BeatKeyper

Music License Enforcement System

Team Name: **No One Speaks Verilog, It's a Dead Language**

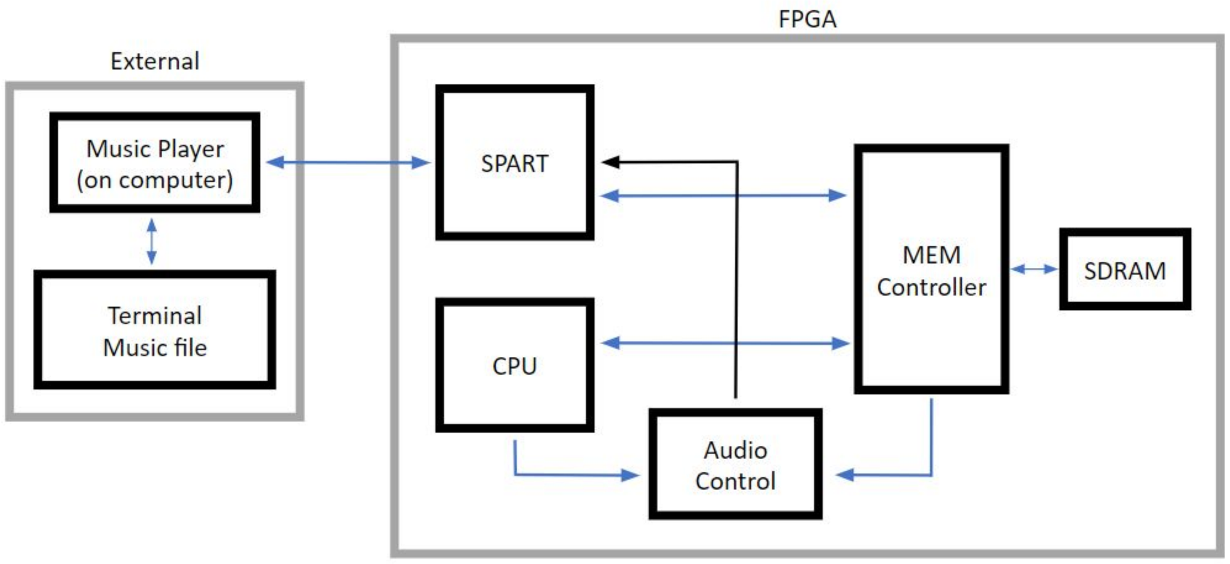
Tristen Hallock, Sadeq Hashemi Nejad, Andrew Pelto, Alec Sivit, Alex Valois

Final Report

University of Wisconsin-Madison
Fall 2018

1. Top Level

1.1 Block Diagram



1.2 System Overview

Our project can be best described as a music license enforcement system, somewhat similar to the offline mode of Spotify's music streaming service. The concept is a music gateway that keeps individuals who have not paid for the song licensing from listening to the song. The FPGA takes in a requested song from the user's computer (in data packets) and checks whether the user holds a license for the respective song. If the user has the necessary music license then the song will be decrypted by the FPGA and sent back through the serial data cable to be played; otherwise the user will hear scrambled song data. While the user has all the songs locally, the data is encrypted by the vendor so without the proper licensing key it's be impossible to listen to or distribute the song illegally.

As we brainstormed ideas for the main project, encryption was a very popular topic with numerous proposals incorporating data encryption in some way. We decided on music encryption because it is a tangible way to see how encryption affects data. We can hear the difference our encryption makes to the various songs we include. The license enforcement concept was directly inspired by the Sirius Satellite radio service, which only allows individuals who have the correct receiver and a subscription to the service to access the stations. The satellites transmit scrambled signals which must be unscrambled by the receiver module in order to be played properly. Our idea stems from this service but aligns more with the offline mode of Spotify which allows users to download songs so they can be listened to without internet access, but only as long as the user continues to pay for the service each month.

2. CPU

2.1 CPU Architecture

2.1.1 Registers

Our CPU has 32 32-bit registers labeled r0-r31 and 32 128-bit registers labeled e0-e31.
32-bit registers are for standard operations, while 128-bit registers are for encryption operations.
There are two special instructions for transferring values between 32-bit and 128-bit registers, but each width of register has its own separate instructions to keep things straight in assembly code.

In addition, there is one 1-bit register for the condition code Z (zero) which can not be written to directly, but is rather updated by the result of arithmetic instructions.

2.1.2 ISA Summary

Unless specified, all operations use the 32-bit registers

Opcode [31:26]	Instruction	Description	Used in Demo
0000_00	NOP	Does nothing	Yes
0000_01	ADD	Adds two registers	Yes
0000_10	SUB	Subtracts two registers	Yes
0000_11	NOT	Inverts a register	No
0001_00	AND	ANDs two registers	No
0001_01	OR	ORs two registers	No
0001_10	XOR	XORs two registers	No
0001_11	SLL	Logical Left shifts a register	No
0010_00	SRA	Arithmetic Right shifts a register	No
0010_01	SRL	Logical Left shifts a register	No
0010_10	LI	Loads a register with an immediate	Yes
0010_11	LD	Loads a value from memory to a register	Yes
0011_00	ST	Stores a value from a register to memory	Yes
0011_01	JI	PC jumps to an immediate	No
0011_10	JR	PC jumps to a register value	No

0011_11	BEQ	Branch if equal (NZ = x1)	Yes
0100_00	BNE	Branch if not equal (NZ = x0)	Yes
0100_01	FTO	Write one 128-bit register to four 32-bit registers	Yes
0100_10	OTF	Write four 32-bit registers to on 128-bit register	Yes
0100_11	DEC	Does one step of AES decryption on a 128-bit register	Yes
0101_00	DECF	Does the final step of AES decryption on a 128-bit register	Yes
0101_01	XORE	Bitwise XOR operation on two 128-bit registers	Yes
0101_10	FLC	Checks flags from the SPART	Yes
0101_11	FLS	Sets flag to the SPART or Audio	Yes
1111_11	HLT	Stalls the CPU forever	Yes

2.1.3 Condition Codes

Our ISA doesn't use conditional execution.

2.1.4 Addressing Modes

Our ISA uses two forms of instruction addressing: PC Relative, where the address is determined from the PC and an offset, and Register Direct, where the address is loaded directly from a register value.

Our ISA uses two forms of data addressing: Base Plus Offset, where the address is determined from a register base plus an immediate offset, and Register Direct, where the address is loaded directly from a register value.

2.1.5 Instruction Descriptions

NOP

Syntax:
NOP

Pseudo Code:
nop

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	unused																									

Usage and Example:
This instruction simply tells the CPU to do nothing. Use it when you want to stall.

