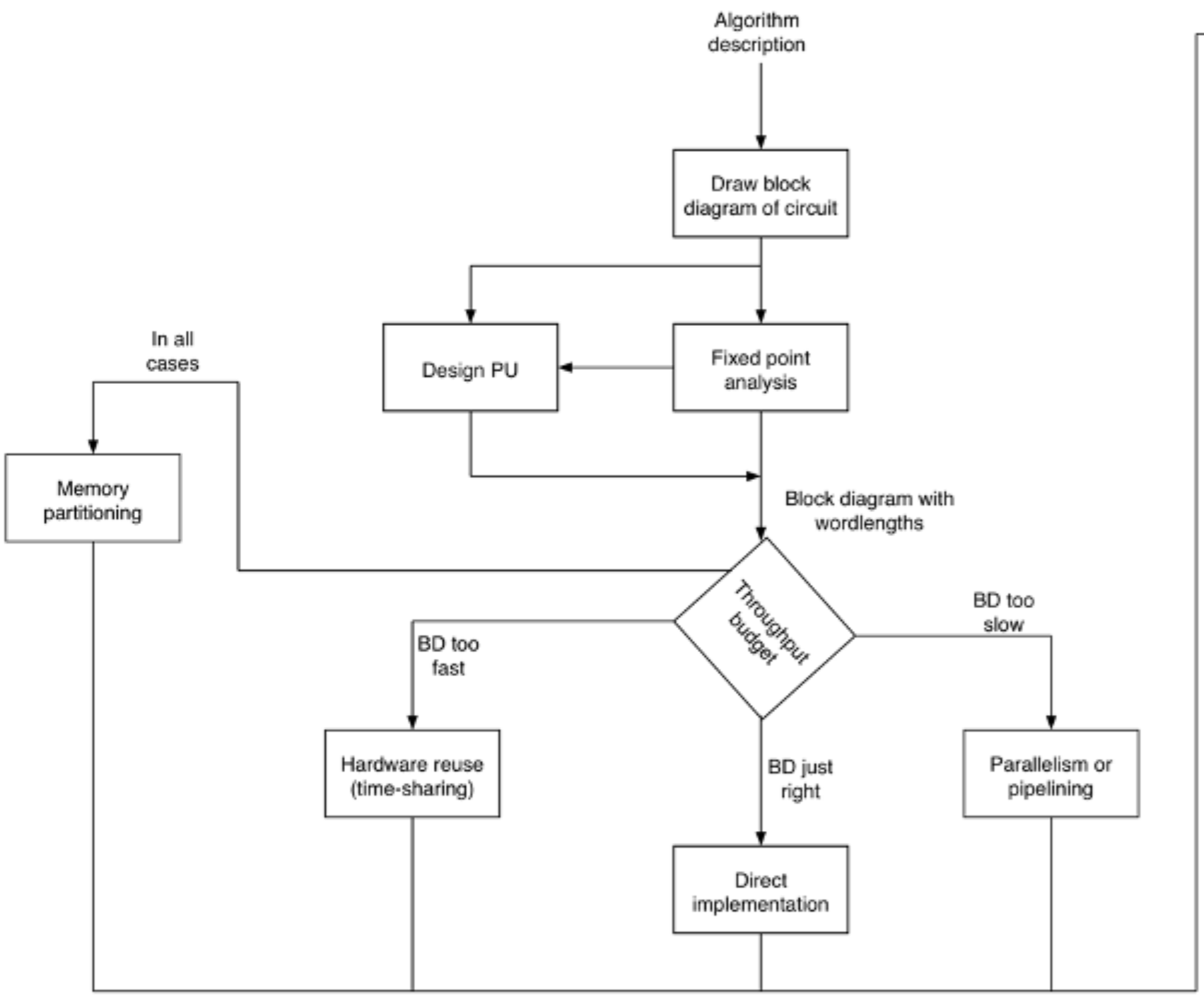
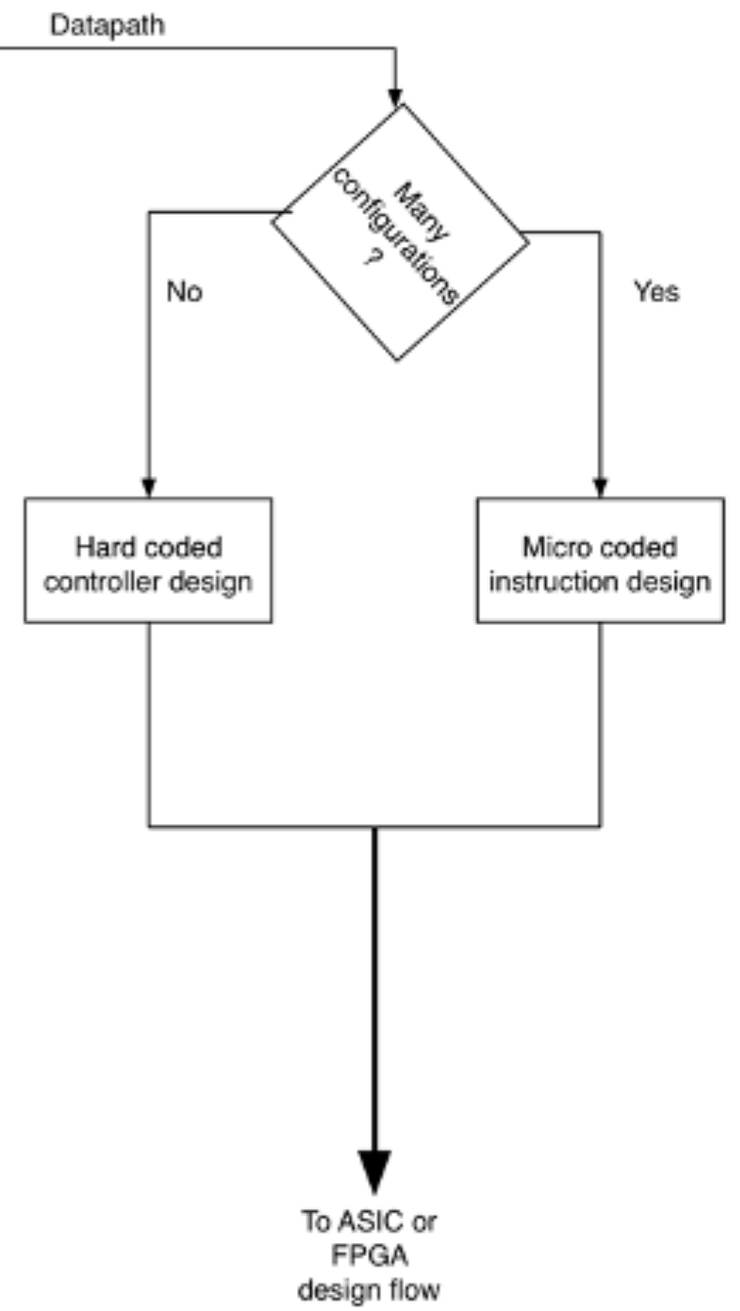
4. HIGH LEVEL DESIGN

ARCHITECTURE DESIGN FLOW





WHAT IS THIS EXACTLY?

- This is not the hardware design flow
- It takes the algorithm and takes it to a design ready for hardware implementation

WHAT DO I GAIN IF I DO IT?

