Quick Points

- Midterm graded and returned
- Average – 84.5
- Median – 85.0
- Maximum – 95.0
- Minimum – 72.0
- Standard Deviation – 6.65

HW #4 Discussion

- Problem 1 – did just a simple adder work?
- Problem 2 – how did you implement the permutation table?
- Problem 3 – did you use a counter?

Overview of AES (cont.)

- 128-bit input is copied into a two-dimensional (4x4) byte array referred to as the state
  - Round transformations operate on the state array
  - Final state copied back into 128-bit output
- AES makes use of a non-linear substitution function that operates on a single byte
  - Can be simplified as a look-up table (S-box)

AES-128E Algorithm

- S-box transformation performed independently on each byte of the state
AES-128E Modules: ShiftRows

- Bytes in the last three rows of the state are cyclically over variable offsets.

\[
\begin{array}{cccc}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \\
\end{array}
\]

\[
\begin{array}{cccc}
S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\
S'_{1,1} & S'_{1,2} & S'_{1,3} & S'_{1,0} \\
S'_{2,2} & S'_{2,3} & S'_{2,0} & S'_{2,1} \\
S'_{3,3} & S'_{3,0} & S'_{3,1} & S'_{3,2} \\
\end{array}
\]

AES-128E Modules: MixColumns

- Modulo polynomial-basis multiplication performed on each column of the state.
- Can be simplified as series of AND and XOR operations.

\[
\begin{array}{cccc}
S_{0,0} & S_{0,2} & S_{0,3} \\
S_{1,0} & S_{1,2} & S_{1,3} \\
S_{2,0} & S_{2,2} & S_{2,3} \\
S_{3,0} & S_{3,2} & S_{3,3} \\
\end{array}
\]

\[
\begin{array}{cccc}
S'_{0,0} & S'_{0,2} & S'_{0,3} \\
S'_{1,1} & S'_{1,2} & S'_{1,3} \\
S'_{2,2} & S'_{2,3} & S'_{2,0} \\
S'_{3,3} & S'_{3,0} & S'_{3,1} \\
\end{array}
\]

MixColumns Implementation

```vhdl
entity MixColumns is
  port (STATE_IN    : in STATEtype;
        RNUM_IN       : in RNUMtype;
        STATE_OUT  : out STATEtype);
end MixColumns;

architecture behavior of MixColumns is
  signal tSTATE : STATEtype;
  begin
    process(STATE_IN)
    variable t1, t2 : std_logic_vector(7 downto 0);
    begin
      for i in 0 to 3 loop
        for j in 0 to Nb-1 loop
          -- Multiply by 2
          t1 := STATE_IN(i mod 4)(j)(6 downto 0) & '0';
          if (STATE_IN(i mod 4)(j)(7) = '1') then
            t1 := t1 xor x"1b";
          end if;
          -- Multiply by 3
          t2 := STATE_IN((i+1) mod 4)(j)(6 downto 0) & '0';
          if (STATE_IN((i+1) mod 4)(j)(7) = '1') then
            t2 := t2 xor x"1b";
          end if;
          t2 := t2 xor STATE_IN((i+1) mod 4)(j);
          tSTATE(i)(j) <= t1 xor t2 xor STATE_IN((i+2) mod 4)(j) xor STATE_IN((i+3) mod 4)(j);
        end loop;
      end loop;
    end process;
end behavior;
```

AES-128E Modules: AddRoundKey

- Words from the round-specific key are XORed into columns of the state.

\[
\begin{array}{cccc}
S_{0,0} & S_{0,2} & S_{0,3} \\
S_{1,0} & S_{1,2} & S_{1,3} \\
S_{2,0} & S_{2,2} & S_{2,3} \\
S_{3,0} & S_{3,2} & S_{3,3} \\
\end{array}
\]

\[
\begin{array}{cccc}
S'_{0,0} & S'_{0,2} & S'_{0,3} \\
S'_{1,1} & S'_{1,2} & S'_{1,3} \\
S'_{2,2} & S'_{2,3} & S'_{2,0} \\
S'_{3,3} & S'_{3,0} & S'_{3,1} \\
\end{array}
\]

