E155 Final Project Status Report: μMudd Mark V Debugging and Lab 6 Revision

Christopher Ferrarin and Kaveh Pezeshki
28 November 2018

1 Completed Deliverables Status

Below is a summary of project deliverables and deliverable status:

Deliverable Category	Deliverable Name	Deliverable Status
Identifying blocking μMudd Bugs	Identifying MCU programming failure	Complete
Revising μMudd to allow MCU functionality	Hardware modification of pre-existing PCBs	Complete
	New JTAG cable	Complete
	Modified schematic and layout	In progress
	Completed μMudd respin	Not started
Reworking Lab 6	Rewrite EasyPIO.h with SAM4S support	Complete
	Integrate MCP3002, photodiode, and BlueSMiRF	Complete
	Formally write up lab for student readability	Not started
Testing other labs	Lab 4	Not started
	Lab 5	Not started
	Lab 7	Not started

2 Deliverable Status: Revised μMudd

A major component of this final project is identifying errors in the PCB design that lead to a non-programmable MCU. We have identified two errors which when solved allowed MCU programming

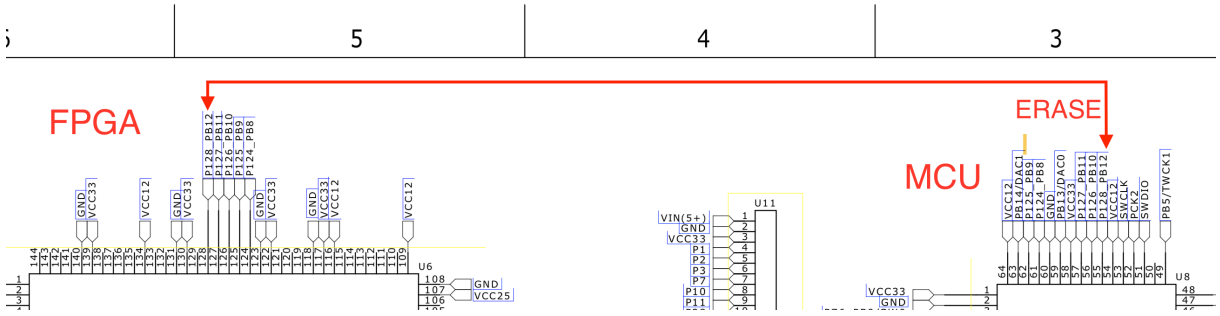
2.1 Schematic Errors

2.1.1 MCU ERASE Pin

The largest problem with the current μMudd design lies in the MCU ERASE pin, which reinitializes the onboard flash and resets the processor. The ERASE pin can also serve as general-purpose I/O after configuration. [1]

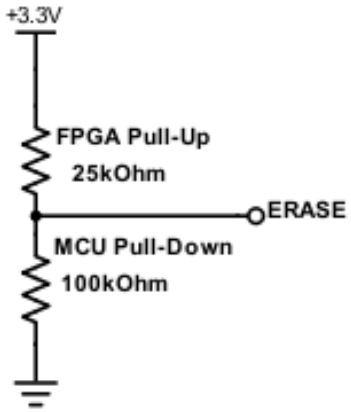
[1] SAM4S Series Datasheet p37

On boot, ERASE must be held low to prevent flash erase and reinitialization of the processor. On the current μMudd, ERASE was tied to a general I/O pin on the Cyclone IV FPGA. The connection can be seen in the following schematic:



The marked connection ties ERASE on the MCU to pin 128 on the FPGA

The ERASE pin contains a 100kΩ pull-down resistor[2].An unconfigured Cyclone IV I/O pin contains a 25kΩ pull-up resistor [3]. This creates a voltage divider circuit as shown below:



This provides a predicted voltage of 2.64V on the MCU ERASE pin, close to the 2.86V we observed. This is a high logic level which prevented FPGA programming.