- This makes the hardware design much easier
- It bridges the gap between the algorithm and what actually gets implemented in silicon

THE ALGORITHM

- This is a high level description
- Could be a flowchart
- Could be a program
- Could be a paragraph
- We do know it is at best floating point
- It also has no conception of how anything gets implemented

ISOLATE PU

- First break down operations
- Try to deduce as few PUs as possible
  - That are used repeatedly
- Think ahead to how these PUs look like in hardware
  - If there is a PU that performs A*X+B this is probably fine
  - If there is a PU that performs abs(sqrt(sin(x)+jtanh(y))) then perhaps this should give us pause

FIXED POINT SIMULATION

- Every operation in the algorithm is transformed to fixed point
- By trial and error (using high level metrics as a measure) determine the sizes of all busses
- The sizes of busses determine the sizes of arithmetic in the PUs

THROUGHPUT BUDGETING

- From specs understand required throughput
- From implementation platform, estimate PU throughput
- Figure out if there is a throughput deficit from a direct implementation
  - Direct architecture
  - Parallelism
  - Hardware reuse
  - Pipelining

MEMORY ALLOCATION

- With large architectural decisions made, move to memory
- Memory allocation is simple except for hardware reuse
- Bank memories to avoid contention
- Figure out schedule for data reads and writes from different memory banks

MAKE A DECISION ABOUT THE CONTROLLER

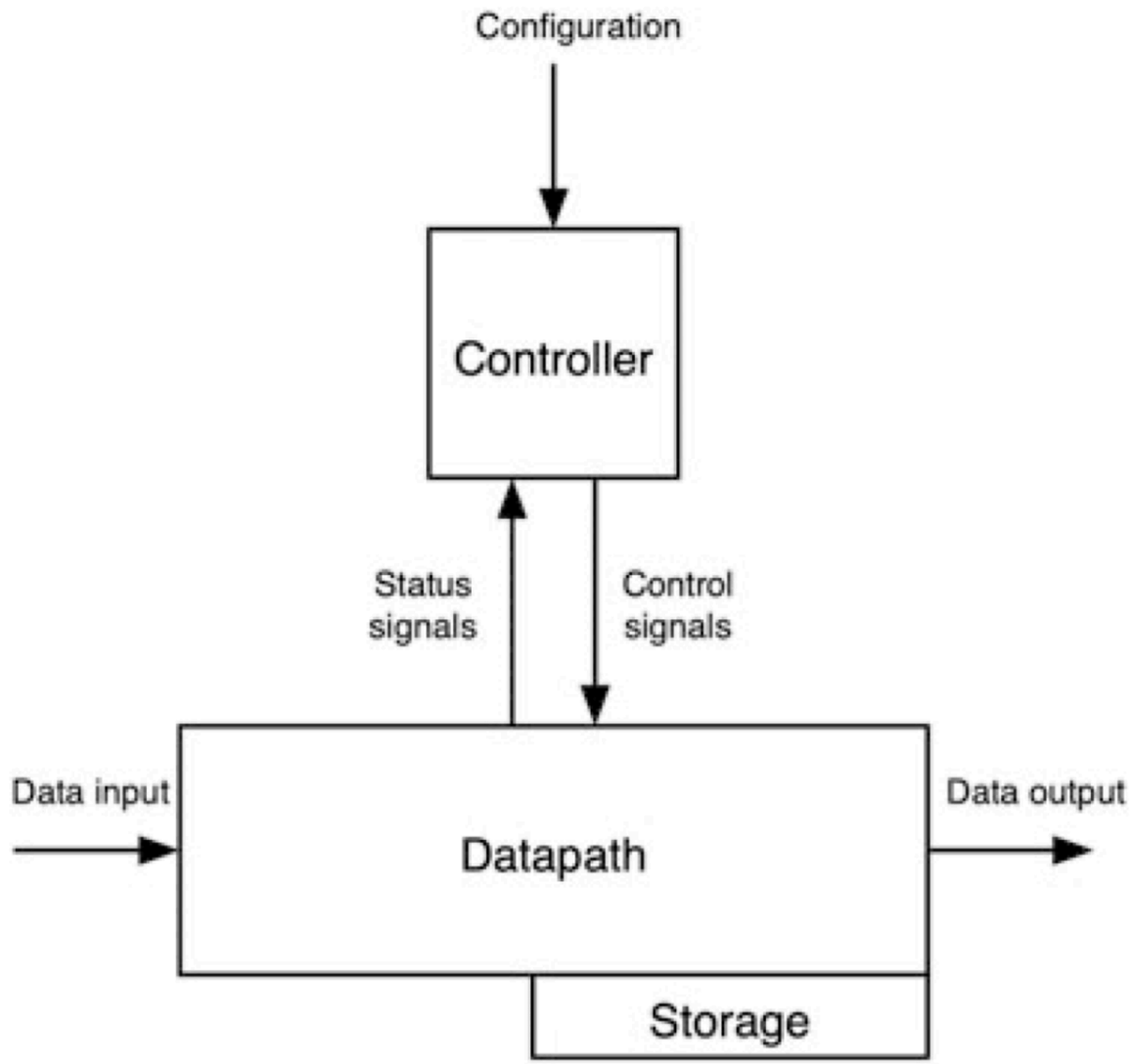
- Does your circuit have many modes?
  - If no, then take your memory control and go to a state machine design
  - If yes, then consider an instruction-based controller

PAPER DESIGN

- End up with a design document ready for implementation
  - PU design
  - Architecture using PU
  - Memories
  - Controller design
  - Bus sizes

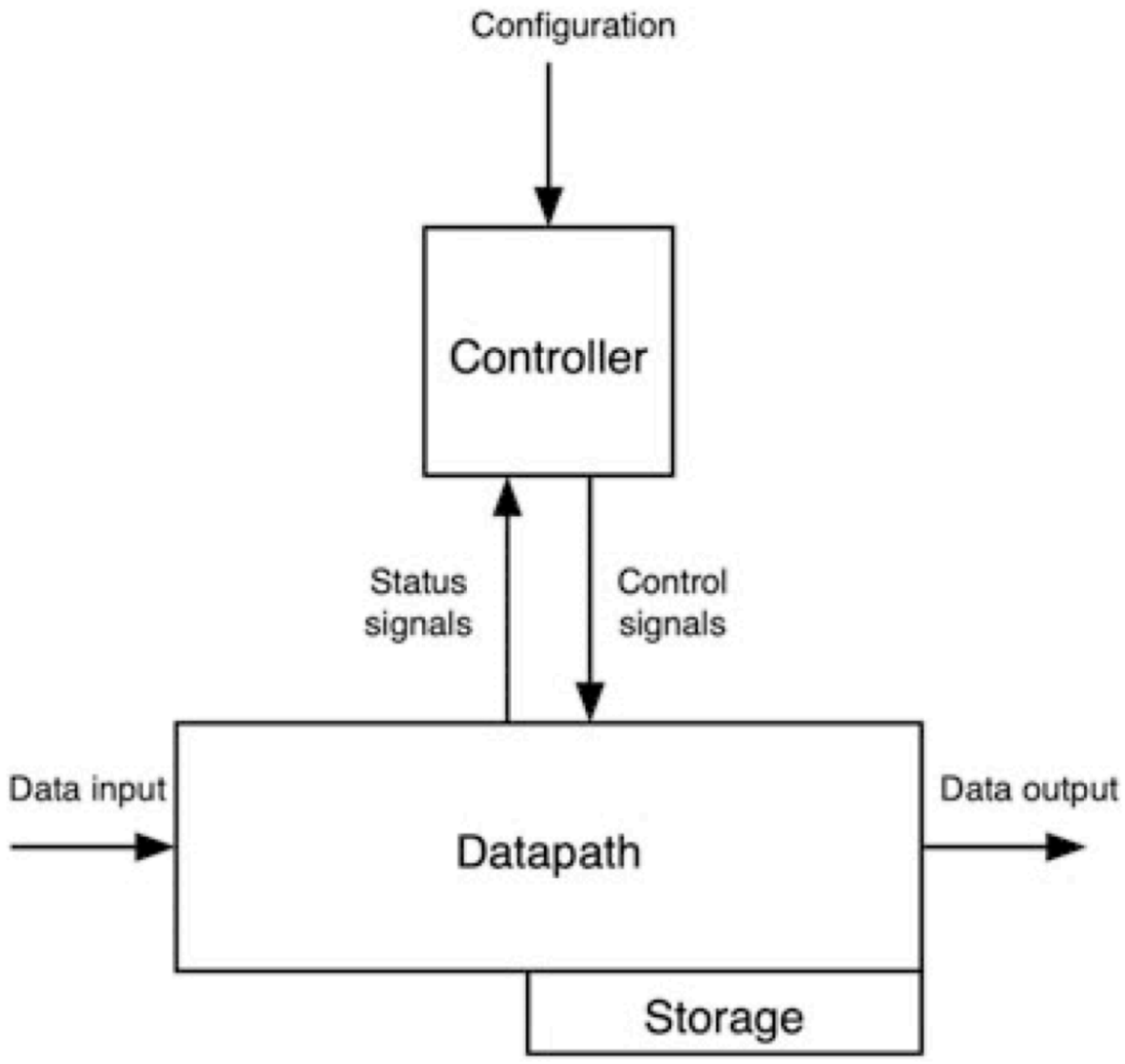
DESIGN PARADIGM

- Datapath is where most processing takes place
  - Where most area and power is
- Controller is where intelligence is
- Communication is only as shown



DESIGN PARADIGM

- Data inputs and outputs should only go to or come from datapath
- Controller must only sample the data path through status signals
- Control signals go from the controller to the data path
- Controller must figure out how to move from status and current state only
- Datapath must not need any more config than the control signals



DATAPATH

- This is where all processing takes place
- A combination of PUs, flow control, and sometimes storage
- No signal processing can take place outside the data path
- No data inputs can go to or come from anywhere else
- Controller samples data path signals to determine what to do next
- Controller imposes what to do next on data path by control signals
- Control signals reconfigure PU, manage memory addresses, and configure flow

PROCESSING UNITS

- This is the main component of the data path
- Try to reduce the number of types of PUs as much as possible
- The number of PUs (as opposed to types) is determined by the architecture later on
- Think about PUs that are repeated a lot carefully
- Determine if you will pipeline between PUs
  - Inside them (not recommended)
- What is your granularity?
  - Butterfly?
  - Multiplier?
  - Full adder?