ADD

Syntax:
ADD <rd>, <rs>, <rt>

Pseudo Code:
rd = rs + rt

Flags updated:
N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	rd					rs					rt					unused										

Usage and Example:
This is instruction performs signed addition.
ADD r0, r9, r10 -- Adds the values in r9 and r10 and stores the result into r0

SUB

Syntax:
SUB <rd>, <rs>, <rt>

Pseudo Code:
rd = rs - rt

Flags updated:
N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	rd					rs					rt					unused										

Usage and Example:
This is instruction performs signed subtraction.
SUB r0, r9, r10 -- Subtracts the value in r10 from the value in r9 and stores the result into r0

NOT

Syntax:
NOT <rd>, <rs>

Pseudo Code:
rd = ~rs

Flags updated:
N

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	rd					rs					unused															

Usage and Example:
This is instruction performs logical negation.
NOT r2, r21 -- Inverts the value in r21 and stores the result in r2

AND

Syntax:
AND <rd>, <rs>, <rt>

Pseudo Code:
rd = rs & rt

Flags updated:
N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	rd					rs					rt					unused										

Usage and Example:
This is instruction performs a logical bitwise AND.
AND r0, r9, r10 -- Bitwise ANDs the values in r9 and r10 and stores the result into r0

OR

Syntax:
OR <rd>, <rs>, <rt>

Pseudo Code:
rd = rs | rt

Flags updated:

N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	rd					rs					rt					unused										

Usage and Example:
This is instruction performs a logical bitwise OR.
OR r0, r9, r10 -- Bitwise ORs the values in r9 and r10 and stores the result into r0

XOR

Syntax:
XOR <rd>, <rs>, <rt>

Pseudo Code:
rd = rs ^ rt

Flags updated:
N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	rd					rs					rt					unused										

Usage and Example:
This is instruction performs a logical bitwise XOR.
XOR r0, r9, r10 -- Bitwise XORs the values in r9 and r10 and stores the result into r0

SLL

Syntax:
SLL <rd>, <rs>, #<shift>

Pseudo Code:
rd = rs << shift

Flags updated:
N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	rd					rs					shift					unused										

Usage and Example:
This is instruction performs logical left shift by the unsigned immediate specified as <shift>.
Shift is an unsigned immediate in the range from 0 to 32
SLL r1, r0, #16 -- Left shifts the value in r0 by 16, filling in 0s, and stores the result in r1

SRA

Syntax:
SRA <rd>, <rs>, #<shift>

Pseudo Code:
rd = rs >>> shift

Flags updated:
N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	rd					rs					shift					unused										

Usage and Example:
This is instruction performs arithmetic right shift by the unsigned immediate specified as <shift>.
Shift is an unsigned immediate in the range from 0 to 32
SRA r1, r0, #3 -- Right shifts the value in r0 by 3, sign extending, and stores the result in r1

SRL

Syntax:
SRL <rd>, <rs>, #<shift>

Pseudo Code:
rd = rs >> shift

Flags updated:
N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	rd					rs					shift					unused										

Usage and Example:
This is instruction performs logical right shift by the unsigned immediate specified as <shift>.
Shift is an unsigned immediate in the range from 0 to 32
SRL r1, r0, #30 -- Right shifts the value in r0 by 30, filling in 0s, and stores the result in r1

LI

Syntax:
LI <rd>, <p>, #<immediate>

Pseudo Code:

```
if (p)
        rd[31:16] = sext(immediate)
else
        rd[15:0] = sext(immediate)
```

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	rd					p	immediate																unused			

Usage and Example:
This instruction loads either the high or low half of a register from an immediate value
Immediate must be in the range from -32,768 to +32,767
LI r22, 1, #100 -- Loads the top 16-bits of r22 with the value 100

LD

Syntax:
LD <rd>, <rs>, #<offset>

Pseudo Code:

```
rd = mem[rs + sext(offset)]
```

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	rd					rs					offset															

Usage and Example:
This is instruction loads the value at mem[rs + offset] into rd.
Offset must be a signed immediate in the range from -32,768 to +32,767
LD r5, r15, #0 -- Loads the value at mem[r15] into r5

ST

Syntax:
ST <rd>, <rs>, #<offset>

Pseudo Code:
mem[rs + sext(offset)] = rd

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	rd					rs					offset															

Usage and Example:
This is instruction stores the value of rd into mem[rs + offset].
Offset must be a signed immediate in the range from -32,768 to +32,767
ST r5, r15, #5 -- Stores the value of r5 into mem[r15 + 5]

JI

Syntax:
JI #<immediate>

Pseudo Code:
r31 = pc + 1
pc = pc + sext(immediate)

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	immediate																									

Usage and Example:
This instruction loads the PC with the value of the PC plus an immediate and saves the PC value.

Immediate must be in the range from -225 to +225-1
JI #-25000 -- Loads the PC with its current value plus -25,000 and stores the old PC +1 into r31

JR

Syntax:
JR <rs>

Pseudo Code:
```
r31 = pc + 1
pc = pc + rs
```

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	rs					unused																				

Usage and Example:
This instruction loads the PC with the value in rs.
JR r30 -- Loads the PC with the value saved in r30 and stores the old PC + 1 in r31

BEQ

Syntax:
BEQ #<immediate>

Pseudo Code:
```
if (z == 1) pc = pc + sext(immediate)
```

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	immediate																									

Usage and Example:
This instruction loads the PC with its value plus the value of the immediate if Z = 1
Immediate must be in the range from -225 to +225-1
BEQ #10 -- Loads the PC with its current value plus 10 when Z = 1

BNE

Syntax:
BNE #<immediate>

Pseudo Code:
if (z == 0) pc = pc + sext(immediate)

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	immediate																									

Usage and Example:
This instruction loads the PC with its value plus the value of the immediate if Z = 0
Immediate must be in the range from -225 to +225-1
BNE #10 -- Loads the PC with its current value plus 10 when Z = 0

FTO

Syntax:
FTO <ed>, <r1>, <r2>, <r3>, <r4>

Pseudo Code:
ed = {r1, r2, r3, r4}

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	ed					r1					r2					r3					r4					unused

Usage and Example:
This instruction loads the 128-bit register ed with the values in the 32-bit registers r1-4.
FTO e0, r4, r5, r6, r7 -- Loads e0 with the values in r4, r5, r6, r7

OTF

Syntax:
OTF <ed>, <r1>, <r2>, <r3>, <r4>

Pseudo Code:
{r1, r2, r3, r4} = ed

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	ed					r1					r2					r3					r4					unused

Usage and Example:
This instruction loads the 32-bit registers r1-4 with the value in the 128-bit register ed.
OTF e0, r4, r5, r6, r7 -- Loads r4, r5, r6, r7 with the value in e0

DEC

Syntax:
DEC <ed>, <es>, <ek>

Pseudo Code:
ed = AES_dec_round(es, ek)

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	ed					es					ek					unused										

Usage and Example:
This is instruction performs one round of AES decryption on the value stored in es, using the value stored in ek as the key, and stores it into ed.
DEC e13, e5, e0 -- One round of decryption is performed on e5 using e0 as key, and stored in e13

DECF

Syntax:
DECF <ed>, <es>, <ek>

Pseudo Code:
ed = AES_dec_final_round(es, ek)

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	ed					es					ek					unused										

Usage and Example:
This is instruction performs the final round of AES decryption on the value stored in es, using the value stored in ek as the key, and stores it into ed.
DECF e13, e5, e0 -- The final round of decryption is performed on e5 using e0 as key, and stored in e13

XORE

Syntax:
XORE <ed>, <es>, <et>

Pseudo Code:
ed = es ^ et

Flags updated:
N, Z

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	ed					es					et					unused										

Usage and Example:
This is instruction performs a logical bitwise XOR on two 128-bit registers.
XORE e12, e11, e0 -- Bitwise XORs the values in e11 and e0 and stores the result into e12

FLC

Syntax:
FLC <rd>, <mask>

Pseudo Code:
rd = mask | cpu_status_flags

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0	1	0	1	1	0	rd	mask	unused

Usage and Example:
This is instruction checks the status flags from other hardware blocks. Mask is a 2-bit constant mask for which flags to check. Mask = 10 for recieved_data, 01 = stop_data.
FLC r12, 10 -- r12 is updated with the value of the SPART recieved_data flag.

