APPENDIX A

The Sedici Microprocessor System

A.1 Introducing the Sedici Microprocessor

Sedici is a simple 16-bit processor designed for educational use. It is based on the simulated processor JASP which was used to demonstrate the computer organization and architecture in the book 'Fundamentals of Computer Architecture'. The main improvements in Sedici are:

- A standard memory map from `$0000` to `$FFFF`;
- All immediate operands are 16-bits;
- Eight scratch registers, D0 through to D7 (compared to only A and B in JASP);
- An improved and simplified instruction set.

Additionally, all the tools that simulate the Sedici processor, the assembler, the firmware builder, etc. have all been written from scratch in `C#`.

A.2 Looking inside the Sedici Microprocessor

This section covers:

- A description of the function of each register;
- The micro-instructions understood by the Sedici control unit;
- A description of the memory map, including a description of the memory-mapped peripherals;

- The interrupt mechanism;
- The file formats used by Sedici for both instruction sets and machine code.

A.2.1 Registers

The processor registers are listed in table A.1.

Register	Width (bits)	Description
PC	16	Program counter - *is used to keep track of the memory address storing the next instruction to be executed*
INC	16	Incrementer - *is used to add one to the value held in the PC, something that needs to occur very often in most programs. Using the incrementer (effectively as a specialist register) is faster than using the ALU for this particular task, and importantly does not affect the PSR flags*
D0 to D7	16	Set of 8 General Registers - *used to store program bit patterns*
MAR	16	Memory Address Register - *is used as a specialist register to store the address of the memory location that we need to read from or write to*
MDR	16	Memory Data Register - *is used as a specialist register to store the data that we have just read from memory or need to write to memory*
IR	16	Instruction Register - *is the specialist register where we store the instruction once it has been fetched from memory*
ALUx	16	Arithmetic Logic Unit X Register *is the first of two specialist registers where we store bit patterns to be used in ALU operations*
ALUy	16	Arithmetic Logic Unit Y Register *is the second of two specialist registers where we store bit patterns to be used in ALU operations*
ALUr	16	Arithmetic Logic Unit Result Register *is the specialist register where the result from an ALU operation is stored*
USP	16	User Stack Pointer - *is the specialist register used to store the address of the top of the user stack held in memory*
SSP	16	Supervisor Stack Pointer - *is the specialist register used to store the address of the top of the supervisor stack held in memory*
PSR	16	Processor Status Register - *is where we store information about the state of the processor, including the state of the last ALU operation*

Table A.1 ***Processor registers***

The Instruction Register

The instruction registers has a number of fields:

```
 ----------------
|XXXXXXXXaaaabbbb|
 ----------------

  XXXXXXXX = opcode
  aaaa     = r0
  bbbb     = r1
```

r0 and r1 can be used as a shortcut for the microprogram for an instruction. For example, a microprogram could use the data movement PC←[IR(r0)] which would use the contents of r0 to 'look-up' the contents of a specific register, and move the contents of that register into the PC.

The short-cut codes are as follows:

Registers	Look-up value	Look-up value (hex)
D0-D7	0 to 7	0 to 7
MAR	8	8
MDR	9	9
ALUx	10	A
ALUy	11	B
ALUr	12	C
USP	13	D
SSP	14	E
PC	15	F

A.2.2 Micro-Instructions

The Sedici processor has a micro-programmed control unit, where each machine code instruction is defined as a sequence of micro-instructions known as a micro-program. These micro-programs are used by the control unit to execute individual instructions.

These micro-programs can be grouped together in an instruction set file, sometimes referred to as a microcode file or micro-instruction file.

All the micro-instructions that are recognized by the processor can be separated into one of five distinct micro-instruction groups. These are:

- Data movement micro-instructions;
- ALU micro-instructions;
- Test micro-instructions;

- Interrupt mechanism micro-instructions;
- Processor control micro-instructions.

All the micro-instructions within each group are described below.

Data Movement Micro-Instructions

The possible data movements are listed in table A.2. Note that the destination and source can be the same location - perfectly valid, albeit pointless.

RTL	Notes
Destination←[Source]	Standard data movement
MDR←[M[MAR]]	Performs a memory read operation
M[MAR]←[MDR]	Performs a memory write operation
CU←[IR(opcode)]	8-bit transfer of instruction to the CU

Table A.2 ***Data movement micro-instructions***

Where Source and Destination come from:

```
Registers     Look-up value
D0-D7          0 to 7
MAR            8
MDR            9
ALUx          10
ALUy          11
ALUr          12
USP           13
SSP           14
PC            15
--------------
PSR           Entries below this line cannot
IR            be referenced through IR(r0), IR(r1)
INC

Shortcuts
IR(r0)
IR(r1)

Misc
JUMPERS(IntBase)
PSR(IntVec)
```

This list is in the order of the codes assigned to the registers - the first 16 (0 to 15) can be used as op1 and op2.

ALU Micro-Instructions

The ALU micro-instructions are listed in table A.3.