STORAGE

- This could be distributed in the form of disperse registers
- Could be centralized memory blocks
- Particularly important for architectures that have hardware reuse
- Address busses, write enables, read enables (if any) are important tasks for the controller to figure out

CONTROLLER

- This sets the pace of the circuit
- Reconfigure configurable PUs
- Provide memory addressing and MUX selection
- Register controls
- Can be hard-coded or so instruction driven that it blurs the line to processors

FLOW CONTROL

- Flow control means anything that reroutes data in the data path
- It is the actuator that imposes the will of the controller
- Can be a mix of MUXes (to pick inputs) and registers to set the storage of outputs

ADDRESS GENERATION

- This is part of the controller
- This could be as simple as a counter
- But it often is not
- And is a large component of the controller complexity

HOW TO STORE IN AN N-BIT REGISTER

- The simplest way to store is to consider the entire register an integer value
- Thus a 4-bit register can store numbers between:
  - “0000” -> 0
  - “1111” ->15
- Thus, in general between 0 to 2 ^ n-1
- There are two questions about this:
  - What is the basis of this representation
  - How do we represent negative numbers

SIGN REPRESENTATION (SIGN-MAGNITUDE)

- In the sign magnitude approach the MSB is the sign bit
- The rest of the register is the magnitude
- Thus “0001” = 1
- “1001”=-1
- The 4 bit register can represent numbers between +7 and -7
- And thus 2^(n-1)-1 as an absolute value

SIGN REPRESENTATION (ONE'S COMPLEMENT)

- In the one's complement approach the entire number is complemented if it is negative
- Again the range of representations is from -7 to 7
- Both one's complement and sign-magnitude have the very wasteful property of representing two zeros

```
0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = -7
1001 = -6
1010 = -5
1011 = -4
1100 = -3
1101 = -2
1110 = -1
1111 = -0
```

SIGN REPRESENTATION (TWO'S COMPLEMENT)

- Positive numbers appear as expected
- Negative numbers are the complement of -16
- The range of numbers is 7 to -8
- There is no zero duplication
- Addition and subtraction are the same operation
- Two's complement is the representation of choice

```
0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = 8 - 16 = -8
1001 = 9 - 16 = -7
1010 = 10 - 16 = -6
1011 = 11 - 16 = -5
1100 = 12 - 16 = -4
1101 = 13 - 16 = -3
```

FLOATING POINT NUMBERS

- The register is not all integer. There is a fractional part
- Where is the binary point?
- In floating point numbers the register is divided into
  - Multiplier
  - Base
  - Exponent

FLOATING POINT NUMBERS

- Floating point registers allow an enormous range of numbers to be represented
- If the exponent is allowed to be negative, extremely small numbers can be represented
- Addition of floating point numbers can be very slow

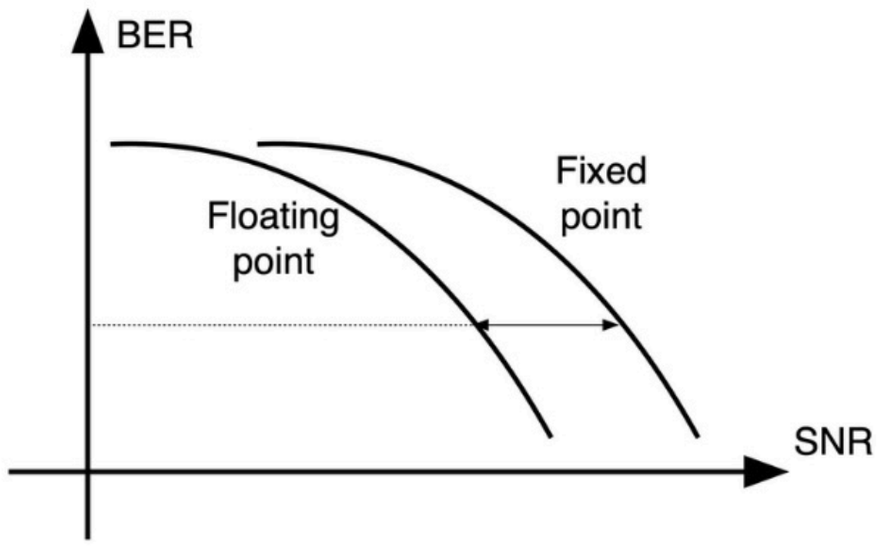
$$Multiplier \times Base^{Exponent}$$

FIXED POINT NUMBERS

- In a fixed-point register the position of the binary point is fixed
- The register is divided into fixed integer and fraction parts
- The range of numbers represented is much smaller
- As a (not so) special case, a full integer register is a fixed point register with the binary point to the right of the LSB
- This gives an idea of the range of numbers
- Fixed-point arithmetic is extremely efficient
- Thus most ASICs use fixed-point arithmetic

QUANTIZATION NOISE IN FIXED POINT REGISTERS

- Floating point registers are much wider than fixed-point
- High-level simulations are always floating-point
- Thus, hardware takes a hit versus simulation just due to number representation
- You can calculate this noise as “quantization noise”
- But ultimately what matters is the hit to a high-level quality metric (e.g. SNR here)



WHAT IS A FIXED-POINT NUMBER?

- We will argue here that the position of the binary point is irrelevant as long as it is preserved consistently across all operations
- The position of the binary point introduces a power of 2 factor to the numbers
- But the proportion of all numbers, and thus their addition and multiplication is preserved for all cases
- In such case, the easiest way to represent a fixed point number is as an integer

POSITION OF THE BINARY POINY

- The examples to the right shows that as the binary point is shifted, all numbers are divided by two
- The ratio of all numbers is preserved
- The results of their addition and multiplication are preserved
- The difference is a scaling factor that is a power of 2

```
111.0 = 14/2 = 7
000.1 = 1/2 = 0.5
001.1 = 3/2 = 1.5
100.1 = 9/2 = 4.5
010.1 = 5/2 = 2.5
```

```
11.10 = 14/4 = 3.5
00.01 = 1/4 = 0.25
00.11 = 3/4 = 0.75
10.01 = 9/4 = 2.25
01.01 = 5/4 = 1.25
```

```
1.110 = 14/8 = 1.75
0.001 = 1/8 = 0.125
0.011 = 3/8 = 0.375
1.001 = 9/8 = 1.125
0.101 = 5/8 = 0.625
```

```
0.1110 = 14/16 = 0.875
0.0001 = 1/16 = 0.0625
0.0011 = 3/16 = 0.1875
0.1001 = 9/16 = 0.5625
0.0101 = 5/16 = 0.3125
```

PSEUDOCODE

- We will use pseudocode to represent algorithms
- This will adhere very closely to matlab syntax
- But if we need to deviate we will not let this hold us back

GENERATING A FIXED POINT NUMBER

- The variable “a” is drawn from a uniform distribution 0->1
- In the second line it becomes a floating point number between 0 and 2 ^ n-1
- On the fourth line, the fractional part is removed
- It becomes an integer between 0 and 2 ^ n-1
- This is an unsigned fixed-point number
- No prejudice assumed on position of binary point