2.1.2 MCU Power Supply

The MCU requires a 3.3V and 1.2V power supply. It can be powered via one 3.3V supply, and use an internal regulator to generate 1.2V, or it can be powered with an external 3.3V and a 1.2V supply. The dual-regulator design of the current board can introduce startup issues if timing is not correct.

We believe that these potential timing errors can cause system instability, as we observed an unresponsive MCU after startup that could only be solved with a full erase and reset.

[2]SAM4S Series Datasheet p37
[3]Cyclone IV Device Handbook p6-3

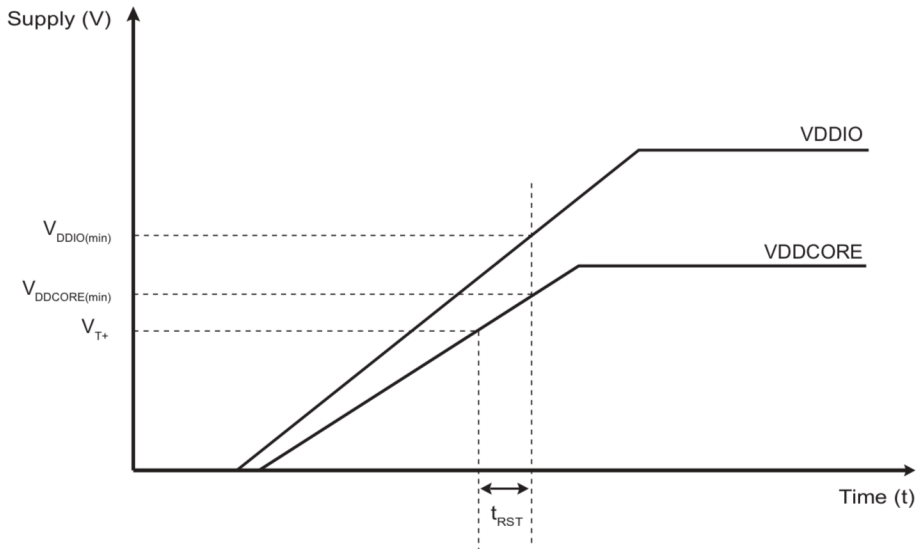


Figure 1: Timing requirements for the 1.2V (VDDCORE) and 3.3V (VDDIO) supplies, taken from the SAM4S Series Datasheet p27

2.1.3 JTAG connector pinout

The MCU JTAG connector was incorrectly wired on the current μMudd. This led to wiring errors in the I^2C connection

2.2 Schematic and Layout Changes

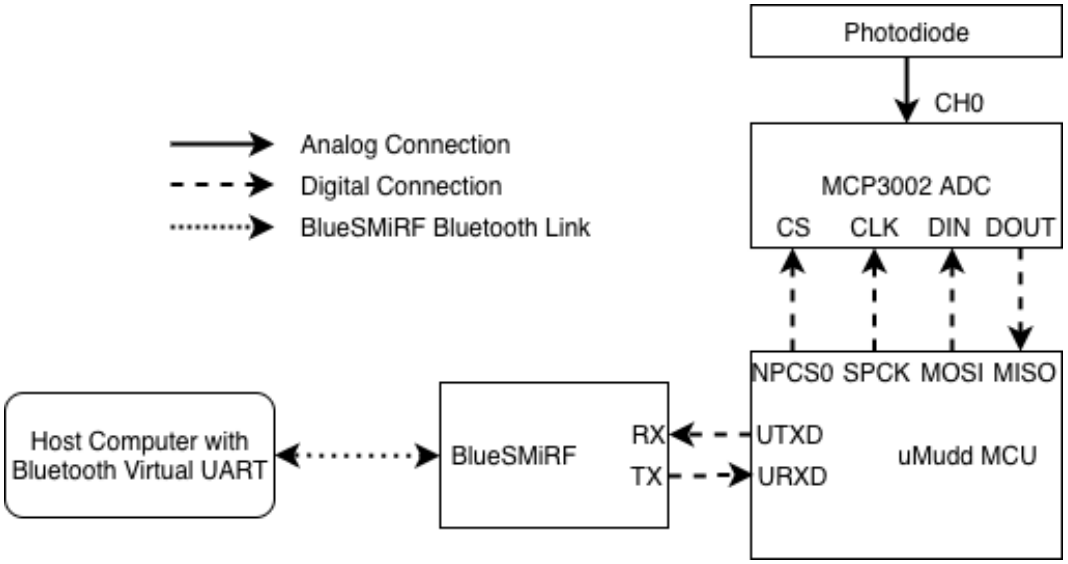
We have implemented a set of changes to the schematic to solve the problems noted above and to improve the PCB, but have not yet propagated these changes to the layout. These include:

1. Moving ERASE control to the MCU RESET pushbutton. RESET will be accessible through JTAG
2. Powering the 1.2V MCU VDDCORE with the onboard regulator, and adding necessary passives
3. Correcting JTAG and I^2C wiring errors
4. Replacing 0.1" pitch JTAG connectors with 0.05" pitch SWD connectors. This adds compatibility with J-Link EDU Mini programmers
5. Adding 0.1" jumpers on critical MCU pin connections to assist debugging

3 Deliverable Status: Reworking Lab 6 and EasyPIO.h

3.1 Reworking Lab 6

The proposed Lab 6 architecture is shown in detail below:



To maintain a low lab cost we exchanged one BlueSMiRF and the serial display for a computer with an integrated Bluetooth module. We added a MCP3002 ADC to retain the datasheet interpretation component of the lab.

We have successfully demonstrated Bluetooth communication between two computers, using a Bus Pirate as a USB to UART converter. We have also successfully demonstrated voltage measurement with the MCP3002 through a SAM3S/SAM4S SPI peripheral.

In reworking Lab 6, we still need to write an updated version of the lab manual and polish it so that it easily readable by a future student. We also seek to implement any recommendations we receive in our presentation for ways to improve this lab, as the replacement of the web server with a Bluetooth link may render the lab simpler than we'd like.

3.2 Targeting easyPIO.h for the SAM4S

We have created an I/O header file, easySamIO.h, which provides Arduino-style access to the GPIO, timer, UART, and SPI peripherals on the SAM4S. In the style of easyPIO.h, we provide only the configuration and functionality necessary to complete labs. We aim to provide adequate inline documentation for students to add functionality as necessary. This documentation includes functional descriptions of memory access, references to the datasheet, and brief descriptions of other peripheral features.

In total, we have completed implementation of the PIO, timer, SPI, and UART peripherals, but still need to provide access to additional ports of the PIO peripheral, additional channels of the timer peripheral, and additional clock routing for the FPGA on the μMudd.

We believe a thorough and well-documented EasySamIO.h will be more useful for lab 4,5, and 7 bringup than testing labs 4,5, and 7 as discussed in the lab manual.