FLS

Syntax:
FLS <mask>

Pseudo Code:
cpu_status_flags = mask

Flags updated:
none

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	mask		unused																							

Usage and Example:
This is instruction sets the status flags from other hardware blocks. Mask is a 2-bit constant mask for which flags to check. Mask = 10 for send_data, 01 = audio_rdy
FLC 01 -- CPU sets the audio_rdy flag

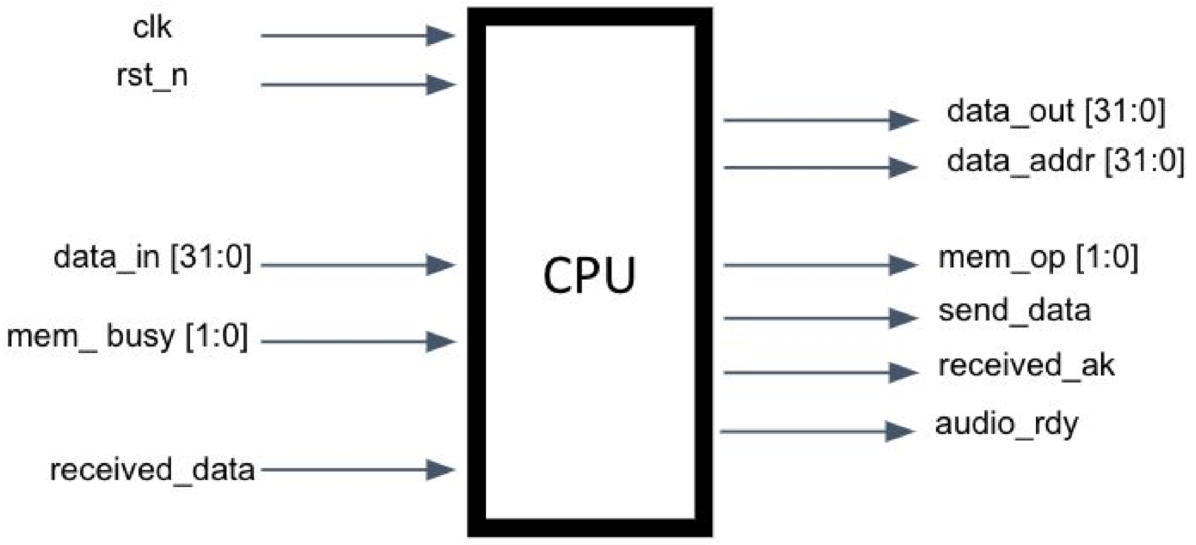
2.2 CPU Microarchitecture

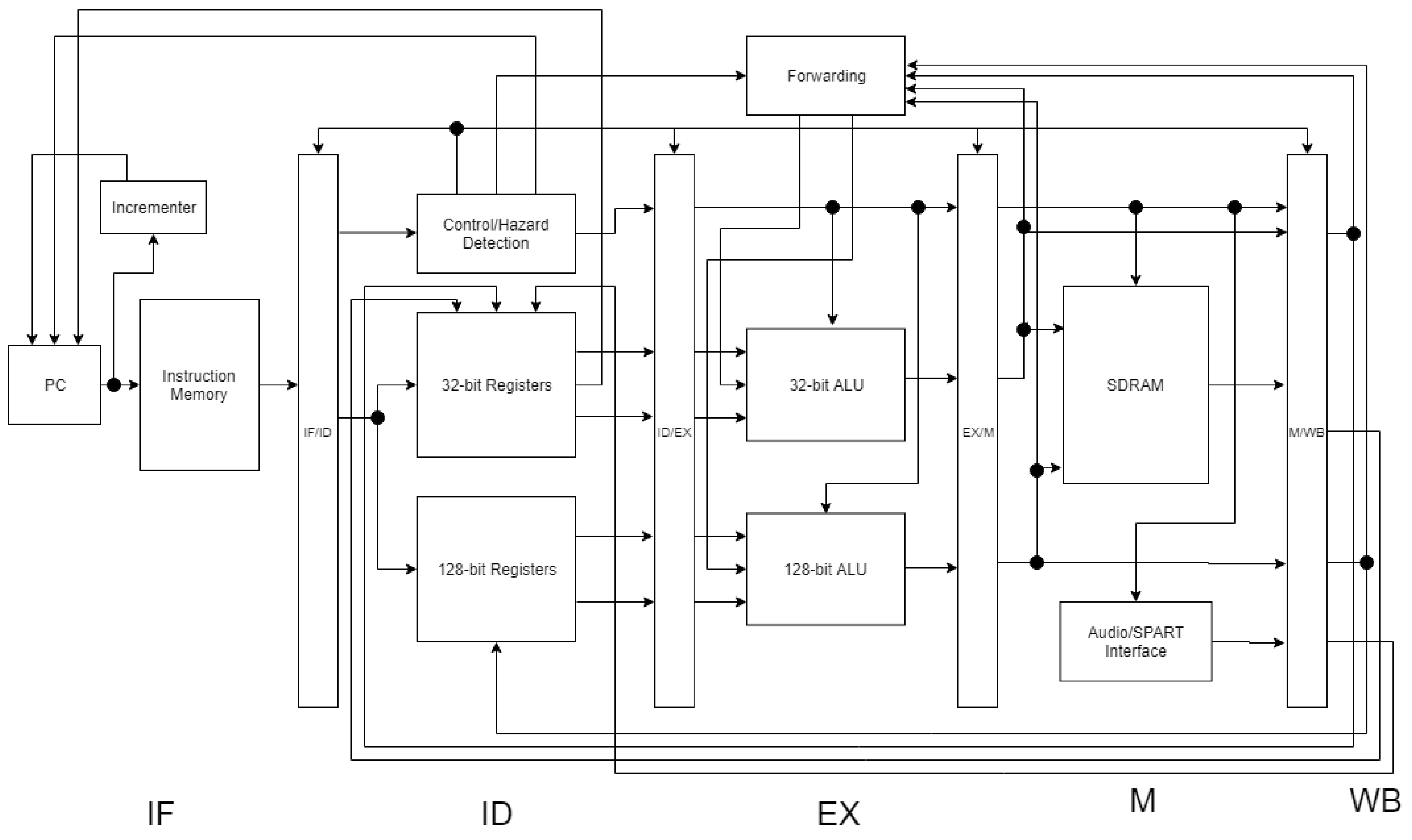
2.2.1 External Interface

The following table summarizes the table-level interface for our CPU in a similar manner to how our other hardware blocks were set up. All signals are 1-bit unless specified otherwise.