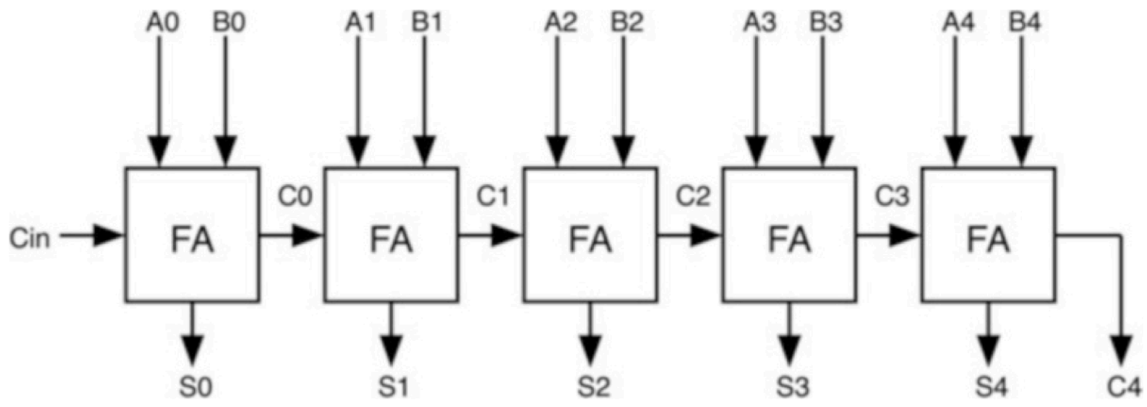
```
a = rand;

a = a × (2^n - 1);

afloating = a;
a = floor(afloating);
```

ADDITION IN HARDWARE

- The RCA is the simplest adder possible
- But it shows what happens with all fixed point adders
- If operands are N-bit, there is an extra carry out from the MSB
  - And the result is N+1 bits long



WHEN IS FLOATING AND FIXED POINT ADDITION THE SAME

- For all practical purposes floating point seems to be infinite for us
- Thus when is adding two N-bit numbers the same in fixed-point and floating point?
- Simply when the result of the fixed-point operation can be stored in N+1 bits

WHAT TO DO WITH THE EXTRA BIT

- We can do three things:
  - Store the result in N+1 bits
  - Throw away the LSB and store only N bits
  - Throw away the MSB and store only N bits

ALLOWING WORDLENGTH GROWTH

- Storing N+1 bits seems like the most logical answer
- It only costs 1 additional bit register at the output (which is negligible)
- The real cost is in subsequent operations
  - If our result is the operand for operations down the line, then it sets the size of these operations
  - The following addition will be and N+1 bit adder

TRIMMING THE CARRY OUT

- The second option is to pretend the result is N bit and ignore the MSB (carry out)
- This will often lead to perfect results (all on the left below)
- When the MSB is significant, it leads to completely wrong results (right below)
- We call this overflow

```
0000 + 0001 = 0001
0 + 1 = 1
0011 + 0100 = 0111
3 + 4 = 7
0111 + 0111 = 1110
7 + 7 = 14
1000 + 0010 = 1010
8 + 2 = 10
```

```
1000 + 1000 = 10,000= > 0000
8 + 8 = 16 = > 0
1110 + 1111 = 11,101 = > 1101
14 + 15 = 29 = > 11
0111 + 1111 = 10,110 = > 0110
7 + 15 = 22= > 6
0001 + 1111 = 10,000= > 0000
1 + 15 = 16= > 0
```

THROWING THE LSB

- In this case the LSB is dropped and we store N bits
- The “scale of results is changed, but this is fine
- Sometimes (even results in unsigned), the result is perfect
- Sometimes (odd results) it is a bit off
  - It is not “wrong”, there is just some noise on top of it
- We call this quantization noise
- Can you calculate it below?

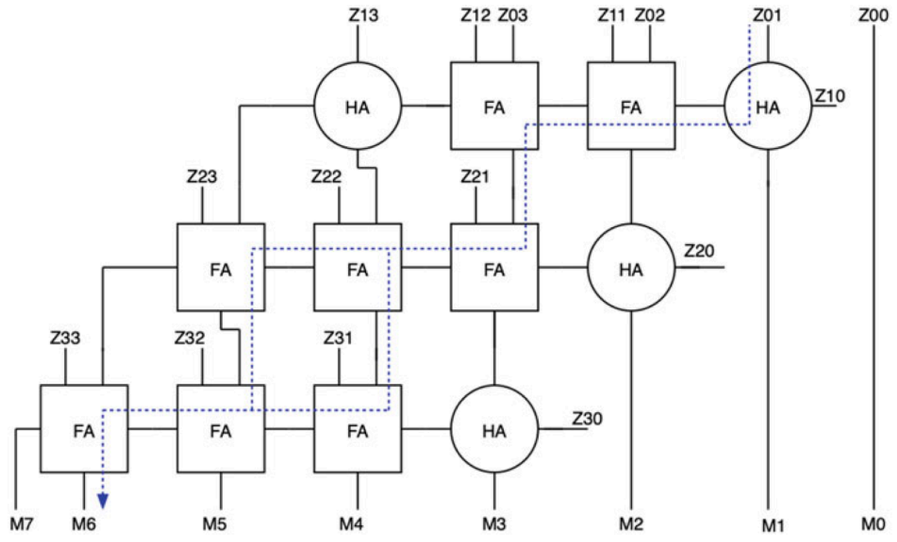
```
1111 + 1111 = 11,110= > 1111
1111 + 1110 = 11,101= > 1110
```

WHICH APPROACH IS MOST COMMON

- There is no need to allow overflow for adders
  - This works only for floating point units where overflow occurs for extremely large or small results
- Actual growth in word length is log2(M) where M is the number of addition stages
  - There is little to justify dealing with the impact of quantization noise
- Normally, we just allow the results of adders to grow because growth is extremely slow (it is not actually even linear)

MULTIPLIERS IN HARDWARE

- Hardware multipliers are implemented as shown below
- Although somehow faster alternatives exist, the basics are the same
- Multipliers have non-unique critical paths (good or bad?)



MULTIPLIERS VS ADDERS

- The delay of a multiplier is double that (roughly) of an adder with a similar input
- The area of multipliers grows quadratically
- But perhaps most critical is that the output is double input word length
  - Actually M+N for unequal inputs

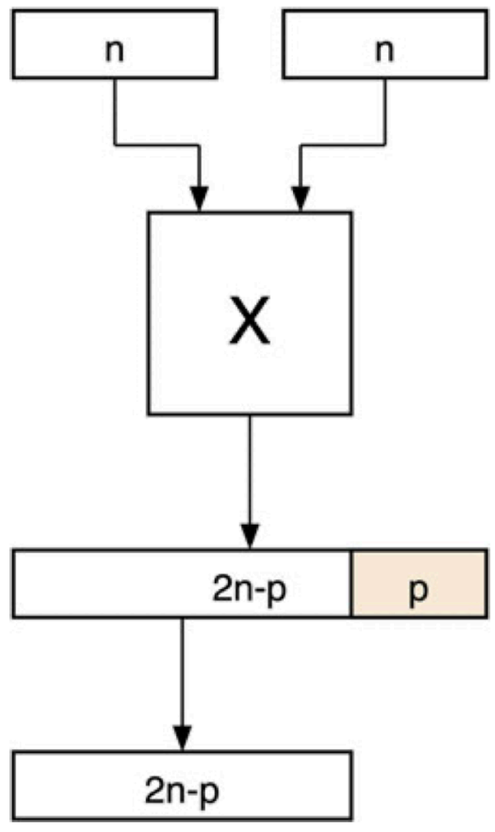
	Delay	Number of FAs	Output wordlength
Adder	$O(n)$	n	$n + 1$
Multiplier	$O(2n)$	n^2	$2n$

WHAT CAN WE DO WITH MULTIPLIER OUTPUTS

- Accepting word growth for multipliers is generally untenable
  - Not because of output registers but because of the size of subsequent multipliers (note size grows quadratically)
- Allowing overflow is also unacceptable
  - Multiplier outputs tend to skew higher
  - Multiplier “overflow” will take almost half of the output word
- The only available option is to truncate LSBs
  - And find an acceptable level of QSNR

TRUNCATION FOR MULTIPLIER OUTPUT

- Truncation is dropping a section of LSBs
- In this diagram we drop p bits and store only 2n-p
- The “noiseless” output is in 2n
- But how much noise is in the truncation



QSNR CALCULATION

- The first line create a floating point or noiseless result
- The second line just changes the scale of reference
- The third line removes the fractional part
  - Essentially storing the integer in 2n-p bits only
- The rest measures the noise
  - What kind of noise
  - This calculation is actually wrong
    - How do you fix it?

```
c = a × b;
c = c/2^p;
c = floor(c);
```