4 Appendix 1: Updated Schematic

5 Appendix 2: C Code

5.1 EasySamIO.h

5.2 lab6.c

```
//TIMER COUNTER 0

#define REG_TC0_CCR  (*(  volatile unsigned int *)0x40010000U) /**< \brief (TC0) Channel Control Register */
#define REG_TC0_CMR  (*(  volatile unsigned int *)0x40010004U) /**< \brief (TC0) Channel Mode Register */
#define REG_TC0_SMMR (*(  volatile unsigned int *)0x40010008U) /**< \brief (TC0) Stepper Motor Mode Register */
#define REG_TC0_CV   (*(  volatile unsigned int *)0x40010010U) /**< \brief (TC0) Counter Value Register */
#define REG_TC0_RA   (*(  volatile unsigned int *)0x40010014U) /**< \brief (TC0) Register A Register */
#define REG_TC0_RB   (*(  volatile unsigned int *)0x40010018U) /**< \brief (TC0) Register B Register */
#define REG_TC0_RC   (*(  volatile unsigned int *)0x4001001CU) /**< \brief (TC0) Register C Register */
#define REG_TC0_SR   (*(  volatile unsigned int *)0x40010020U) /**< \brief (TC0) Status Register */
#define REG_TC0_IER  (*(  volatile unsigned int *)0x40010024U) /**< \brief (TC0) Interrupt Enable Register */
#define REG_TC0_IDR  (*(  volatile unsigned int *)0x40010028U) /**< \brief (TC0) Interrupt Disable Register */
#define REG_TC0_IMR  (*(  volatile unsigned int *)0x4001002CU) /**< \brief (TC0) Interrupt Mask Register */

#define REG_TC_BCR   (*(  volatile unsigned int *)0x400100C0U) /**< \brief (TC) Block Control Register */
#define REG_TC_BMR   (*(  volatile unsigned int *)0x400100C4U) /**< \brief (TC) Block Mode Register */
#define REG_TC_QIER  (*(  volatile unsigned int *)0x400100C8U) /**< \brief (TC) QDEC Interrupt Enable Register */
#define REG_TC_QIDR  (*(  volatile unsigned int *)0x400100CCU) /**< \brief (TC) QDEC Interrupt Disable Register */
#define REG_TC_QIMR  (*(  volatile unsigned int *)0x400100D0U) /**< \brief (TC) QDEC Interrupt Mask Register */
#define REG_TC_QISR  (*(  volatile unsigned int *)0x400100D4U) /**< \brief (TC) QDEC Interrupt Status Register */
#define REG_TC_FMR   (*(  volatile unsigned int *)0x400100D8U) /**< \brief (TC) Fault Mode Register */
#define REG_TC_WPMR  (*(  volatile unsigned int *)0x400100E4U) /**< \brief (TC) Write Protect Mode Register */

#define TC_WPMR_WPKEY_PASSWD (0x54494Du << 8) /**< \brief (TC_WPMR) Writing any other value in this field aborts the write
operation of the WPEN bit. Always reads as 0. */

void samInit() {
//Many peripherals on the SAM4S are write protected: unless the correct password is written in a peripheral memory address,
write access to peripheral control registers is disabled. This is done for security reasons, but is not necessary in this header file.
In the first part of this function, we enable write access to the PMC, PIO, SPI, and UART by writing a password into the peripheral's
Write Protect Mode Register (WPMR)


//disabling PMC write protection (Password: "PMC")
REG_PMC_WPMR = PMC_WPMR_WPKEY_PASSWD;
//disabling PIO write protection (Password: "PIO")
REG_PIOA_WPMR = PIO_WPMR_WPKEY_PASSWD;
//disabling SPI write protection (Password: "SPI")
REG_SPI_WPMR = SPI_WPMR_WPKEY_PASSWD;
//There is no UART write protection

//disabling timer write protection (Password: "TIM")
REG_TC_WPMR = TC_WPMR_WPKEY_PASSWD;

//We next need to supply a clock to these peripherals. For a given peripheral, clock is enabled by writing a 1 into a specific bit
