EEL 4783: Hardware/Software Co-design with FPGAs

Lecture 5: Digital Camera: Software Implementation*

Prof. Mingjie Lin

UCF
Stands For Opportunity

* Some slides based on ISU CPrE 588

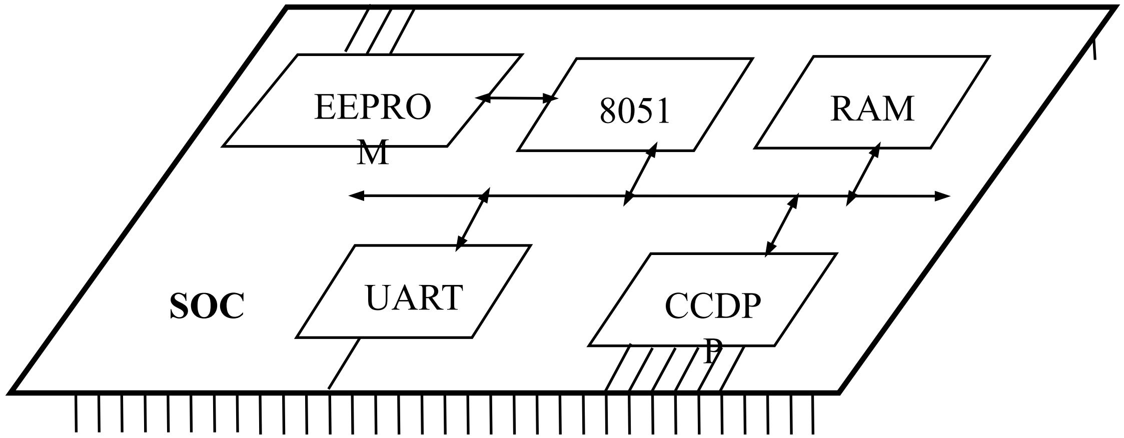
Design

- Determine system's architecture
  - Processors
    - Any combination of single-purpose (custom or standard) or general-purpose processors
  - Memories, buses
- Map functionality to that architecture
  - Multiple functions on one processor
  - One function on one or more processors
- Implementation
  - A particular architecture and mapping
  - Solution space is set of all implementations
- Starting point
  - Low-end general-purpose processor connected to flash memory
    - All functionality mapped to software running on processor
    - Usually satisfies power, size, and time-to-market constraints
    - If timing constraint not satisfied then later implementations could:
      - use single-purpose processors for time-critical functions
      - rewrite functional specification

Implementation 1

- Low-end processor could be Intel 8051 microcontroller
- Total IC cost including NRE about $5
- Well below 200 mW power
- Time-to-market about 3 months
- However, one image per second not possible
  - 12 MHz, 12 cycles per instruction
    - Executes one million instructions per second
  - *CcdppCapture* has nested loops resulting in 4096 (64 x 64) iterations
    - ~100 assembly instructions each iteration
    - 409,000 (4096 x 100) instructions per image
    - Half of budget for reading image alone
  - Would be over budget after adding compute-intensive DCT and Huffman encoding

Implementation 2

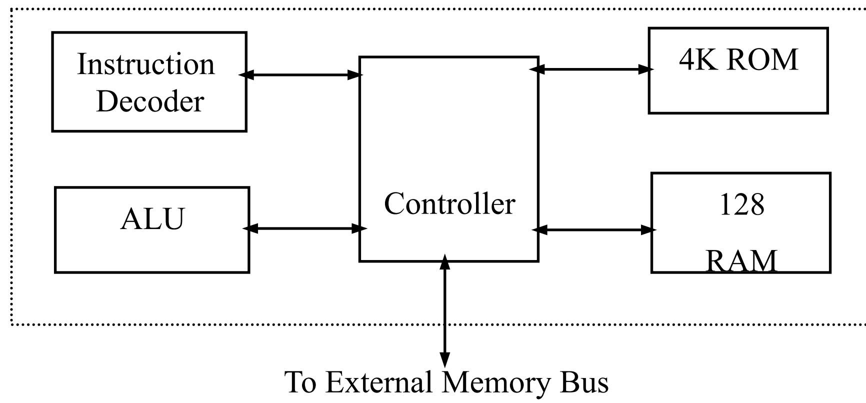


- CCDPP function implemented on custom single-purpose processor
  - Improves performance – less microcontroller cycles
  - Increases NRE cost and time-to-market
  - Easy to implement
    - Simple datapath
    - Few states in controller
- Simple UART easy to implement as single-purpose processor also
- EEPROM for program memory and RAM for data memory added as well