Signal Name	I/O	Source/Target Block	Description
clk	in		
rst_n	in		
data_in [31:0]	in	MEM	32-bit input data line to MEM

data_out [31:0]	out	MEM	32-bit output data line from MEM
data_addr [31:0]	out	MEM	32-bit Address for MEM
mem_op [1:0]	out	MEM	2-bit MEM control line: ● 00 = CPU not requesting memory ● 01 = CPU requesting read ● 10 = CPU requesting write
mem_busy [1:0]	in	MEM	2-bit MEM busy signal: ● 00 = memory not busy ● 01 = memory doing CPU task ● 10 = memory doing SPART task ● 11 = memory doing Audio task
send_data	out	SPART	Alert to SPART saying data is ready to be sent after using SPARTS instruction
recieved_data	in	SPART	Alert from SPART saying new data has been received, held high till ak'd by CPU
stop_data	in	SPART	Alert from SPART saying to stop playback, held high till ak'd by CPU
recieved_ak	out	SPART	Acknowledge signal to SPART after CPU checks recieved_data using FLC instruction
stop_ak	out	SPART	Acknowledge signal to SPART after CPU checks stop_data using FLC instruction
audio_rdy	out	Audio	Alert to Audio that data is ready to be played





2.2.2 Internal Diagram

Our CPU is set up as a traditional 5-stage pipelined Harvard architecture. Our 5 stages are:

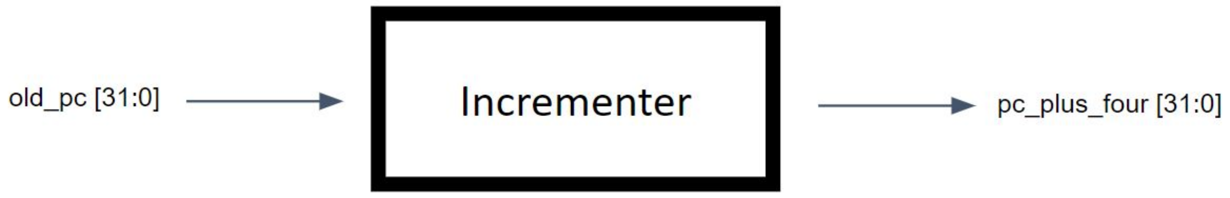
Instruction Fetch (IF) - fetching instructions from instruction memory and loading the PC
Instruction Decode (ID) - determining the relevant control signals and register values
Execution (EX) - dataflow instructions are executed and results calculated
Memory (M) - memory and external peripheral requests are sent and received
Write Back (WB) - values are sent to the registers for saving

The M stage features two blocks, “SDRAM” and “Audio/SPART Interface,” that simply represent the external interface to the CPU. The “SDRAM” block is a state machine that controls memory operation, and the “Audio/SPART Interface” block represents the alerts to/from the Audio and SPART blocks that are controlled by the FLC and FLS instructions.

The following diagram is a high-level representation of the CPU. Smaller external diagrams for each functional block depicted follow the main diagram, as does a table summarizing our ISA.

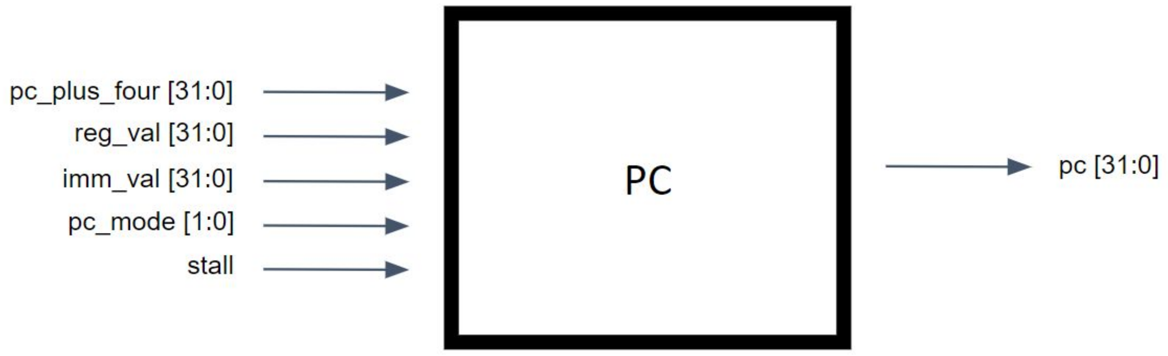
2.2.3 Incrementer

This block simply adds four to the PC value



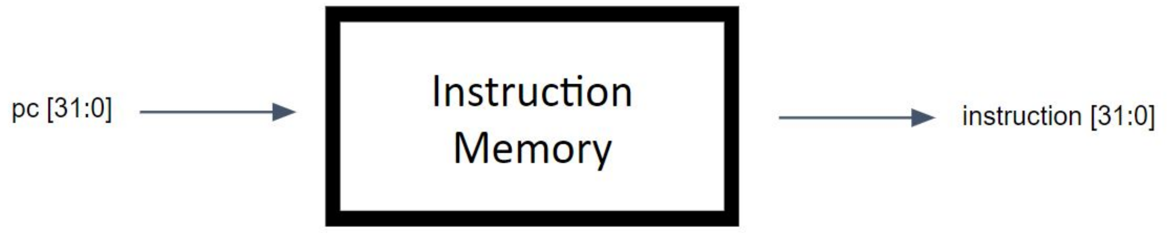
2.2.4 PC

Our Program Counter has three possible inputs to update its value: The old value plus four from the Incrementer, an immediate value included in a JMP instruction, or a value from one of the 32-bit registers. Additionally, the stall flag will prevent the PC from updating in the event of a pipeline stall to wait for SDRAM.



