VLSI System Design

Vincent Immler
vincent.immler@oregonstate.edu

Prepared for: ECE474/574, Spring 2024.

Oregon State University

rootoftrust.io

Where Are We?

- **Week 1:** learned about basic gate modeling in HDL
  - Basic elements learned: FF, Latch, n-register, n-counter
  - Practiced how to write Verilog and testbench
- **Week 2:** learned about FPGA and its fabric
  - CLBs (mainly: LUT, FF, carry logic) and routing fabric
  - What is a “hardwire” vs. “procedural block” (edge detector)
  - Lab: learn how to SHR, SHL, ROR, ROL (perhaps also LOL)
  - Insight: shifting/rotating is basically for free
- **Week 3:** time to start the VLSI design process ← we are here
  - Topics: loop unrolling, pipelining (+ perhaps: retiming/unfolding, bit-slicing)
  - Example 1: Advanced Encryption Standard (AES)
  - Example 2: multiplier (and why you want to use a DSP instead)
  - Example 3: CRC checksum computation (LFSR)
  - Lab: using DSPs in the FPGA (i.e., how to avoid multiplier design)
  - Insight: “basic design knowledge”

Where Are We? (Cont'd)

- (*preliminary*) **Week 4:** learn more about the design process
  - Topics: more on implementation techniques; state machines
  - Lab: using BRAM in the FPGA and practice state machines
  - Insight: "more design knowledge"
- (*preliminary*) **Week 5:** "special assignment"
  - Goal is to make this our rotating assignment (new every year)
  - 474 students: implement given assignment (could be AES?)
  - 574 students: create new assignment (could be: multiplier, CRC, …)
  - Insight: "life is tough"
- (*preliminary*) **Week 6-end:** most likely: start with microprocessor

Digital Design: Problem Review

- Why?
  - Looking at problems using different perspectives
  - Avoid bad habits (ad hoc, asynchronous design, old technology)
  - You need a method that will work for more complex designs
- We will learn on simple designs, then add complexity

Digital Design Review / Very High Level Design Approach

- Why not jump more into (System-)Verilog now?
  - If you can't correctly do digital design, Verilog only helps you make mistakes more quickly
  - Verilog describes circuits, it does not design circuits. (coding ≠ design)
  - Verilog ≠ design tool. Block, timing and state machine diagrams are.
  - Typically, coding begins once the design is finished.
  - We use Verilog templates to implement our designs. Writing Verilog code should be a nearly mindless translation step.
  - Clever code ≠ clever circuit design. "Optimizing" Verilog code is useless in the conventional sense except for simulation. Synthesis probably ignores your clever code anyway!
  - Good designs stem from good architecture (structure), not clever code.

A problem occurs when we think that using (System-)Verilog is all about "coding up" designs. Coding is not the emphasis. Instead, the focus is on describing an already defined hardware structure. The design process is not "coding". Design is a separate, earlier step, done with different tools.

Digital Design Review

- C and System Verilog are very different animals
  - C compilation is a translation from complex instructions to simple in a model that is inherrently sequential with a single thread of control.

```
1       for(i=0;i<=5;i++){numb++}
```

  becomes

```
1           mov i, 5;
2       loop: inc numb;
3           dec i;
4           jnz lp;
```

- Verilog “compilation” is very different
  - Logic synthesis *infers* or deduces from code structure a gate-level logic structure from the abstract description of circuit behavior given by a language that models parallel operation.