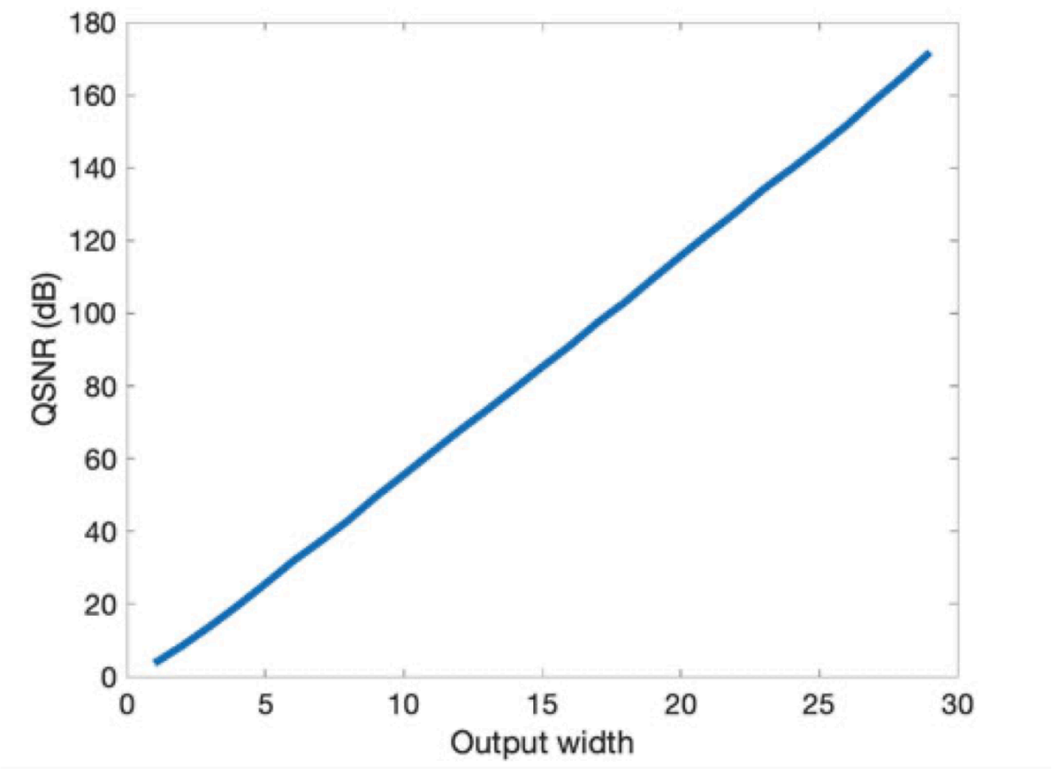
```
c_noiseless = a × b;
noise = c_noiseless-c;
```

QSNR SIMULATION

- This code calculates the quantization noise for different values of p
- This is a fixed point simulation
- Use it to determine sizes of registers

```
operand_size = w;
noiseless_size = 2 × w;

for m = 1:noiseless_size-1
   signal_power = 0;
   noise_power = 0;
   for iterations = 1:max_iterations;
      a = floor(rand × 2^operand_size);
      b = floor(rand × 2^operand_size);
      c_noiseless = a × b;
      c = floor(c_noiseless/2^(noiseless_size-M);
      current_noise=c_noiseless-c × 2^(noiseless_size-M);
      current_noise_power=current_noise^2;
      current_signal_power=c_noiseless^2;
       signal_power = signal_power + current_signal_power;
      noise_power = noise_power + current_noise_power;
   end;
   qsnr(m)=10 × log10(signal_power/noise_power);
end;
```



WHAT IS A GOOD QSNR?

- Generally QSNR that is much higher than fundamental sources of noise in the system (e.g. two or three decades higher)
- But ultimately you must measure the impact on a high-level metric
  - BER in communications
  - Compression rate
  - Recognition rate
- Ultimately something that relates to the application

CONSTANT MULTIPLIERS

- Constant multipliers are not actual multipliers

```
C1 = 12 × 3;
C2 = 4 × a;
C3 = a × b;
```

- There is only one “multiplier” to the right
- Implement each of the following using only adders and shifters

```
D1 = 2 × a;
D2 = 4 × a;
D3 = 5 × a;
D4 = 7 × a;
D5 = 125 × a;
```

- Ultimately all multipliers are adders-shifters but with no summands assumed trivial

MULTIPLIERS IN PRACTICE

- If present multipliers are the source of complexity in most digital circuits
  - They are bigger and slower than adders
  - The largest multiplier dominates critical path
    - Large area also dominates power
- Make an effort to use as few as possible

HOW MANY MULTIPLIERS ARE NEEDED?

- How many **multipliers** are in this pseudocode
  - The answer is we do not know
- There are five multiplications but we do not know how many multipliers are needed to perform them
  - To know we need the specs for both the application and the platform
  - Then we can figure out an architecture

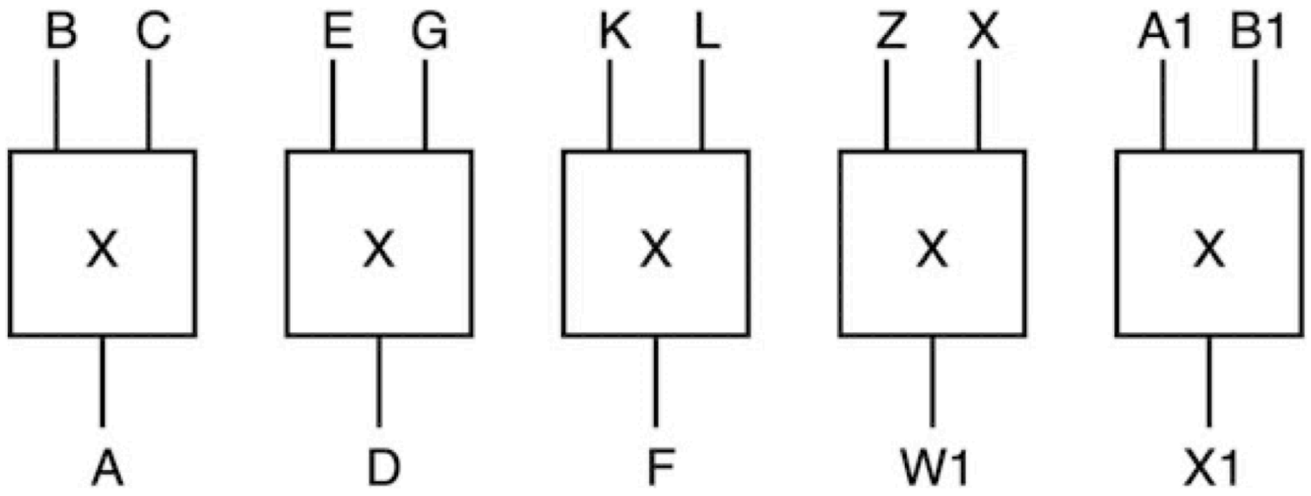
```
A = B × C;
D = E ×G;
F = K × L;
W1 = Z × X;
X1 = A1 × B1;
```

ALTERNATIVES TO IMPLEMENTATION

- Direct implementation
  - Every operation in pseudocode corresponds to a PU
- Parallelism
  - More PUs used then operations
- Pipelining
  - Can be combined with other architectures to reduce critical paths
- Hardware reuse
  - Fewer PUs are used then there are operations in code

DIRECT IMPLEMENTATION

- This is pretty simple
- Every operation corresponds to a multiplier
- Note the multiplier is our PU here




```
A = B × C;
D = E ×G;
F = K × L;
W1 = Z × X;
X1 = A1 × B1;
```