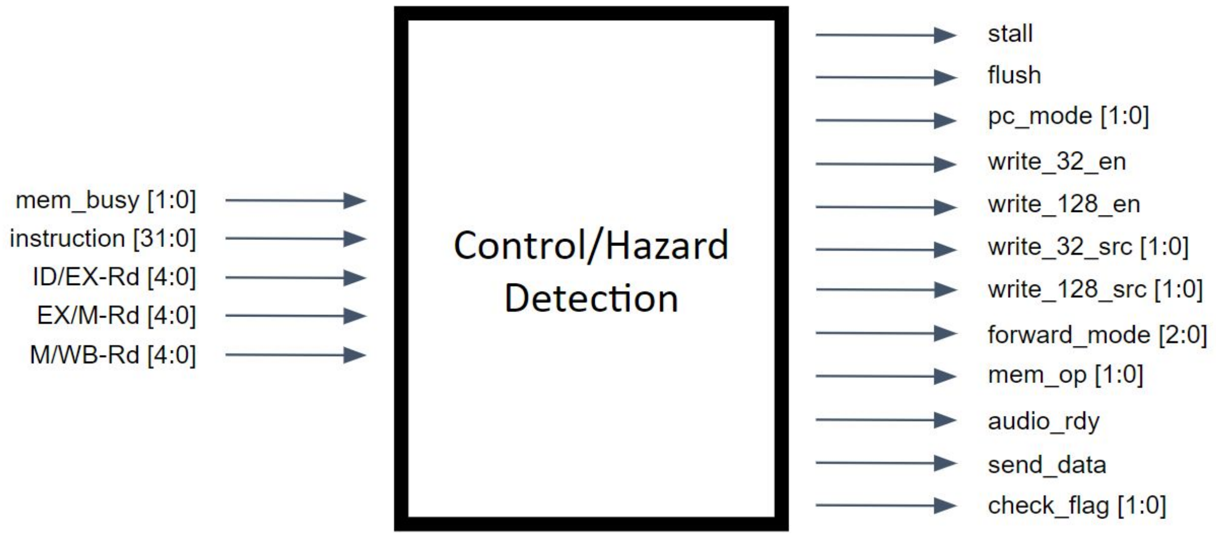
2.2.5 Instruction Memory

Instruction memory acts as a ROM and contains all instructions. Given the address of an instruction, Instruction Memory fetches the 32 bit instruction.



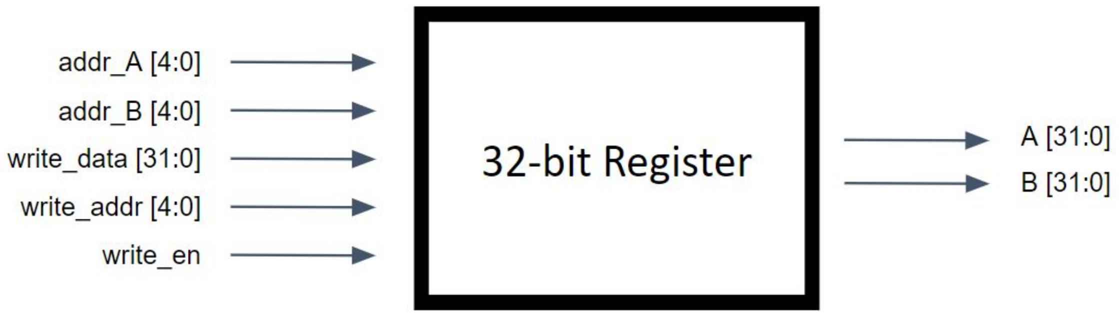
2.2.6 Control/Hazard Detection

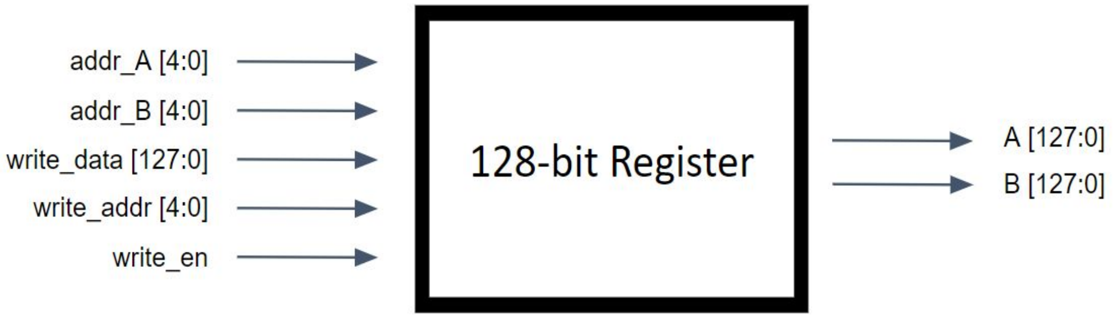
The control block takes the current instruction and the value of the mem_busy flag from the MEM controller and determines how to configure the rest of the CPU to ensure instructions are executed correctly. This block also examines the registers being used to determine when data forwarding through the pipeline needs to occur. The stall flag is used to stall the pipeline while waiting for a memory operation with the mem_busy flag.



2.2.7 Registers

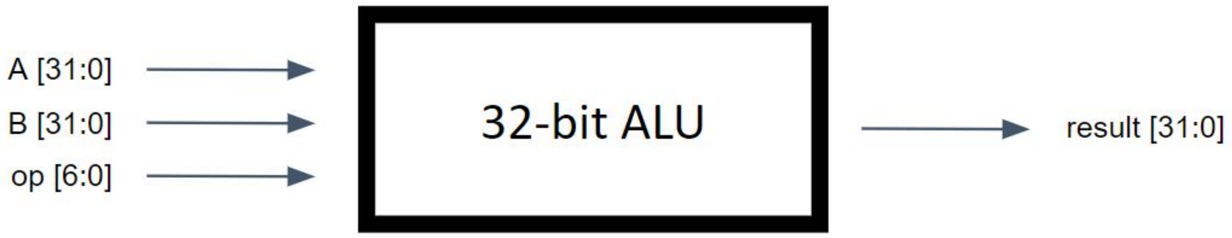
There are two sets of registers, one 32-bit and one 128-bit, each with 32 locations. These registers can output two values each clock cycle. If write_en is enabled, one value can also be written per cycle.

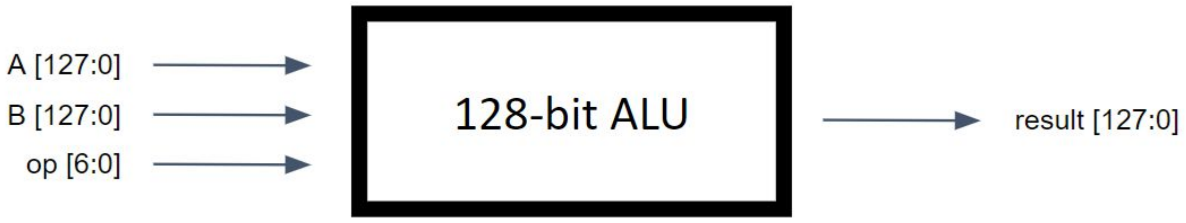




2.2.8 ALUs

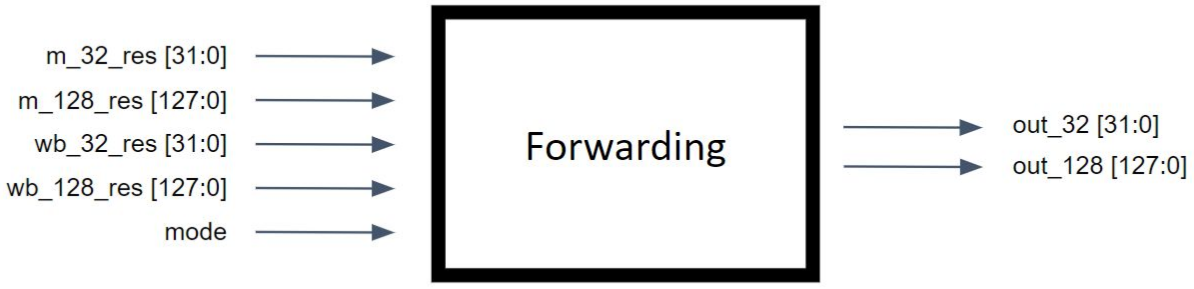
There are also two ALUs in our system, one 32-bit and one 128-bit. These ALUs implement the arithmetic, logic, and decryption operations listed in the ISA.