Example 1: AES

(a symmetric block cipher for encryption/decryption of data)

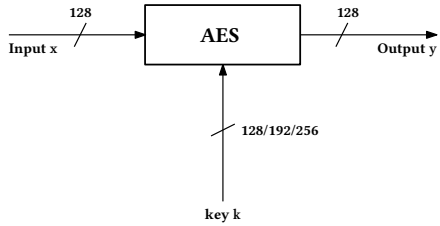
AES Timeline

- AES competition announced in January, 1997
- 15 candidates submitted by August, 1998
- 5 finalists announced in August 1999:
  - Mars, IBM Corporation
  - RC6, RSA Laboratories
  - Rijndael, Joan Daemen and Vincent Rijmen
  - Serpent, Eli Bihman et al.
  - Twofish, B. Schneier et al.
- Rijndael standardized as AES on October, 2001

(source: schneierfacts.com)

High-level View on AES

- AES supports 3 key lenghts: 128, 192, 256 bit
- 128 bit key already resistant against brute-force attack for foreseeable future
- No relevant weaknesses despite 20+ years of research in the crypto community
- NSA allows AES for classified documents up to TOP SECRET; similar to other countries

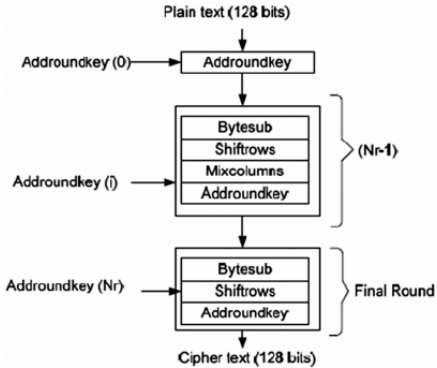


NSA algorithm suite linked here: [1]

[1] https://en.wikipedia.org/wiki/Commercial_National_Security_Algorithm_Suite

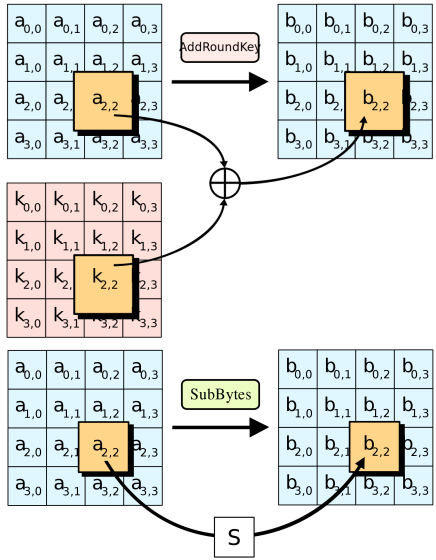
AES Basic Structure

- AES comprised of a Substitution Permutation Network (SPN) repeated for several rounds
- The block size and data path are 128 bit
- Number of rounds: 10, 12, 14 based on selected key length 128, 192, 256
- Notable components:
  - Key Addition: XOR with roundkey
  - Byte substitution: 8x8 S-Boxes
  - Diffusion Layer: create an avalanche effect by spreading influence of each input bit over many output bits
  - Here: diffusion layer created by ShiftRows and MixColumns



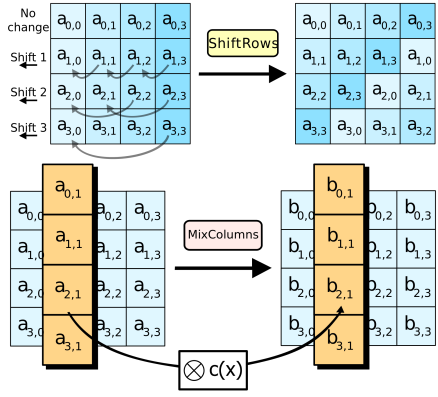
AES AddRoundKey and SubBytes

- 4x4 byte-wise matrix to represent internal state
- All fundamental operations with this matrix
- AddRoundKey is a bitwise XOR operation
  - Applies to full width of data path
- SubBytes operates on each byte independently
  - all Sboxes are the same
  - Sbox is inversion in GF(2^8)
  - followed by affine transformation



AES ShiftRows and MixColumns

- ShiftRows
  - Rotates each row by 0, 1, 2, 3 bytes to the left
  - In Software: adjust index
  - In Hardware: just (re-)wiring
- MixColumns
  - Columns represent quarter-state of AES
  - Each column is multiplied with fixed matrix
  - See next slides for details