THROUGHPUT

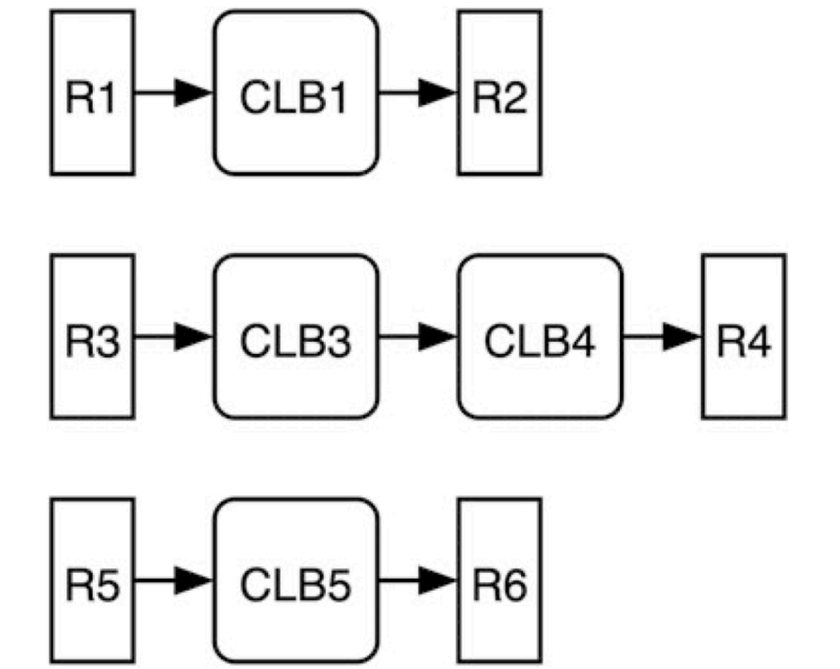
- Assume we use a platform where the multipliers can be clocked at 100MHz
- The direct implementation produces outputs at 100Msps
- Note very well what is a sample here
- It is a vector of outputs [A D F W1 X1]
- If we are talking about scalar products then the throughput is 500Msps
- It does not matter which you use as long as you communicate very well what is an "**s**" in M**s**ps

THROUGHPUT DISCREPANCY

- What if the throughput we get from the direct implementation is different from that needed
  - A throughput deficit will require parallelism or pipelining
  - A throughput excess is actually more common and requires hardware reuse
- With advanced technologies, processing units are more often faster than needed

PATHS, CRITICAL PATHS

- Speed in a digital circuit is controlled by the critical path
- A path is the sum of combinational delays between two consecutive registers
- It dictates the clock period
- The critical path is the longest such delay



THROUGHPUT AND LATENCY

- In a pipeline, throughput is the number of outputs per second
- The latency is the number of cycles it takes to get one useful output
  - It is also the number of cycles the pipeline takes to fill
- We must take both into consideration for some applications
  - This is particularly true for communication systems, especially packet-based systems

WHY IS POWER IMPORTANT

- Power dictates how long we take between battery charges
  - Actually a combination of power, “speed”, and usage
- A phone takes 7.56 microseconds to receive a packet
- Each packet has 1000 bits on average
- While receiving, the phone dissipates 1W
- It has a battery of 2000mAhr at a supply of 1V
- How many bits can the phone receive before needing a recharge?

FACTORS THAT AFFECT ACTIVE POWER

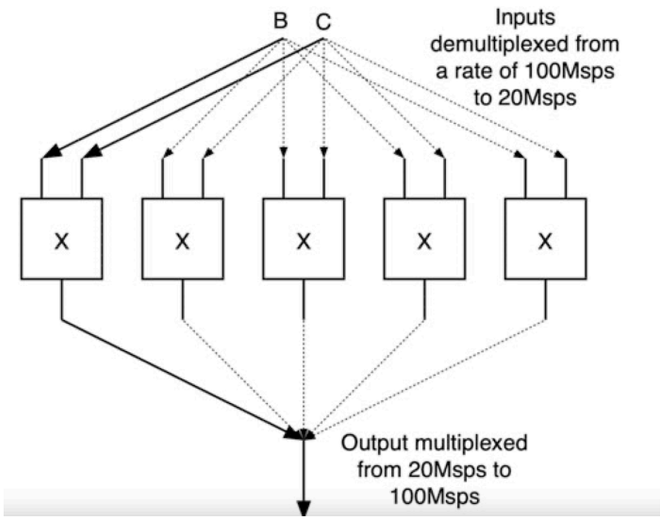
- Active power is a function of switching
- It is a function of area
- Operating frequency
- And above all supply
- But operating frequency is also directly proportional to supply

$$P = \alpha C f V_{DD}^2$$

$$f \propto V_{DD}$$

PARALLELISM

- Assume we need outputs at 100Msps but multipliers operate at (admittedly unrealistically slow) 20MHz
- We have a gap of five fold
- For A only this is going to need five mults
- The problem here is often providing inputs
- Is this design smart?
- Is this design stupid?



PIPELINING

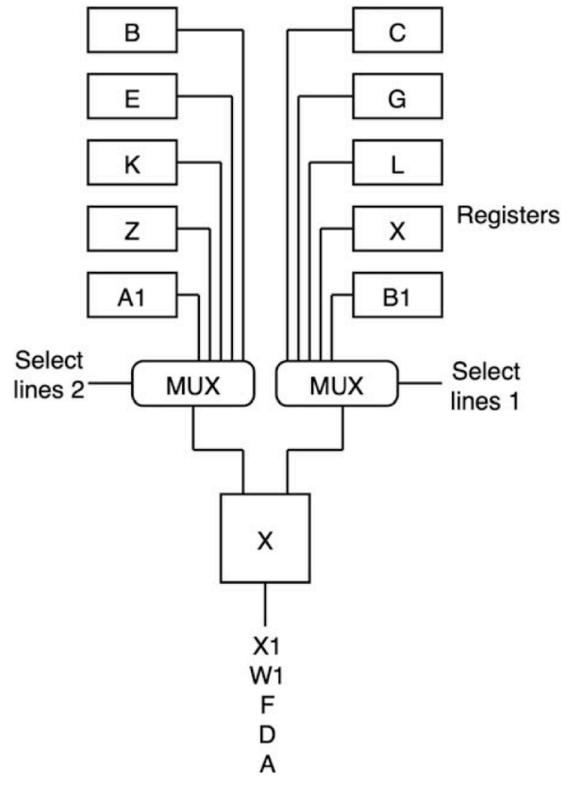
- A better way to increase throughput is to internally pipeline
- This raises frequency
- Increases latency
- And raises power
- But has a much lower area hit than parallelism
- The real cost of pipelining lies in complicated control and equalization of delays

UNDERCLOCKING

- If the elements are too fast, we can under clock
- For example, assume we can clock the multipliers at 100MHz, but need output at only 20Msps
- We can clock the multipliers at a lower frequency of 20MHz in the direct implementation
- Revisit the delay and power relations and see how you can leverage this
- What limitations does the chip impose on this
- What impact does this have on noise, if any?

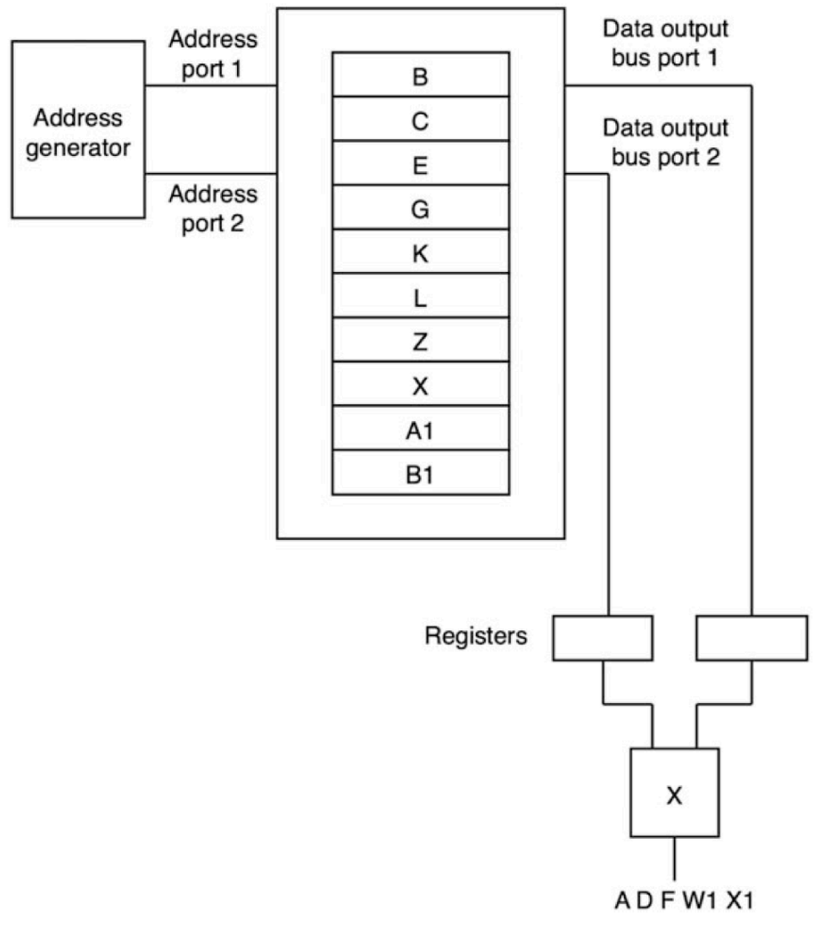
HARDWARE REUSE – PART 1

- This is the main approach to leverage fast PUs
- Here the multiplier is reused five times at 100MHz
- The effective throughput is 20Msps
- We need to control which MUX input we read from
- Also where do the outputs go?