of the PMC Peripheral Clock Enable Register (PCER). There are two registers for the 34 peripherals. Peripheral - bit number mapping
is given in p36: Peripheral Identifiers.

//Activating clocks for UART 0 (PID 8), PIO A (PID 11), SPI (PID 21), TC0 (Timer/Counter CH0) (PID 23)

REG_PMC_PCER0 |= (1<<8);
REG_PMC_PCER0 |= (1<<11);
REG_PMC_PCER0 |= (1<<21);
REG_PMC_PCER0 |= (1<<23);
}


/*----------------------------
--------PIO METHODS-------------
----------------------------*/

void pinMode(int pin, int function) {
REG_PIOA_IFDR  |=  (1 << pin);
REG_PIOA_IER   &= ~(1 << pin);
REG_PIOA_IDR   |=  (1 << pin);
REG_PIOA_MDER  &= ~(1 << pin);
REG_PIOA_MDDR  |=  (1 << pin);
REG_PIOA_PUER  &= ~(1 << pin);
REG_PIOA_PUDR  |=  (1 << pin);
REG_PIOA_PPDER &= ~(1 << pin);
REG_PIOA_PPDDR |=  (1 << pin);
REG_PIOA_OWER  &= ~(1 << pin);
REG_PIOA_OWDR  |=  (1 << pin);

if (function == INPUT || function == OUTPUT) {
REG_PIOA_PER |=  (1 << pin);
REG_PIOA_PDR &= ~(1 << pin);
} else {
REG_PIOA_PER &= ~(1 << pin);
REG_PIOA_PDR |=  (1 << pin);
}

switch (function) {
case INPUT:
REG_PIOA_OER &= ~(1 << pin);
REG_PIOA_ODR |=  (1 << pin);
break;
case OUTPUT:
REG_PIOA_OER |=  (1 << pin);
REG_PIOA_ODR &= ~(1 << pin);
```

```
break;
case A:
REG_PIOA_ABCDSR1 &= ~(1 << pin);
REG_PIOA_ABCDSR2 &= ~(1 << pin);
break;
case B:
REG_PIOA_ABCDSR1 |=  (1 << pin);
REG_PIOA_ABCDSR2 &= ~(1 << pin);
break;
case C:
REG_PIOA_ABCDSR1 &= ~(1 << pin);
REG_PIOA_ABCDSR2 |=  (1 << pin);
break;
case D:
REG_PIOA_ABCDSR1 |= (1 << pin);
REG_PIOA_ABCDSR2 |= (1 << pin);
break;
// Otherwise, do nothing
}
}

void digitalWrite(int pin, int val) {
if (val == HIGH) {
REG_PIOA_SODR |= (1 << pin);
} else {
REG_PIOA_CODR |= (1 << pin);
}
}

int digitalRead(int pin) {
return (REG_PIOA_PDSR >> pin) & 1;
}

void toggle(int pin) {
int currentVal = digitalRead(pin);
digitalWrite(pin, !currentVal);
}


/*-----------------------------
--------SPI METHODS-------------
-----------------------------*/

void spiInit(char clkdivide, int cpol, int ncpha) {
/*Initializes the SPI interface for Chip Select line 0

clkdivide (0x01 to 0xFF). The SPI clk will be the master clock / clkdivide
cpol: clock polarity (0: inactive state is logic level 0, 1: inactive state is logic level 1)
ncpha: clock phase (0: data changed on leading edge of clk and captured on next edge, 1: data captured on leading edge of clk and changed on next edge)
Please see p585-p586 for cpol/ncpha timing diagrams

This implements only: (p601/p610)
1) SPI Master Mode
2) Fixed Peripheral Select
3) Mode Fault Detection Enabled
4) Local Loopback Disabled
5) 8 Bits Per Transfer
Please read the SPI User Interface section of the datasheet for more advanced configuration features
*/

//Initially assigning SPI pins (PA11-PA14) to peripheral A (SPI). Pin mapping given in p38-p39
pinMode(11, A);
pinMode(12, A);
pinMode(13, A);
pinMode(14, A);

//next setting the SPI control register (p600). Set to 1 to enable SPI
REG_SPI_CR = 1;

//next setting the SPI mode register (p601) with the following:
//master mode
//fixed peripheral select
//chip select lines directly connected to peripheral device
//mode fault detection enabled
//WDRBT disabled
//LLB disabled
//PCS = 0000 (Peripheral 0 selected), means NPCS[3:0] = 1110
REG_SPI_MR = 1;