Closer Look at MixColumns

- MixColumns is a matrix multiplication as follows:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

- Resulting in:

$$\begin{aligned} b_0 &= 02a_0 + 03a_1 + 01a_2 + 01a_3 \\ b_1 &= 01a_0 + 02a_1 + 03a_2 + 01a_3 \\ b_2 &= 01a_0 + 01a_1 + 02a_2 + 03a_3 \\ b_3 &= 03a_0 + 01a_1 + 01a_2 + 02a_3 \end{aligned}$$

- Since the elements are in GF(2^8), "special" math applies here
- Plus sign is simply XOR; Multiply by 2 and by 3 needs to be implemented

Multiply by 2 and 3 (from FIPS 197)

4.2.1 Multiplication by *x*

Multiplying the binary polynomial defined in equation (3.1) with the polynomial *x* results in

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x\,. \qquad (4.4)$$

The result $x \bullet b(x)$ is obtained by reducing the above result modulo $m(x)$, as defined in equation (4.1). If $b_7 = 0$, the result is already in reduced form. If $b_7 = 1$, the reduction is accomplished by subtracting (i.e., XORing) the polynomial $m(x)$. It follows that multiplication by x (i.e., {00000010} or {02}) can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with {1b}. This operation on bytes is denoted by `xtime()`. Multiplication by higher powers of x can be implemented by repeated application of `xtime()`. By adding intermediate results, multiplication by any constant can be implemented.

- To multiply by 2, we need to shift left, check the carry, if the carry is 1, the result must be XORed with 0x1b for polynomial reduction
- To multiply by 3, after multiplying by 2, result is XORed with input, i.e., 03a = 02a + a
- Many example AES codes can be found online making use of this (Do Not Use This)

AES Key Schedule

Omitted for reasons of brevity. It is not that important for what we want to do.

AES: How to Implement

- Relatively straightforward for initial functionality
  - Performance may not be well-adapted to targeted platform
  - Very unlikely to obtain a secure implementation
- In Software:
  - Sbox can be implemented as Look Up Tables (LUT)
  - 4 Sboxes and a MixColumns merged into 8-bit to 32-bit LUT called T-table
  - Typically the fastest approach if not constrained by memory (4kb)
- In Hardware:
  - Same as software when not constrained by memory (use BRAM?!)
  - Alternatively, composite field GF multipliers used for Sbox
  - In addition: various implementation styles possible for size/speed
  - Pipelined; iterated round designs: single Sbox, quarter, full

Risks of Look Up Tables . . . Caches

- Most platforms are optimized for performance and low power (not security!)
- Primary two reasons why complex processors use a cache
  - Performance: accessing a data block in memory is significantly slower than a cache access
  - Power consumption: accessing a data block in memory consumes more energy than a cache access
- Caches can be implemented as
  - data cache
  - instruction cache
  - unified data and instruction cache
- Nowadays, computers typically have 3 or more levels of caching!

Timing-Dependency of Caches

- Cache hits and misses affect the timing characteristics of the implementation
- Instruction flow may be constant, but data-fetch process is not
- If the crypto algorithm is implemented with LUTs and caches exist
  - Knowing if a cache miss or hit gives away information
  - Prevention is difficult, e.g., loading LUT in cache completely
- Information leak is essential to this type of attack
- Similar to pointer leaks for software exploits, this improper caching behavior can be provoked with suitable gadgets that interfere with regular caching

Example 2: 32x32 Multiplier

(and why you want to use a DSP instead)

Digital Design Review: 32x32 Multiplier

- Longhand multiplication

```
   1110
 x 1001
 ------
   1110
  0000
 0000
1110
-------
1111110
```

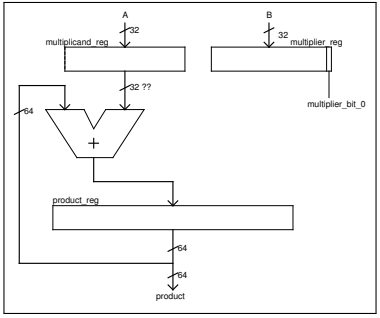
- Check first (LSB) bit of multiplier, if multiplier is a one, save a copy of multiplicand, else save all zeros.
- Check next bit of multiplier, if its a one, left-shift the multiplicand once and add it to the previously saved value. Else, add only zeros.
- Once all bits of the multiplicand are checked, and additions performed, the sum is the product.