AddRoundKey Implementation

```vhdl
entity AddRoundKey is
  port(STATE_IN : in STATEtype;
       KEY_IN : in KEYtype;
       STATE_OUT     : out STATEtype);
end AddRoundKey;

architecture behavior of AddRoundKey is
  begin
    process(STATE_IN, KEY_IN)
    begin
      for j in 0 to (Nb-1) loop
        STATE_OUT(0)(j) <= STATE_IN(0)(j) xor KEY_IN(j)(31 downto 24);
        STATE_OUT(1)(j) <= STATE_IN(1)(j) xor KEY_IN(j)(23 downto 16);
        STATE_OUT(2)(j) <= STATE_IN(2)(j) xor KEY_IN(j)(15 downto 8);
        STATE_OUT(3)(j) <= STATE_IN(3)(j) xor KEY_IN(j)(7 downto 0);
      end loop;
    end process;
  end behavior;
```

AES-128E Modules: KeyExpansion

- Initial 128-bit key is converted into separate keys for each of the 10 required rounds.
- Consists of Sbox transformations and some XORs.

\[
\begin{array}{cccc}
Rkey_{10} & Rkey_{9} & Rkey_{8} & Rkey_{7} \\
Rkey_{6} & Rkey_{5} & Rkey_{4} & Rkey_{3} \\
Rkey_{2} & Rkey_{1} & Rkey_{0} \\
\end{array}
\]
Design Decisions

- Online/offline key generation
- Inter-round layout decisions
  - Round unrolling
  - Round pipelining
- Intra-round layout decisions
  - Transformation pipelining
  - Transformation partitioning
- Technology mapping decisions
  - S-box synthesis as Block SelectRAM, distributed ROM primitives, or logic gates

Round Unrolling / Pipelining

- **Unrolling** replaces a loop body (round) with \( N \) copies of that loop body
- AES-128E algorithm is a loop that iterates 10 times
  - \( N \in \{1, 10\} \)
  - \( N = 1 \) corresponds to original looping case
  - \( N = 10 \) is a fully unrolled implementation
- **Pipelining** is a technique that increases the number of blocks of data that can be processed concurrently
  - Pipelining in hardware can be implemented by inserting registers
  - Unrolled rounds can be split into a certain number of pipeline stages
  - These transformations will increase throughput but increase area and latency

Round Unrolling / Pipelining (cont.)

Unrolling factor = 10
Unrolling factor = 2
Unrolling factor = 1
Unrolling factor = 5

Transformation Partitioning/Pipelining

- FPGA maximum clock frequency depends on critical logic path
- Inter-round transformations can’t improve critical path
- Individual transformations can be pipelined with registers similar to the rounds
- Transformations that are part of the maximum delay path can be partitioned and pipelined as well
- Can result in large gains in throughput with only minimal area increases

Transformation partitioning = ON

S-box Technology Mapping

- With synthesis primitives, can map the S-box lookup tables to different hardware components
- Two S-boxes can fit on a single Block SelectRAM

```vhdl
constant SSYMB_STYLE: string := "select_rom"; -- (logic, select_rom)

entity Sbox is
  port(BYTE_IN : in std_logic_vector(7 downto 0);
       BYTE_OUT : out std_logic_vector(7 downto 0));
attribute syn_romstyle : string;
attribute syn_romstyle of BYTE_OUT : signal is SSYMB_STYLE;
end Sbox;
```

Sample VHDL code
Recap – Retiming

Recap – Retiming (cont.)

weight(e') = weight(e) + lag(head(e)) - lag(tail(e))

Retiming and Pipelining

• Can use this retiming to pipeline
• Assume have enough (infinite supply) of registers at edge of circuit
• Retime them into circuit
• See [WeaMar03A] for details

Recap – Retiming and Covering

Outline

• HW #4 Discussion
• Recap
• HW/SW Codesign
  • Motivation
  • Specification
  • Partitioning
  • Automation

Hardware/Software Codesign

• Definition 1 – the concurrent and co-operative design of hardware and software components of an embedded system
• Definition 2 – A design methodology supporting the cooperative and concurrent development of hardware and software (co-specification, co-development, and co-verification) in order to achieve shared functionality and performance goals for a combined system [MicGup97A]
Motivation

- Not possible to put everything in hardware due to limited resources
- Some code more appropriate for sequential implementation
- Desirable to allow for parallelization, serialization
- Possible to modify existing compilers to perform the task

Why put CPUs on FPGAs?