2.2.9 Forwarding

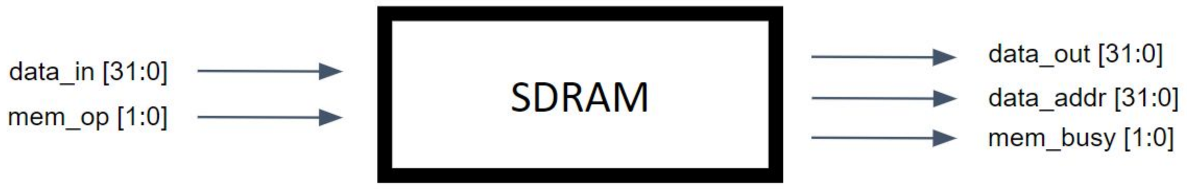
The forwarding unit sends back the 32 and 128 bit outputs from the ALU of either the MEM or WB stage in the pipeline. The mode determines which output is forwarded back.



2.2.10 SDRAM Interface

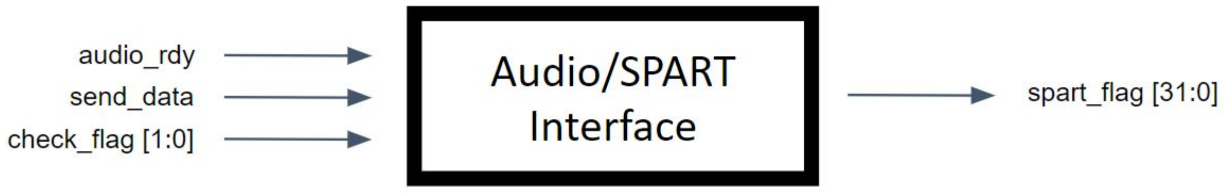
The SDRAM interface block is merely the connection point for the CPU's internal signals to the external interface to the MEM controller defined in the section 2 table.

24 bits address multiplied by two for mem controller



2.2.11 Audio/SPART Interface

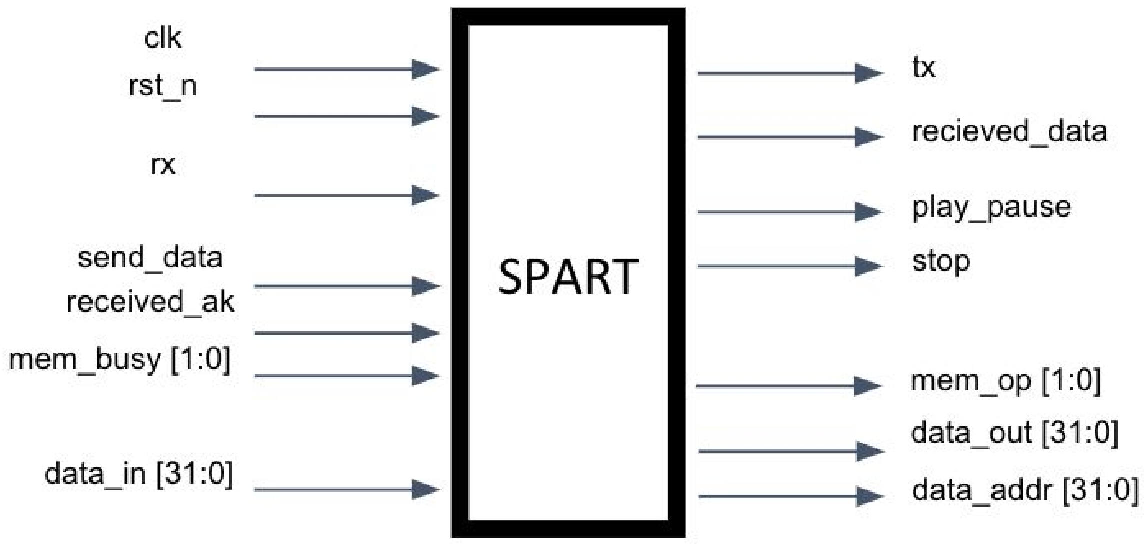
The Audio/SPART interface is the bridge between the audio module and the SPART. It outputs a 32 bit audio info stream to be OR'd with the register data.



3. SPART

The SPART is used to communicate with the music application running on an external computer. The SPART will handle requests to play various songs. The external source will drop the Rx bit and initiate the request. The spart will output the decrypted song if the user holds a valid license. It will send this output on Tx. To maximize our transmit speed we will run the SPART at a fixed rate of 19200. The following chart outlines the SPART's hardware interface.

Signal Name	I/O	Source/Target Block	Description
clk	in		
rst_n	in		
tx	out	GPIO0	Serial transmission line
rx	in	GPIO0	Serial receiving line
send_data	in	CPU	Alert from CPU saying data is ready to be sent after using SPARTS instruction
recieved_data	out	CPU	Alert for CPU saying new data has been received, held high till ak'd by CPU
stop_data	out	CPU	Alert for CPU saying to stop playback,, held high till ak'd by CPU
recieved_ak	in	CPU	Acknowledge signal from CPU after it checks recieved_data using FLC instruction
stop_ak	in	CPU	Acknowledge signal from CPU after it checks stop_data using FLC instruction
data_in [31:0]	in	MEM	32-bit Data in line to MEM
data_out [31:0]	out	MEM	32-bit Data out line from MEM
data_addr [31:0]	out	MEM	32-bit Address for MEM
mem_op [1:0]	out	MEM	2-bit MEM control line: ● 00 = SPART not requesting memory ● 01 = SPART requesting read ● 10 = SPART requesting write
mem_busy [1:0]	in	MEM	2-bit MEM busy signal: ● 00 = memory not busy ● 01 = memory doing CPU task ● 10 = memory doing SPART task ● 11 = memory doing Audio task