Microcontroller

- Synthesizable version of Intel 8051 available
  - Written in VHDL
  - Captured at register transfer level (RTL)
- Fetches instruction from ROM
- Decodes using Instruction Decoder
- ALU executes arithmetic operations
  - Source and destination registers reside in RAM
- Special data movement instructions used to load and store externally
- Special program generates VHDL description of ROM from output of C compiler/linker

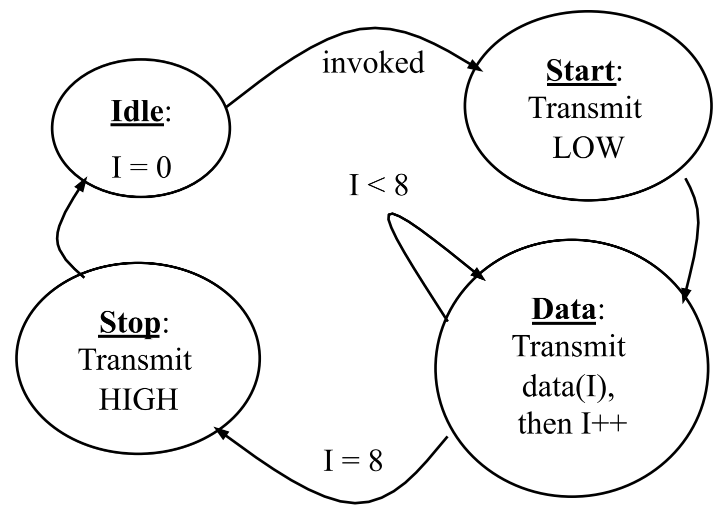
Block diagram of Intel 8051 processor core



UART

- UART in idle mode until invoked
  - UART invoked when 8051 executes store instruction with UART's enable register as target address
    - Memory-mapped communication between 8051 and all single-purpose processors
    - Lower 8-bits of memory address for RAM
    - Upper 8-bits of memory address for memory-mapped I/O devices
- Start state transmits 0 indicating start of byte transmission then transitions to Data state
- Data state sends 8 bits serially then transitions to Stop state
- Stop state transmits 1 indicating transmission done then transitions back to idle mode

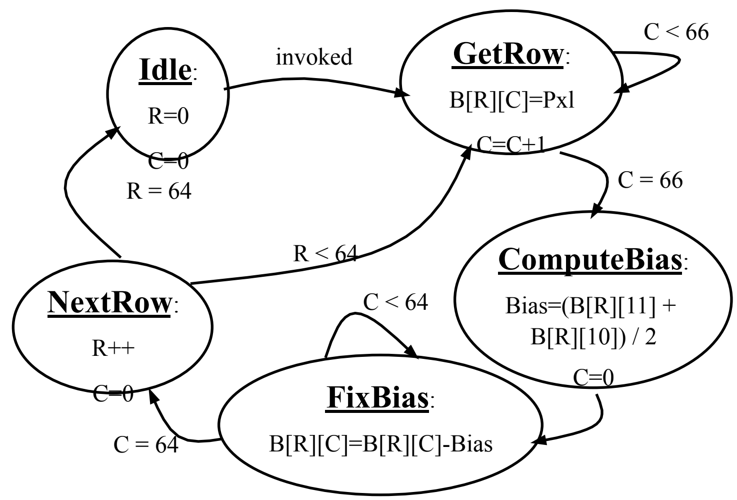
FSMD description of UART



CCDPP

- Hardware implementation of zero-bias operations
- Interacts with external CCD chip
  - CCD chip resides external to our SOC mainly because combining CCD with ordinary logic not feasible
- Internal buffer, *B*, memory-mapped to 8051
- Variables *R*, *C* are buffer's row, column indices
- GetRow state reads in one row from CCD to *B*
  - 66 bytes: 64 pixels + 2 blacked-out pixels
- ComputeBias state computes bias for that row and stores in variable *Bias*
- FixBias state iterates over same row subtracting *Bias* from each element
- NextRow transitions to GetRow for repeat of process on next row or to Idle state when all 64 rows completed

FSMD description of CCDPP



Connecting SOC Components

- Memory-mapped
  - All single-purpose processors and RAM are connected to 8051's memory bus
- Read
  - Processor places address on 16-bit address bus
  - Asserts read control signal for 1 cycle
  - Reads data from 8-bit data bus 1 cycle later
  - Device (RAM or SPP) detects asserted read control signal
  - Checks address
  - Places and holds requested data on data bus for 1 cycle
- Write
  - Processor places address and data on address and data bus
  - Asserts write control signal for 1 clock cycle
  - Device (RAM or SPP) detects asserted write control signal
  - Checks address bus
  - Reads and stores data from data bus