Digital Design Review: 32x32 Multiplier

- Design constraints:
  - Synchronous (not combinatorial) using edge-triggered flip-flops
  - One clock, rising edge only, never gated
  - One reset, asynchronous, applied at power-up
  - One input to tell us to start, one output to indicate completion
- Now, ask, “what structures and paths do I need to implement the multiplication?”
  - Need storage for operands and product plus an adder
  - These will be flip-flops, not RAM. RAM makes no sense and is very expensive.
  - Then, what data paths will be needed between the blocks?

Digital Design Review: 32x32 Multiplier

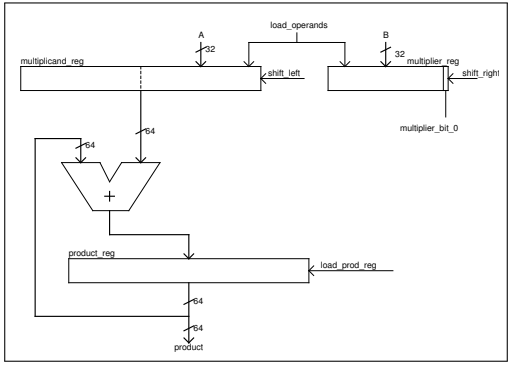
- First cut for storage and data paths
  - 32x32 multiplier must have a 64-bit output
  - Only looking at the LSB of the multiplier register
  - One clock, rising edge only, never gated
  - Clock and reset not shown. They are understood signals.



- What's missing, broken? What do we need now?

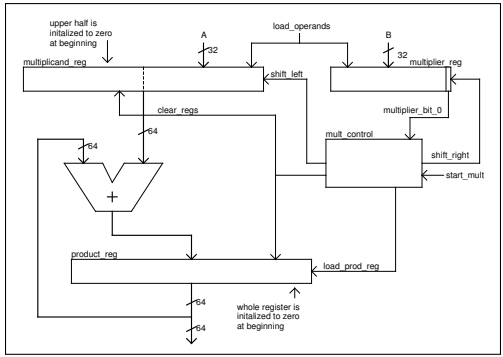
Digital Design Review: 32x32 Multiplier

- Clean up initial design - fix data paths, add some control stubs
  - Need 64-bit multiplicand register to shift left up-to 32 times
  - Need to be able to load the operand registers
  - Need to be able to load or hold value in product (free running clock)



Digital Design Review: 32x32 Multiplier

- Add remaining control signals
  - Upper 32 bits of multiplicand, product register needs to be cleared
  - Add "start" and "done" signals



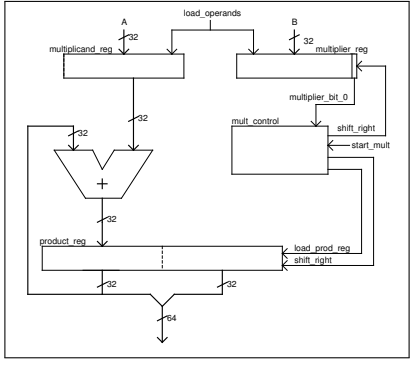
- High-level block diagram is done! Time for a *noggin simulation*.
- Control is abstracted away into its own block.

Digital Design Review: 32x32 Multiplier

- By now, you should see some optimizations that can be made.
- Approaching the design as we have exposes optimizations clearly.
- A "code first" approach will not reveal these optimizations.
- "Optimizing" your HDL code **will never create these optimizations**, as they are based in the structure of the hardware.
- "Optimizing" HDL code very often gives you exactly the same structure, only with more difficult to read code.