MORE REALISTIC HARDWARE REUSE SETUP

- The extensive use of MUX and regs suggests we should use RAMs
- Control is now focused on the address generator
- We have a two port memory to simplify but this is not a must
- We have not yet answered where the outputs should go
  - Maybe regs with enables
  - Or maybe a memory too!



WHAT TO DO WITH OUTPUTS

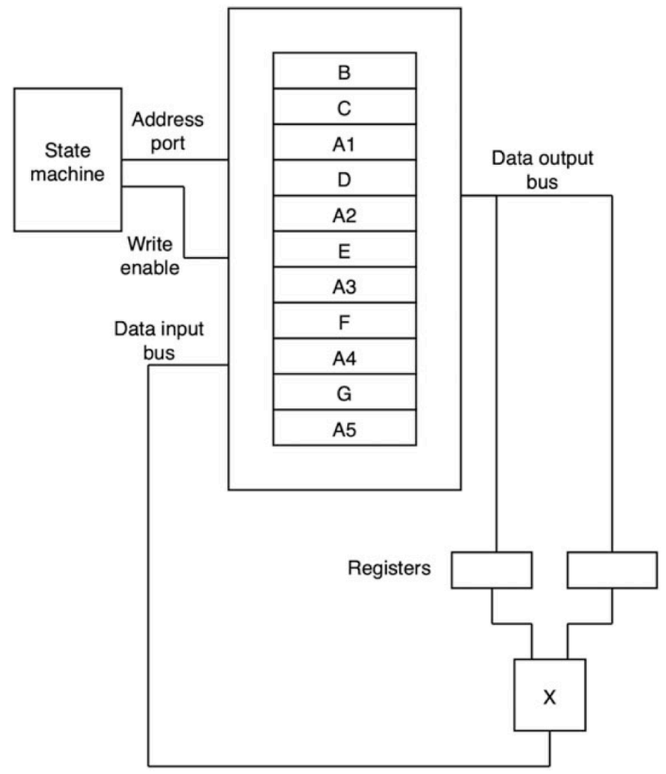
- Hardware reuse becomes more complicated with dependent operations
- What should we do with the output A1?
- We should absolutely save it somewhere we can get it
- Because it is an input for operation A1xD
- Thus, save it where inputs are saved

```
A1 = B × C;
A2 = A1 × D;

A3 = A2 × E;
A4 = A3 × F;
A5 = A4 × G;
```

HARDWARE REUSE WITH INTERDEPENDENCY

- Here the outputs are saved in the same memory bank as the inputs
- We assume a two port memory
- We dedicate one port to the inputs and one to the output
- Address generation (part of controller design) is now even more complicated
- Note the need for input registering



MEMORY BANDWIDTH

- Assume we can clock the PU at 100MHz
- We need to clock the memory three times per PU cycle
  - Twice for two inputs and once for one output
- But the memory is two-port, so we can clock it only twice
- We need the memory to operate at 200MHz
- What if the memory can only operate at 100MHz?

MEMORY BANK BREAKUP

- If there is memory bandwidth contention, we must partition the memory
- Here, we can split the memory into two banks
- Each can be clocked at 100MHz, leading to an effective read/write rate of 200MHz
- The main problem is how to distribute data between the banks

MEMORY BANK BREAKUP CONTENTION

- Your main issue is which ports of the PU are hard-wired to which memory ports (or are they hard-wired)?
- Assume for example output D is produced in one cycle and stored in bank 2, but must be read in the next cycle and the input is connected to bank 1
- You must find a way to tackle this without leading to contention again
  - Contention occurs when you need to read and write from and to the same bank more than its bandwidth can support

WHY BE COGNIZANT OF COST OF ARITHMETIC

- Systems designers use mathematics liberally
- In hardware, this translates into area, speed, and power
- We must be aware of the number of PUs we use
  - But also the arithmetic that makes up a PU
    - e.g. a single adder is not equal to a single multiplier
    - An 18-bit multiplier is not the same as two 4-bit multipliers

WHAT ARITHMETIC MIGHT YOU DO

- Addition (almost no circuit has no adders)
- Multiplication (extremely common)
- Division (becomes a major sticking point sometimes)
- Transcendental functions (we must figure out what to do with them)

ADDITION AND MULTIPLICATION

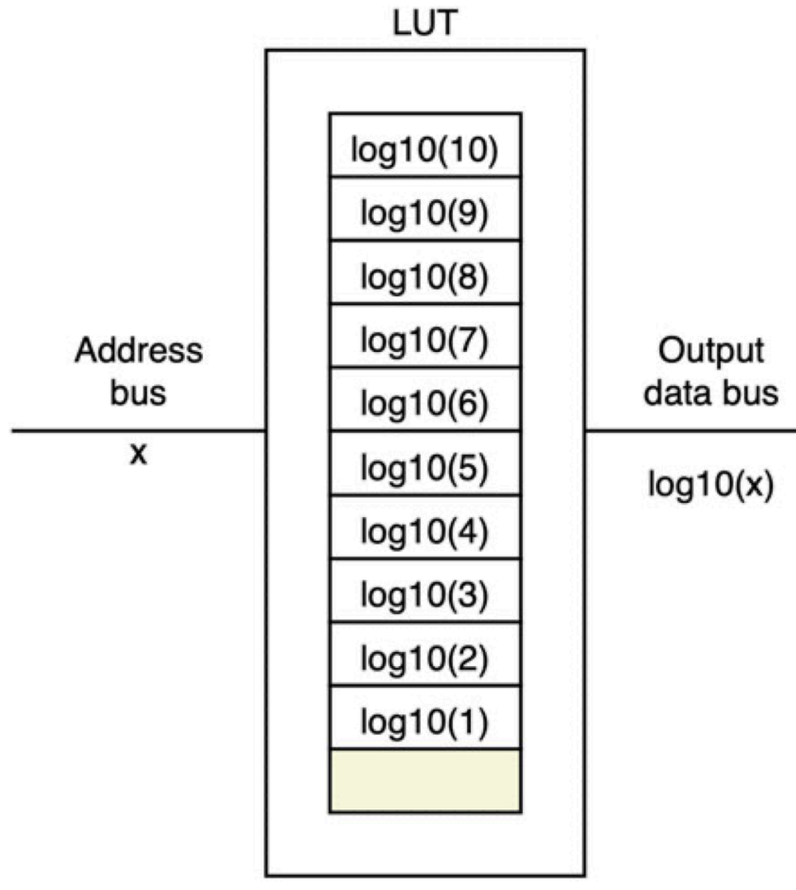
- These constitute the majority of our hardware
- Even other arithmetic uses them
- Adders grow linearly in area and in delay with wordlength
- Multipliers grow quadratically in area and their delay is nearly double a comparable adder (same input word length)
- Critical paths usually concentrate on paths with multipliers if any
  - Otherwise the longest adder or chain of adders

TRANSCENDENTAL FUNCTIONS

- This is any function that is not basic arithmetic
- All trigonometric functions
- Log and ln, exp, sqrt, angle(), abs() etc.
- CORDIC is a very efficient way to implement many of these functions using adders and shifters later
- But is a bit of a tangent