Software

- System-level model provides majority of code
  - Module hierarchy, procedure names, and main program unchanged
- Code for UART and CCDPP modules must be redesigned
  - Simply replace with memory assignments
    - *xdata* used to load/store variables over external memory bus
    - *_at_* specifies memory address to store these variables
    - Byte sent to *U_TX_REG* by processor will invoke UART
    - *U_STAT_REG* used by UART to indicate its ready for next byte
      - UART may be much slower than processor
  - Similar modification for CCDPP code
- All other modules untouched

Original code from system-level model

```
#include <stdio.h>
static FILE *outputFileHandle;
void UartInitialize(const char *outputFileName) {
    outputFileHandle = fopen(outputFileName, "w");
}
void UartSend(char d) {
    fprintf(outputFileHandle, "%i\n", (int)d);
}
```

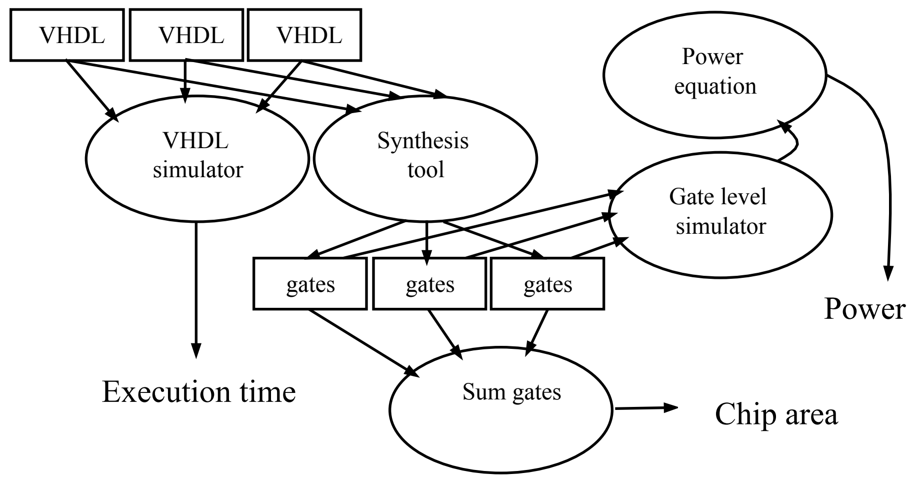
Rewritten UART module

```
static unsigned char xdata U_TX_REG _at_ 65535;
static unsigned char xdata U_STAT_REG _at_ 65534;
void UARTInitialize(void) {}
void UARTSend(unsigned char d) {
    while( U_STAT_REG == 1 ) {
        /* busy wait */
    }
    U_TX_REG = d;
}
```

Analysis

- Entire SOC tested on VHDL simulator
  - Interprets VHDL descriptions and functionally simulates execution of system
    - Recall program code translated to VHDL description of ROM
  - Tests for correct functionality
  - Measures clock cycles to process one image (performance)
- Gate-level description obtained through synthesis
  - Synthesis tool like compiler for SPPs
  - Simulate gate-level models to obtain data for power analysis
    - Number of times gates switch from 1 to 0 or 0 to 1
  - Count number of gates for chip area

Obtaining design metrics of interest



Implementation 2 (cont.)

- Analysis of implementation 2
  - Total execution time for processing one image:
    - 9.1 seconds
  - Power consumption:
    - 0.033 watt
  - Energy consumption:
    - 0.30 joule (9.1 s x 0.033 watt)
  - Total chip area:
    - 98,000 gates

Implementation 3

- 9.1 seconds still doesn't meet performance constraint of 1 second
- DCT operation prime candidate for improvement
  - Execution of implementation 2 shows microprocessor spends most cycles here
  - Could design custom hardware like we did for CCDPP
    - More complex so more design effort
  - Instead, will speed up DCT functionality by modifying behavior

DCT Floating-Point Cost

- Floating-point cost
  - DCT uses ~260 floating-point operations per pixel transformation
  - 4096 (64 x 64) pixels per image
  - 1 million floating-point operations per image
  - No floating-point support with Intel 8051
    - Compiler must emulate
      - Generates procedures for each floating-point operation
        - mult, add
      - Each procedure uses tens of integer operations
  - Thus, > 10 million integer operations per image
  - Procedures increase code size
- Fixed-point arithmetic can improve on this