//next setting the chip select register for peripheral 0 (p610)
//ignoring delays

//REG_SPI_CSR = (cpol<<0) | (ncpha<<1) | (clkdivide << 16);
REG_SPI_CSR = 0x0000FF00;
}

char spiSendReceive(char send) {
//Sends one byte over SPI and returns the received character
REG_SPI_TDR = send;
//Wait until Receive Data Register Full (RDRF, bit 0) and TXEMPTY (bit )
while ( !( (REG_SPI_SR & 1) & ((REG_SPI_SR >> 9) & 1) ) );
//After these status bits have gone high, the transaction is complete
```

```
return REG_SPI_RDR;
}

short spiSendReceive16(short send) {
//sends one 16-bit short over SPI and returns the received short
short rec;
rec = spiSendReceive((send & 0xFF00) >> 8); // send data MSB first
rec = (rec << 8) | spiSendReceive(send & 0xFF);
return rec;
}


/*----------------------------
--------TIMER METHODS-------------
----------------------------*/

void timerInit() {
// Enable clock to channel
REG_TC0_CCR    &= ~(1 << 1);
REG_TC0_CCR    |= 1;

REG_TC0_CMR    |= (1 << 15); // WAVE = 1 (waveform mode)
REG_TC0_CMR    |= (1 << 14); // WAVESEL = 2 (UP_RC waveform)
}

// Works until 2097 ms
// Resolution of 32 us
void delay_ms(int num) {
REG_TC0_CMR |= 0x3; //0b11; // Using TIMER_CLOCK4
REG_TC0_RC   = (unsigned long) (TIMER_CLOCK4 * (((float) num) / 1000)); // Compare value
REG_TC0_CCR |= (1 << 2); // Reset counter
while (!((REG_TC0_SR >> 4) & 1)); // Delay until match
}


/*----------------------------
--------UART METHODS-------------
----------------------------*/

/* Initialize UART. Note that pin PA9 is used as receive and pin PA10
* is used as transmit. samInit() must be called first.
* parity:
*  0: Even
*  1: Odd
*  2: Space (forced to 0)
*  3: Mark (forced to 1)
*  4: No (no parity)
* Baud Rate =  MCK/(16*CD), CD is an unsigned short
*/
void uartInit(int parity, int CD) {
pinMode(9, A);   // Set URXD0 pin mode
pinMode(10, A);   // Set ITXD0 pin mode

REG_UART_CR   |= 1 << 6; // Enable transmitter
REG_UART_CR   |= 1 << 4; // Enable receiver

// Parity
//REG_UART_MR   |= parity << 9;
//REG_UART_MR   &= ~(parity << 9);
REG_UART_MR   |= 1 << 11;
REG_UART_MR   &= ~((0x3) << 9);

REG_UART_BRGR = CD;   // Set baud rate divider
}

// Transmits a character (1 byte) over UART
void uartTx(char data) {
while (!((REG_UART_SR >> 1) & 1)); // Wait until previous data has been transmitted
REG_UART_THR = data; // Write data into holding register for transmit
}

// Returns a character (1 byte) received over UART
char uartRx() {
while(!((REG_UART_SR) & 1)); // Wait until data has been received
return (char) REG_UART_RHR; // Return received data in holding register

}
```

5.2 lab6.c

```
#include <easySamIO.h>
#include <stdio.h>

int main() {
samInit();
uartInit(4, 2);
timerInit();
spiInit(255, 0, 0);
pinMode(17, OUTPUT);

unsigned short received = 0;
float voltage;
char strTransmit[5];
char introTransmit[10] = "Voltage: ";

while (1) {
digitalWrite(17, HIGH);

received = spiSendReceive16(0x6000);
received &= 0x03FF; // Removing 6 MSbs
voltage = 3.3 * ((float) received / 1023.0);

snprintf(strTransmit, 5, "%f", voltage);

for (int charNum = 0; charNum < 9; charNum++) {
uartTx(introTransmit[charNum]);
}

for (int charNum = 0; charNum < 5; charNum++) {
uartTx(strTransmit[charNum]);
}

uartTx(0x0A);

digitalWrite(17, LOW);

delay_ms(1000);
}
return 1;
}
```