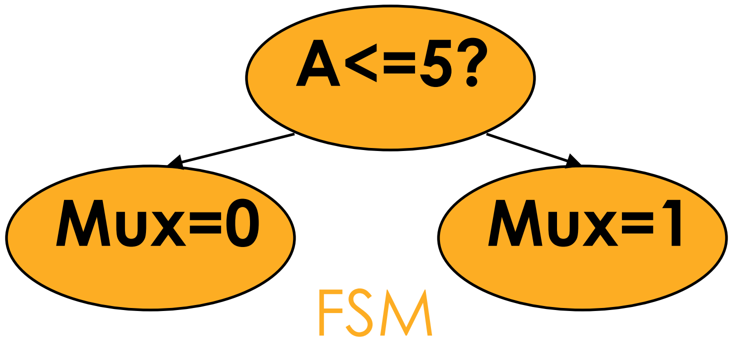


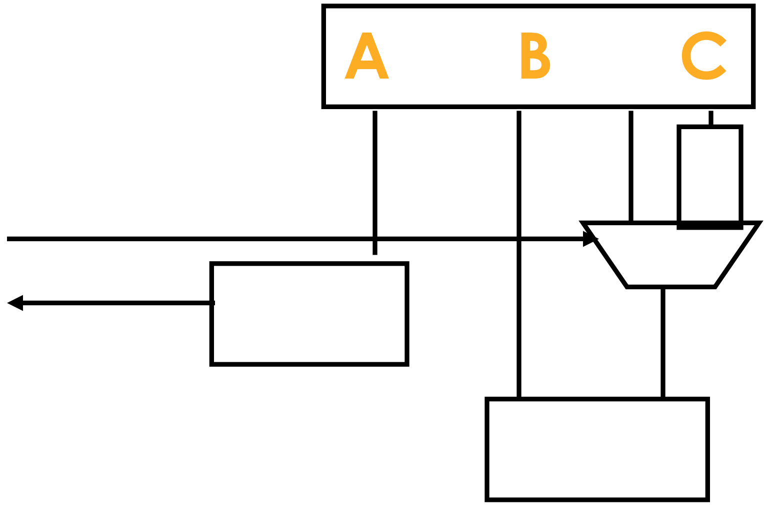
```
always @(posedge clock)
  if (start == 0) count <= 12´b0;
  else if (completed == 0) count <= count + 1´b1;

always @(count)
  begin
    for (i=0; i<15; i = i+1)
      begin
       NewDist[i] = (count[7:0] == i);
       PEready[i] = (NewDist[i] && !(count < 8´d256));
       S1S2mux[i] = (count[3:0] > i);
      end
    AddressR = count[7:0];
    AddressS1 = (count[11:8] + count[7:4]>>4)*5´d32 + count[3:0];
    AddressS2 = (count[11:8] + count[7:4]>>4)*4´d16 + count[3:0];
    VectorX = count[3:0] - 4´d7;
    VectorY = count[11:8]>>4 - 4´d7;
    complete = (count = 4´d16 * (8´d256 + 1));
  end

endmodule
```

- Reset Strategy
- Reset needed to initialize entire chip in known state
  - Does not apply here, as long as “start” comes from a unit that does use a reset

C to Verilog

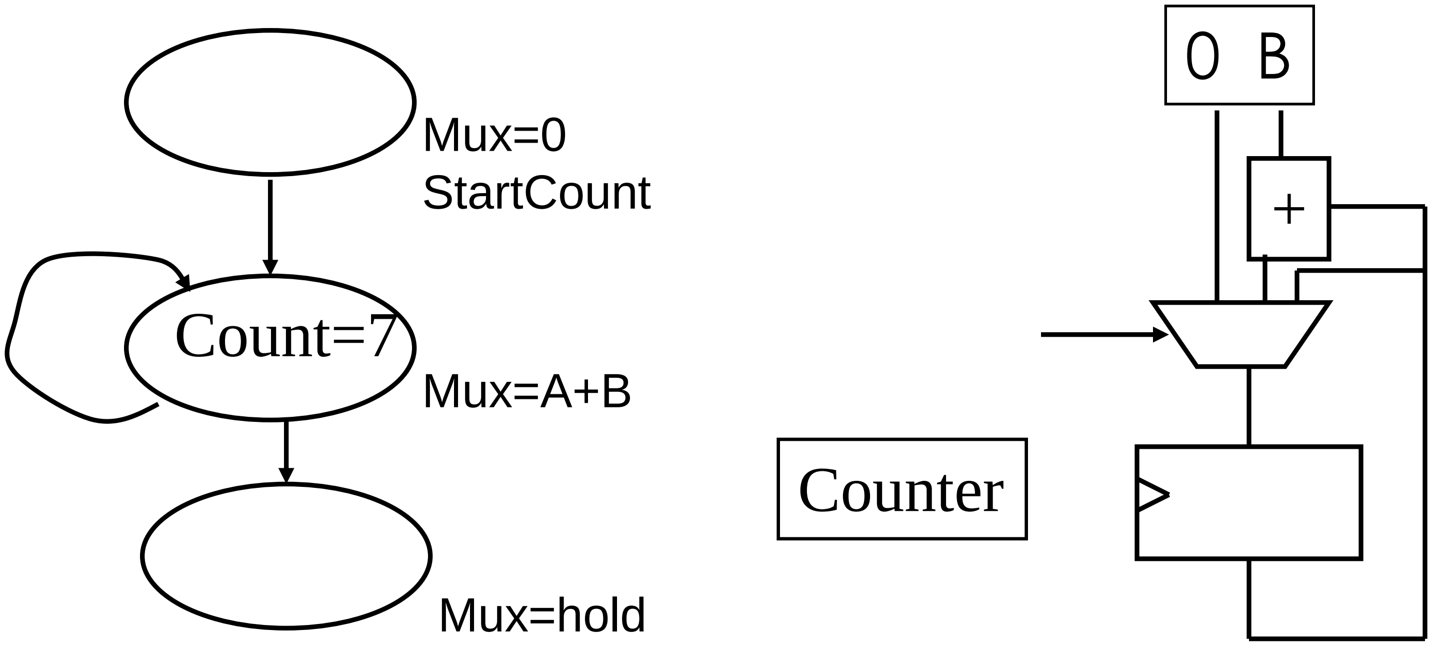
- Generally the "flow" constructs in C correlate to controller designs in Verilog, e.g.
- In "C": If (A<=5) {B=A+C;} else {B=A-C;}
- In Hardware:





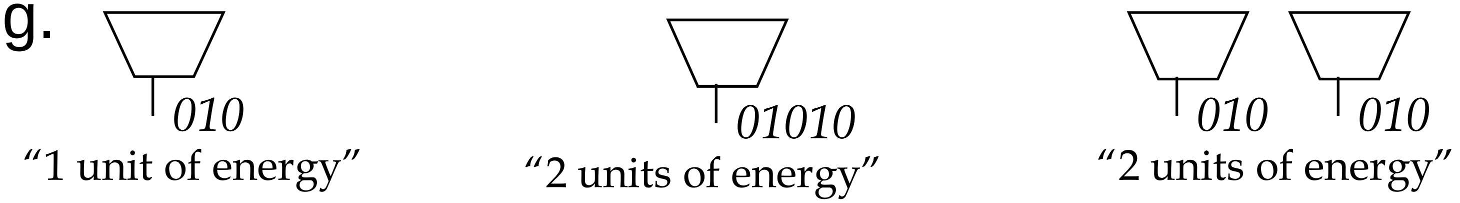
C to Verilog (cont'd)

- For loop:
  - In "C": B=0; for (i=0;i<=7;i++) B=B+A;
  - In Hardware:



Minimizing Power Consumption

- Will go over in a later set of notes, but here is the logic design impact…
  - In general, at the logic level, the energy required to complete a complex task is roughly proportional to:
    - Σ_{nodes} 01 and 10 logic transitions
  - E.g.

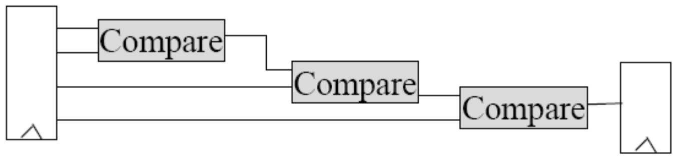


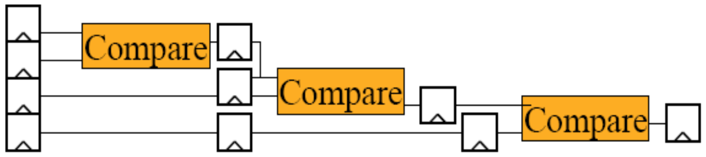
- Note:
  - Complex logical units (e.g. Multiplier) have a lot more internal nodes than simpler logical units
    - And thus consume more energy per operation

How to Minimize Power Consumption

- Simpler, smaller design will often also more energy efficient
- There is often a speed-power tradeoff

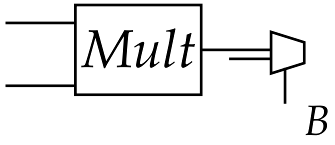
  E.g. Which design is more energy efficient?

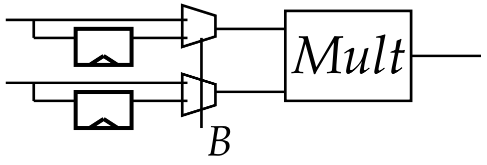




- Try to eliminate useless toggling

  E.g. Which design is LESS energy efficient if B mostly DESELECTS mult output?





… How to minimize power consumption

- Memory accesses are particularly energy hungry, especially with larger memories
- Complex data motion is particularly power hungry
  - E.g. Long range on-chip interconnect
  - Off-chip interconnect
  - Using an on-chip network to move data, especially a store and forward network

Review Questions

- What was the critical resource optimized in the Motion Estimator?
- How was the use of this resource optimized?
- What is the control strategy?

Review Questions

- What are my four “mantras”
- What are common control strategies?
- What are common speed-up strategies?
- What is “reset” for?

Review Questions

- What are the main principles to follow to minimize power consumption?