Fixed-Point Arithmetic

- Integer used to represent a real number
  - Constant number of integer's bits represents fractional portion of real number
    - More bits, more accurate the representation
  - Remaining bits represent portion of real number before decimal point
- Translating a real constant to a fixed-point representation
  - Multiply real value by 2 ^ (# of bits used for fractional part)
  - Round to nearest integer
  - E.g., represent 3.14 as 8-bit integer with 4 bits for fraction
    - 2^4 = 16
    - 3.14 x 16 = 50.24 ≈ 50 = 00110010
    - 16 (2^4) possible values for fraction, each represents 0.0625 (1/16)
    - Last 4 bits (0010) = 2
    - 2 x 0.0625 = 0.125
    - 3(0011) + 0.125 = 3.125 ≈ 3.14 (more bits for fraction would increase accuracy)

Fixed-Point Arithmetic Operations

- Addition
  - Simply add integer representations
  - E.g., 3.14 + 2.71 = 5.85
    - 3.14 $\rightarrow$ 50 = 00110010
    - 2.71 $\rightarrow$ 43 = 00101011
    - 50 + 43 = 93 = 01011101
    - 5(0101) + 13(1101) x 0.0625 = 5.8125 ≈ 5.85
- Multiply
  - Multiply integer representations
  - Shift result right by # of bits in fractional part
  - E.g., 3.14 * 2.71 = 8.5094
    - 50 * 43 = 2150 = 100001100110
    - >> 4 = 10000110
    - 8(1000) + 6(0110) x 0.0625 = 8.375 ≈ 8.5094
- Range of real values used limited by bit widths of possible resulting values

Fixed-Point CODEC

- COS_TABLE gives 8-bit fixed-point representation of cosine values
- 6 bits used for fractional portion
- Result of multiplications shifted right by 6

Code

- COS_TABLE gives 8-bit fixed-point representation of cosine values
- 6 bits used for fractional portion
- Result of multiplications shifted right by 6

```
static const char code COS_TABLE[8][8] = {
    {   64,   62,   59,   53,   45,   35,   24,   12 },
    {   64,   53,   24,  -12,  -45,  -62,  -59,  -35 },
    {   64,   35,  -24,  -62,  -45,   12,   59,   53 },
    {   64,   12,  -59,  -35,   45,   53,  -24,  -62 },
    {   64,  -12,  -59,   35,   45,  -53,  -24,   62 },
    {   64,  -35,  -24,   62,  -45,  -12,   59,  -53 },
    {   64,  -53,   24,   12,  -45,   62,  -59,   35 },
    {   64,  -62,   59,  -53,   45,  -35,   24,  -12 }
};
```

```
static const char ONE_OVER_SQRT_TWO = 5;
static short xdata inBuffer[8][8], outBuffer[8][8], idx;

void CodecInitialize(void) { idx = 0; }
```

```
void CodecPushPixel(short p) {
    if( idx == 64 ) idx = 0;
    inBuffer[idx / 8][idx % 8] = p << 6; idx++;
}
```

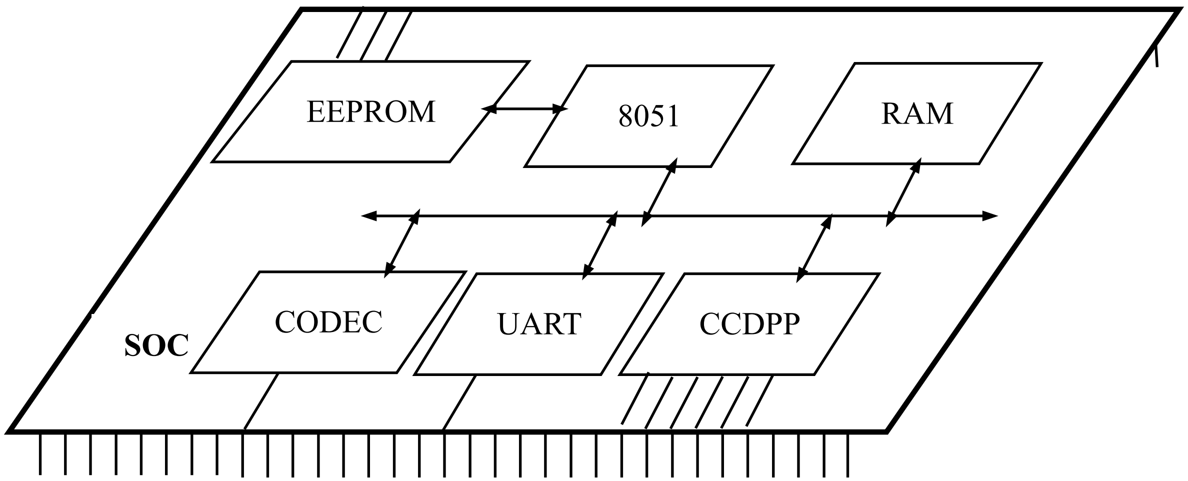
```
static unsigned char C(int h) { return h ? 64 : ONE_OVER_SQRT_TWO;}
static int F(int u, int v, short img[8][8]) {
    long s[8], r = 0;
    unsigned char x, j;
    for(x=0; x<8; x++) {
        s[x] = 0;
        for(j=0; j<8; j++)
            s[x] += (img[x][j] * COS_TABLE[j][v] ) >> 6;
    }
    for(x=0; x<8; x++) r += (s[x] * COS_TABLE[x][u]) >> 6;
    return (short)((((r * (((16*C(u)) >> 6) *C(v)) >> 6)) >> 6) >> 6);
}
```

```
void CodecDoFdct(void) {
    unsigned short x, y;
    for(x=0; x<8; x++)
        for(y=0; y<8; y++)
            outBuffer[x][y] = F(x, y, inBuffer);
    idx = 0;
}
```