- Shrink a board to a chip
- What CPUs do best:
  - Irregular code
  - Code that takes advantage of a highly optimized datapath
- What FPGAs do best:
  - Data-oriented computations
  - Computations with local control

Computational Model

- Most recent work addressing this problem assumes relatively slow bus interface
- FPGA has direct interface to memory in this model

Hardware/Software Partitioning

- If foo < 8 {
  for (i=0; i<N; i++)
    x[i] = y[i] * z[i];
}

Methodology

- Separation between function, and communication
- Unified refinable formal specification model
  - Facilitates system specification
  - Implementation independent
  - Eases HW/SW trade-off evaluation and partitioning
- From a more practical perspective:
  - Measure the application
  - Identify what to put onto the accelerator
  - Build interfaces

System-Level Methodology
Concurrency

- Concurrent applications provide the most speedup

No data dependencies

```
if (a > b) ...
CPU
\[ x[i] = y[i] \times z[i] \]
```

Partitioning

- Can divide the application into several processes that run concurrently
- Process partitioning exposes opportunities for parallelism

```
if (i > b) ... Process 1
for (i = 0; i < N; i++) ... Process 2
for (j = 0; j < N; j++) ... Process 3
```

Automating System Partitioning

- Good partitioning mechanism:
  1) Minimize communication across bus
  2) Allows parallelism \( \rightarrow \) both hardware (FPGA) and processor operating concurrently
  3) Near peak processor utilization at all times (performing useful work)

```
process (a, b, c) in port a, b; out port c;
{
read(a);
...
write(c);
}
```

Partitioning Algorithms

- Assume everything initially in software
- Select task for swapping
- Migrate to hardware and evaluate cost
  - Timing, hardware resources, program and data storage, synchronization overhead
- Cost evaluation and move evaluation similar to what we’ve seen regarding mincut and simulated annealing

Multi-threaded Systems

- Single thread:
- Multi-thread:

Performance Analysis

- Single-threaded:
  - Find longest possible execution path
- Multi-threaded with no synchronization:
  - Find the longest of several execution paths
- Multi-threaded with synchronization:
  - Find the worst-case synchronization conditions
Multi-threaded Performance Analysis

- Synchronization causes the delay along one path to affect the delay along another

\[ \text{Delay} = \max(t_a, t_b) + t_d \]

Control

- Need to signal between CPU and accelerator
  - Data ready
  - Complete
- Implementations:
  - Shared memory
  - Handshake
- If computation time is very predictable, a simpler communication scheme may be possible

Communication Levels

- Easier to program at application level
  - (send, receive, wait) but difficult to predict
- More difficult to specify at low level
  - Difficult to extract from program but timing and resources easier to predict

Other Interface Models

- Synchronization through a FIFO
- FIFO can be implemented either in hardware or in software
- Effectively reconfigure hardware (FPGA) to allocate buffer space as needed
- Interrupts used for software version of FIFO

Debugging

- Hard to test a CPU/accelerator system:
  - Hard to control and observe the accelerator without the CPU
  - Software on CPU may have bugs
- Build separate test benches for CPU code, accelerator
- Test integrated system after components have been tested

POLIS Codesign Methodology
Codesign Finite State Machines

- POLIS uses an FSM model for
  - Uncommitted
  - Synthesizable
  - Verifiable
Control-dominated HW/SW specification

- Translators from
  - State diagrams,
  - Esterel, ECL, ReactiveJava
  - HDLs
Into a single FSM-based language

CFSM behavior

- Four-phase cycle:
  - Idle
  - Detect input events
  - Execute one transition
  - Emit output events

- Software response could take a long time:
  - Unbounded delay assumption
  - Need efficient hw/sw communication primitive:
    - Event-based point-to-point communication

Network of CFSMs

- Globally Asynchronous, Locally Synchronous (GALS) model

Summary

- Hardware/software codesign complicated and limited by performance estimates
- Algorithms not generally as good as human partitioning
- Other interesting issues include dual processors, special memory interfaces
- Will likely evolve at faster rate as compilers evolve