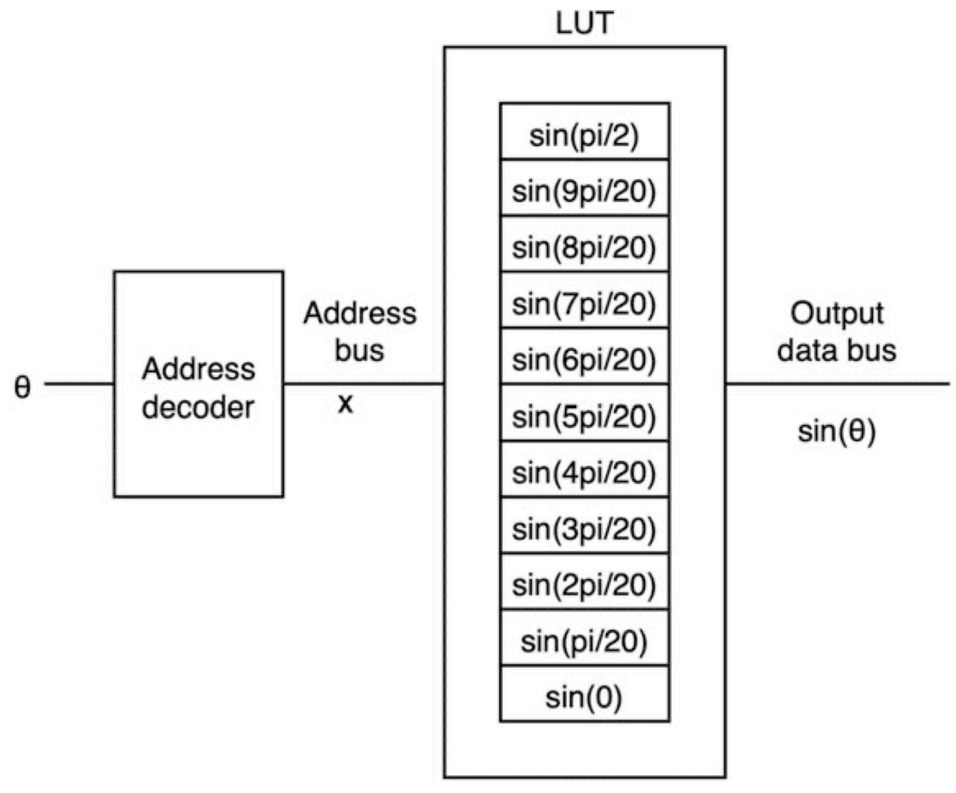
LUT BASED FUNCTIONS 1

- LUTs are the easiest way to implement transcendental functions
- Content of address x are f(x)
- Uses a single ROM
  - Normally a RAM loaded at initialization in practice
- Can be very memory intensive in functions with a lot of granularity in x



LUT BASED FUNCTIONS 2

- LUT based logic is not limited to integer addresses
- Insert an address decoder in between
- This translates arguments into integer addresses
- Some tricks are possible using symmetries to save on storage space (especially with trig)



HARDWARE DIVISION

- By division we mean dividing a number A by a smaller number D
- We get Q, the quotient and R the remainder

$$\frac{A}{D} = Q + \frac{R}{D}$$

HARDWARE DIVISION

- Repeated subtraction is the main hardware division method we have available
- This algorithm is simple
- But it has non-constant throughput

```
Q = 0;
R = A;
Repeat forever
   R = R - D;
   Q = Q + 1;
   If R < 0
      Break;
   End if;
End repeat;
Q = Q - 1;
R = R + D;
```

RESTORING DIVISION

- For fixed-point constant throughput division algorithms exist
- See for example the algorithm to the right

```
R = A;
D = D << N;
For i = N - 1:1:0
   R=2R - D;
   If R > 0
      Q(i) = 1;
   Else
      Q(i) = 0;
      R = R + D;
   End if;
End for;
```

WHAT DO YOU DO WHEN YOU FIND ARITHMETIC IN THE ALGORITHM

- You will accept most adders
  - The question is where to pipeline them
- Zoom in on multiplications, divisions, other functions
  - Are multiplications and divisions really so or are they shifts?
- Reduce their number as much as possible
- Is there a way to not do division (e.g. make it a multiplication)?

WHY USE FSM

- Finite State Machines are easy to characterize
- They are easy to debug and program
- There is very little that can go wrong here

WHY NOT USE FSM

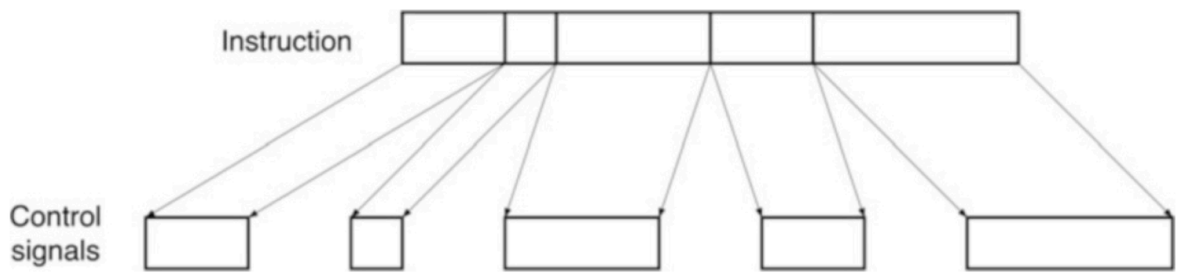
- FSMs can get excessively complicated for circuits with multiple modes
- They are also very bad at dealing with circuits that need reconfiguration on the fly
- It is important to distinguish whether reconfiguration happens before fabrication or after fabrication

INSTRUCTION BASED CONTROLLERS

- Circuits where reconfiguration happens on the fly are more akin to programming
- Thus, we are better off using an instruction-based approach to control
- We will build this up from simplest to most complex

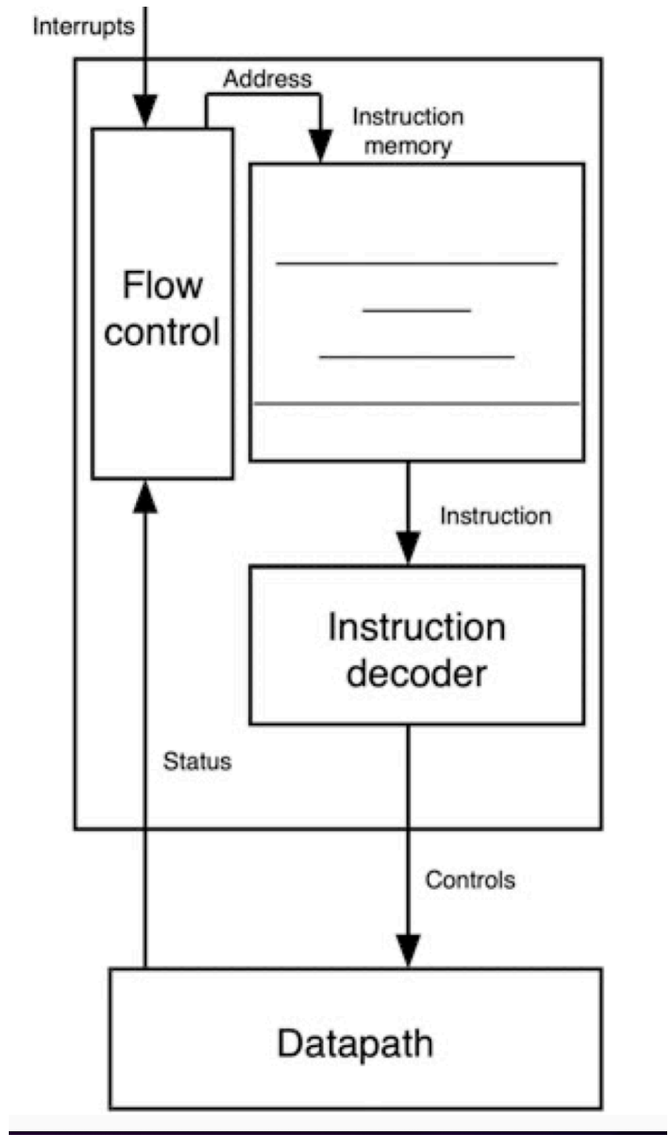
FULLY EXPANDED INSTRUCTION

- In this approach the instruction counter only counts up from zero
- The instruction is a concatenation of all control bits
- There are as many lines in the “program” as cycles in operation
- Word (instruction) can be extremely wide
- This is basically shoving the state machine into a RAM



CODED INSTRUCTION

- We consider the problem of instruction size
- If the concatenation of control bits is 100 bits long, this is 2 ^ 100 possibilities
  - Of which a tiny minority is legitimate
- Assume only 1000 possibilities are legitimate
  - Need only a 10 bit instruction
  - But also need a decoder to expand the 10 bits to 100 bits



UNCONDITIONAL JUMPS (LOOPS)

- We are still dealing with the fact that we can only count up by 1 for the instruction memory address
- If the address is obtained from an address generator rather than a counter, then we can sometimes loop over a section of code

CONDITIONAL BRANCHING

- But if the address generator can also make the decision to loop conditionally
  - Then we have if-else statements
- But what will the conditions test?
  - Status signals
  - Flags
  - Interrupts

WHERE IS THIS GOING?

- This is starting to resemble a microprocessor
- The line here is blurred
- Are we making an ASIC or an accelerator?
- How close to a processor should we go?