4. SDRAM

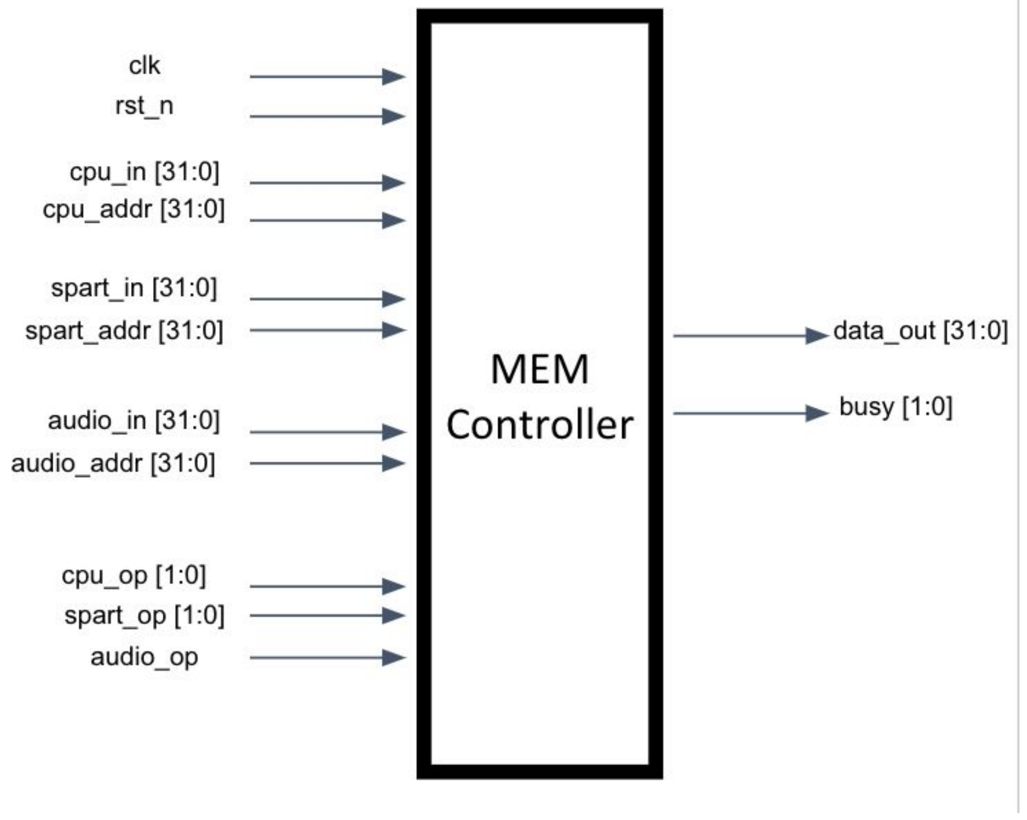
The CPU and SPART both can read and write to SDRAM to facilitate data transfer. We use an Altera-generated IP to control the chip I/O, but since both modules can ask to use the SDRAM at the same time, we made a wrapper controller to organize memory requests.

4.1 MEM Controller

This peripheral is used to coordinate the memory with the other hardware blocks, in conjunction with an IP-generated SDRAM controller. All signals are 1-bit unless specified otherwise.

Signal Name	I/O	Source/Target Block	Description
clk	in		
rst_n	in		
cpu_in [31:0]	in	CPU	32-bit CPU data in line
spart_in [31:0]	in	SPART	32-bit SPART data in line
audio_in [31:0]	in	Audio	32-bit Audio data in line
data_out [31:0]	out	CPU, SPART	32-bit data out line
cpu_addr [31:0]	in	CPU	32-bit CPU address to memory

spart_addr [31:0]	in	SPART	32-bit SPART address to memory
audio_addr [31:0]	in	Audio	32-bit Audio address to memory
cpu_op [1:0]	in	CPU	2-bit CPU control line: ● 00 = CPU not requesting memory ● 01 = CPU requesting read ● 10 = CPU requesting write
spart_op [1:0]	in	SPART	2-bit SPART control line: ● 00 = SPART not requesting memory ● 01 = SPART requesting read ● 10 = SPART requesting write
audio_op	in	Audio	Audio control line: ● 0 = Audio not requesting memory ● 1 = Audio requesting read
busy [1:0]	out	CPU, SPART, Audio	2-bit busy signal: ● 00 = memory not busy ● 01 = memory doing CPU task ● 10 = memory doing SPART task ● 11 = memory doing Audio task

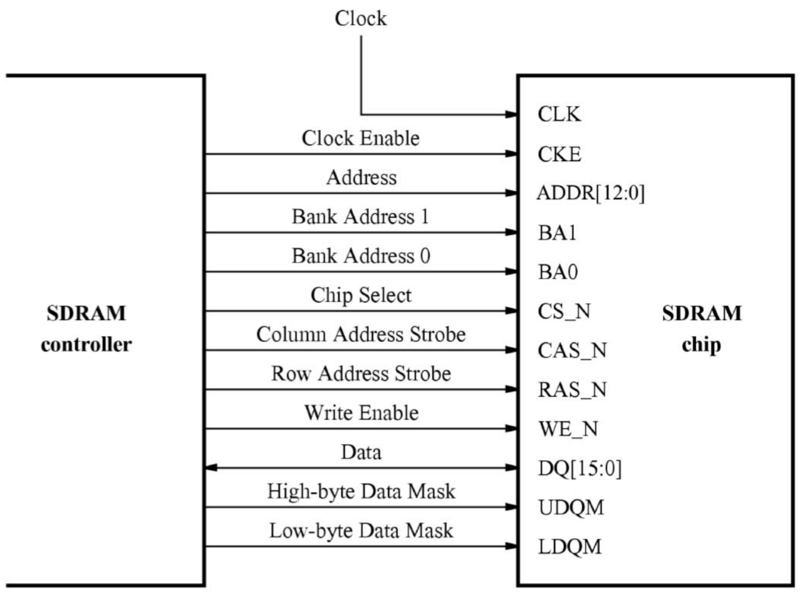


4.2 MEM Controller

Our MEM controller will act as a wrapper for the SDRAM chip and SDRAM controller modules that are included on the DE1-SoC board. The SDRAM Controller handles memory access requests from the SDRAM. It can access data from any of the four 16-bit memory banks.

Signal Name	I/O	Source/Target Block	Description
clock_enable	out	SDRAM	Enable signal for the SDRAM clock
address [12:0]	out	SDRAM	13-bit address line for output vectors
bank_addr1	out	SDRAM	Selector for SDRAM bank
bank_addr0	out	SDRAM	Selector for SDRAM bank
cs	out	SDRAM	Chip Select

cas	out	SDRAM	Column address strobe
ras	out	SDRAM	Row address strobe
w_en	out	SDRAM	Write enable
data [15:0]	inout	SDRAM	Transmitted 16-bit bank data between SDRAM and SDRAM Controller.
hbdm	out	SDRAM	High-byte data mask
lbdm	out	SDRAM	Low-byte data mask