Implementation 3 (cont.)

- Analysis of implementation 3
  - Use same analysis techniques as implementation 2
  - Total execution time for processing one image:
    - 1.5 seconds
  - Power consumption:
    - 0.033 watt (same as 2)
  - Energy consumption:
    - 0.050 joule (1.5 s x 0.033 watt)
    - Battery life 6x longer!!
  - Total chip area:
    - 90,000 gates
    - 8,000 less gates (less memory needed for code)

Implementation 4



- Performance close but not good enough
- Must resort to implementing CODEC in hardware
  - Single-purpose processor to perform DCT on 8 x 8 block

CODEC Design

- 4 memory mapped registers
  - *C_DATAI_REG/C_DATAO_REG* used to push/pop 8 x 8 block into and out of CODEC
  - *C_CMND_REG* used to command CODEC
    - Writing 1 to this register invokes CODEC
  - *C_STAT_REG* indicates CODEC done and ready for next block
    - Polled in software
- Direct translation of C code to VHDL for actual hardware implementation
  - Fixed-point version used
- CODEC module in software changed similar to UART/CCDPP

Code

Rewritten CODEC software

```
static unsigned char xdata C_STAT_REG _at_ 65527;
static unsigned char xdata C_CMND_REG _at_ 65528;
static unsigned char xdata C_DATAI_REG _at_ 65529;
static unsigned char xdata C_DATAO_REG _at_ 65530;
void CodecInitialize(void) {}
void CodecPushPixel(short p) { C_DATAO_REG =
(char)p; }
short CodecPopPixel(void) {
    return ((C_DATAI_REG << 8) | C_DATAI_REG);
}
void CodecDoFdct(void) {
    C_CMND_REG = 1;
    while( C_STAT_REG == 1 ) { /* busy wait */ }
}
```

Implementation 4 (cont.)

- Analysis of implementation 4
  - Total execution time for processing one image:
    - 0.099 seconds (well under 1 sec)
  - Power consumption:
    - 0.040 watt
    - Increase over 2 and 3 because SOC has another processor
  - Energy consumption:
    - 0.00040 joule (0.099 s x 0.040 watt)
    - Battery life 12x longer than previous implementation!!
  - Total chip area:
    - 128,000 gates
    - Significant increase over previous implementations

Summary of Implementations

- Implementation 3
  - Close in performance
  - Cheaper
  - Less time to build
- Implementation 4
  - Great performance and energy consumption
  - More expensive and may miss time-to-market window
    - If DCT designed ourselves then increased NRE cost and time-to-market
    - If existing DCT purchased then increased IC cost
- Which is better?

	Implementation 2	Implementation 3	Implementation 4
Performance (second)	9.1	1.5	0.099
Power (watt)	0.033	0.033	0.040
Size (gate)	98,000	90,000	128,000
Energy (joule)	0.30	0.050	0.0040

Summary

- Digital camera example
  – Specifications in English and executable language
  – Design metrics: performance, power and area
- Several implementations
  – Microcontroller: too slow
  – Microcontroller and coprocessor: better, but still too slow
  – Fixed-point arithmetic: almost fast enough
  – Additional coprocessor for compression: fast enough, but expensive and hard to design
  – Tradeoffs between hw/sw – one of the main lessons of this course!

Final issues

- Come by my office hours (right after class)
- Any questions or concerns?