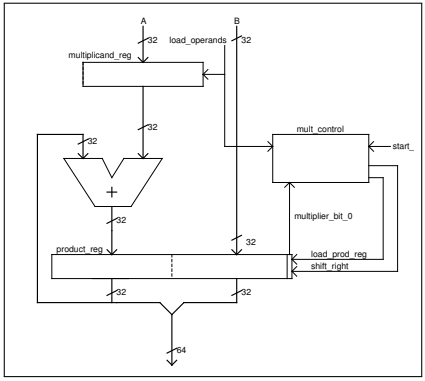
Digital Design Review: 32x32 Multiplier

- First Optimization
  - The multiplicand register is filled with zeros as its shifted left. Half of it only holds zeros that the 64-bit ALU had to process.
  - Consider: don't shift the multiplicand but shift the product right.
  - Result: 32-bit multiplicand register and ALU. No synchronous reset required for the product or multiplicand register.



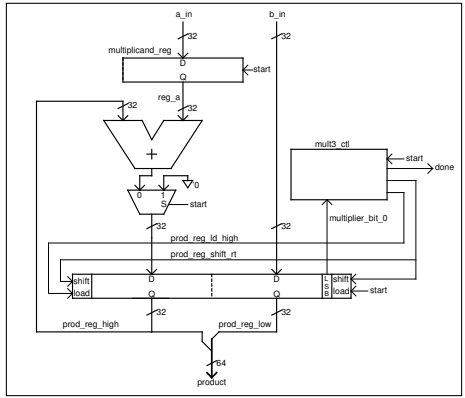
Digital Design Review: 32x32 Multiplier

- Final Optimization
  - Any values in the lower half of the product register are discarded as it is shifted right. The wasted space is exactly the size of the multiplier.
  - As wasted space in the product disappears, so does the multiplier.
  - Lower half of the product register temporarily stores multiplier.
  - Result: Removal of the 32-bit multiplier register and simpler control.



Digital Design Review: 32x32 Multiplier

- With an acceptable architecture we can create the control signals and tidy things up a bit. Fortunately, we have a simpler datapath to control.



Digital Design Review: 32x32 Multiplier

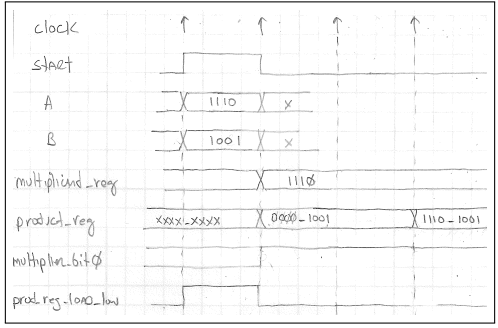
- Ground-rules for our control design
  - All control signals are synchronous to one clock (rising edge)
  - All control signals are created in synchronous state machines.
  - All control signals change state at rising clock edge.
  - All data movement happens at the rising clock edges.
  - Clock is never gated, we use enabled flip-flops.
- Use timing diagram and state machine diagrams to develop
- Start with either, go back and forth making sure they agree and that the signals are created at the correct time.

Digital Design Review: 32x32 Multiplier

- Why timing and state machine diagrams?
- Algorithms describe **what** things happen in a sequential manner. *Parallelism is hidden.*
- Timing Diagrams describe **when** things happen in a parallel manner. *Parallelism is exposed.*
- State Machine Diagrams separate parallel operations into easier to understand mini-machines.
- The visual nature of timing and state machines allow the use of visual pattern matching to observe commonality, see mistakes and to simplify our designs.

Digital Design Review: 32x32 Multiplier

- Timing and state machine diagrams are best worked out with paper and pencil or whiteboards.
- Producing these diagrams is another *eraser-intensive* exercise. Its hard work, and worth every minute.
- Time spent here is equal to 5x-10x time debugging HDL code.



- Some of this could become part of a future homework assignment

Questions?