5. Software

5.1. Assembler

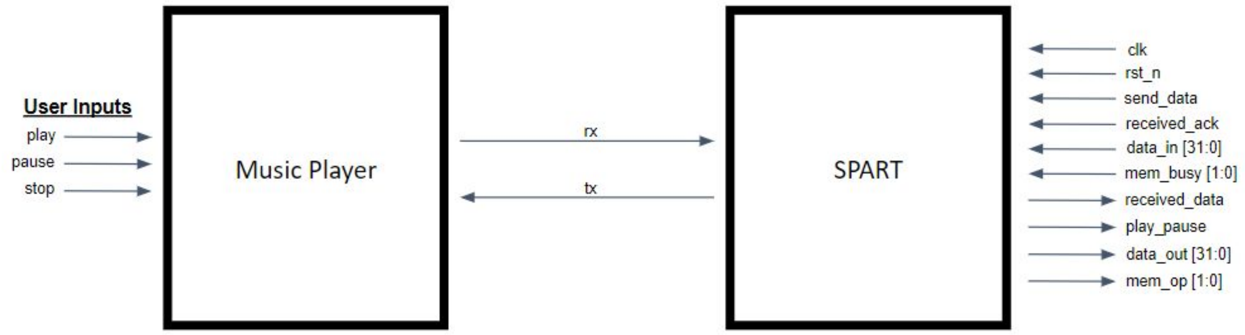
We will be writing an assembler that will first generate a symbol table and then generate the machine code. This will be used to write the hashing algorithm on the board. Assembly instructions will take the form INST [REG] [REG] and be converted into 32 bit instruction that will be readable by our processor. Registers will be referred to by R1, R2, or ED1, ED2, etc.

5.2. Application

Our application will mimic a basic music player. It will display the current music file. The interface will be displayed via monitor which will have to be connected serially to the FPGA along with a Putty link.

Users will interact with the application through the play button, deciding which song they want to play and having the ability to stop the song at will.

The application we made will use the 4,460kb of on-board memory to store our encrypted songs. Then the CPU will decrypt the encrypted data and store this on the board too. This data will be then sent back to our software audio player to be played back. Once the music is sent the initial time, you can simply flip a switch on the board to have it decode again, be sent back to the music player.



6. Demo/Verification

6.1 Demo

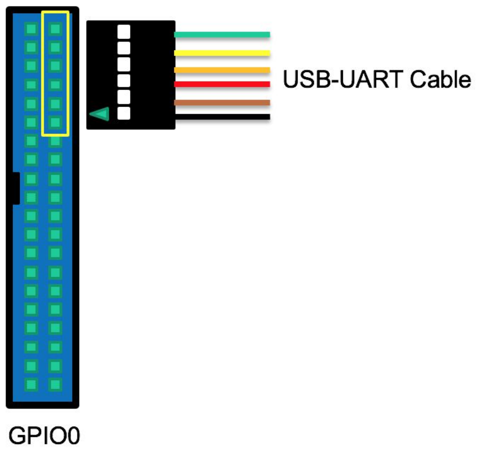
For our demo, we wrote an assembly program that loads the AES key into the CPU, then waits for the SPART to send the encrypted WAV file over. Once the Python software initiates the data transfer, our hardware writes the data to SDRAM at memory locations 0-80000. Once data transfer is finished, our assembly loops through reading the encrypted data, decrypting it, then writing it back to SDRAM locations 160000-80000. Finally, the SPART reads the unencrypted data and sends it back to the software, which receives it and plays the WAV file.

6.2 Testing / Verification

To test our full application, we first generated the encrypted file in python using 128 AES encryption with the key we expected. This was saved a hex file with all the data we would be sending to the board. We then output a file with the decrypted hex data and saved this. As the application runs and we received the unencrypted data back from the SPART and saved this to a file. This allows us to verify all the data was sent and unencrypted correctly.

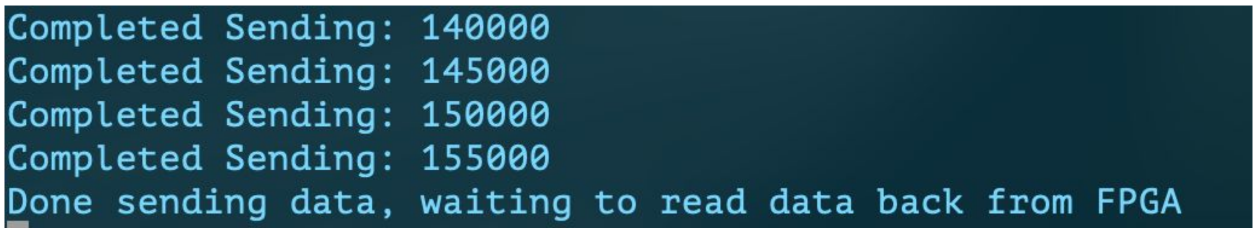
6.3 Demo Running Instructions

To Demo the project, you have to hook up a USB-UART FTDI Cable to the top right 6 pins of GPIO0 as depicted below:



Follow the steps below:

1) Load the project on the DE1-SoC FPGA
2) Make sure all switches on the board are down to begin.
3) Run python script named ‘play_audio.py’.
4) Wait for the encrypted song to send, and you should see an output similar to the one below:



```
Completed Sending: 140000
Completed Sending: 145000
Completed Sending: 150000
Completed Sending: 155000
Done sending data, waiting to read data back from FPGA
```

5) Switch only SW[8] and SW[9] to high:
6) Wait for send back to complete
7) Audio will play in an infinite loop, if the encryption key from the python script is incorrect, a static noise will play, otherwise your intended audio file will play.

7. Results

If we had additional time we would have liked to use the HPS to send data from the computer through the ethernet cable. This would have allowed much faster transfer times. The SPART was limited to 19200 baud rate which results in a very slow data transfer speeds.

Ideally, we would have liked the audio to be played through the FPGA’s aux port; This would have allowed us to better secure songs by not sending unencrypted data back to the computer where it could be recorded and saved by the user. It would have also sped up the application by not having to wait for the SPART to transfer it back. However, due to time constraints, the IP generated audio driver proved to be more difficult to understand and manipulate.

Our biggest mistake was underestimating the time it took to get the Altera IP modules up and running. The documentation on them isn’t very extensive, so we had to spend the majority of our time debugging and testing them as the project demo drew near. We eventually conquered the SDRAM IP using smart debugging and targeted tests.

8. Team Contributions

Tristen: Wrote assembler and a command line interface in python for the initial GUI application. Also worked on testing different versions of SPART modules in order to optimize transfer.

Andrew: Focused on the Python application, the SPART, and the memory controller which is shared between the CPU and SPART. Worked on the debugging for all applications.

Alec: Focused on sub-modules within the CPU including the decode stage and control/hazard detection. Worked mostly on the Audio_controller module and Audio IP’s generated by Quartus and others that were found online.

Alex: I mostly focused on the CPU and then stitching the toplevel together. That basically consisted with writing modules and debugging the CPU in ModelSim first, and then plugging it in and getting it working with the SPART and SDRAM controller. I also spent some time trying to understand the audio IP we failed to get working towards the end, and then switched to modifying the SPART to send data from memory back to the software.

Sadeq: Designed the memory controller that allowed multiple modules access and use the sdram. Generated and learnt the Quartus generated SDRAM IP. Assisted in testing audio drivers.