Code	Operation	RTL	Notes
0000	ADD	ALUr=[ALUx]+[ALUy]	Perform a 2's complement ADD operation, adding the ALUx and ALUy bit patterns together and storing the result in the ALUr register
0001	ADC	ALUr=[ALUx]+[ALUy]+[PSR(c)]	Perform a 2's complement ADC operation, adding the ALUx and ALUy and C flag together and storing the result in the ALUr register
0010	SUB	ALUr=[ALUx]-[ALUy]	Perform a 2's complement SUB operation, subtracting the ALUy from the ALUx bit pattern and storing the result in the ALUr register
0011	SL	ALUr=[ALUx]<<1	Perform a logical shift left on the ALUx, storing the result in the ALUr
0100	SR	ALUr=[ALUx]>>1	Perform a logical shift right on the ALUx, storing the result in the ALUr
0101	AND	ALUr=[ALUx]&[ALUy]	Perform a logical AND operation on the ALUx and ALUy bit patterns and storing the result in the ALUr register
0110	OR	ALUr=[ALUx]|[ALUy]	Perform a logical OR operation on the ALUx and ALUy bit patterns and storing the result in the ALUr register
0111	NOT	ALUr=~[ALUx]	Perform a logical NOT operation on the ALUx and storing the result in the ALUr register
1001	INC	ALUr=[ALUx]+1	Add 1 to the ALUx bit pattern, storing the result in the ALUr
1010	DEC	ALUr=[ALUx]-1	Subtract 1 from the ALUx bit pattern, storing the result in the ALUr
1010	ASR		
1010	ASL		
1010	ROR		
1010	ROL		

Table A.3 ***ALU micro-instructions***

Whenever an ALU operation is executed, the PSR flags V, N, C and Z are updated. Table A.4 shows how the flags are updated by each ALU operation - a key to this table is given in table A.5.

Operation	V	N	Z	C
ADD	*	*	*	*
ADC	*	*	*	*
SUB	*	*	*	*
SL	0	*	*	*
SR	0	*	*	*
AND	0	*	*	0
OR	0	*	*	0
NOT	0	*	*	0
INC	*	*	*	*
DEC	*	*	*	*
ASR	?	?	?	?
ASL	?	?	?	?
ROR	?	?	?	?
ROL	?	?	?	?

Table A.4 ***How ALU operations affect the PSR flags***

Flag	Meaning
V	* means that if 2's complement overflow occurs then V=1 else V-0 (division overflow in cases of DIV and MOD) 0 means that V=0
N	* means N=MSB(ALUr)
Z	* means if (ALUr==0) then Z=1 else Z=0
C	* means if (carry from MSB of ALUr) then C=1 else C=0 * except with SR this means if (carry from LSB of ALUr) then C=1 else C=0 0 means that C=0

Table A.5 ***The key to figure A.4***

Test Micro-Instructions

The four PSR flags may be tested. If a test evaluates to TRUE, any remaining micro-instructions in that microprogram are executed. Otherwise the micro-instructions following the test are ignored.

The valid test micro-instructions are listed in table A.6.

RTL	Notes
if(PSR(flag)==1)	Flag set
if(PSR(flag)==0)	Flag clear

Table A.6 ***Test micro-instructions***

Where flag in set of (V,N, C, Z).

Interrupt Mechanism Micro-Instructions

The valid interrupt mechanism micro-instructions are listed in table A.7.

RTL	Notes
PSR(e)=1	Enable interrupts
PSR(e)=0	Disable interrupts
PSR(i)=1	Raise interrupt
PSR(i)=0	Kill interrupt

Table A.7 ***Interrupt mechanism micro-instructions***

Where intflag in set of (I, E).

Processor Control Micro-Instructions

The valid processor control micro-instructions are listed in table A.8.

RTL	Notes
HALT	Processor halt
NOP	No operation

Table A.8 ***Processor control micro-instructions***

A.2.3 The Interrupt Mechanism

The interrupt mechanism makes use of an interrupt vector table stored in memory, and the I and E flags of the PSR.

The Sedici processor can only deal with a single interrupt at any given time - any further interrupts generated while the first interrupt is being handled will be ignored. The actual details of the interrupt mechanism are definable within the instruction set. Note that currently the interrupt mechanism uses the User Stack Pointer (USP), not the Supervisor Stack Pointer (SSP).

Within the default instruction set the interrupt mechanism is defined as:

```
PSR(I)=0                    interrupt flag = 0
MAR<-[USP]                     }     save PSR
MDR<-[PSR]                   }     on the stack
M[MAR]<-[MDR]                }
ALUx<-[USP]                 }  decrement
ALUr=[ALUx]-1              }  SP
USP<-[ALUr]                 }
ALUx<-[PC]                   }
MDR<-[ALUx]                  }     write PC
MAR<-[USP]                    }     to the stack
M[MAR]<-[MDR]                }
ALUx<-[USP]                 }  decrement
ALUr=[ALUx]-1              }  SP
USP<-[ALUr]                 }
PSR(E)=0                   interrupt enable flag = 0
ALUy<-[JUMPERS(IntBase)]     }
ALUx<-[PSR(IntVec)]          } build the vector address
ALUr=[ALUx]+[ALUy]           }
MAR<-[ALUr]                } obtain the handler address
MDR<-[M[MAR]]              }
PC<-[MDR]                  load address of handler into PC
```

The position of the vector table is configurable, but it defaults to the locations `$00F0` to `$00F7`.

A.3 Memory

Sedici has 128Kb of memory, accessed as $65,536$ 16-bit words (addresses $0000 to $FFFF).

An address points to a 16-bit word and all memory accesses are words.

Two registers are associated with memory accesses. The Memory Data Register (MDR) contains the data value which is about to be written to memory or a value which has been read from memory. The Memory Address Register (MAR) contains the memory address of a read or write operation.

Think of MAR as a pointer to a word of memory. The pointer may be moved by altering the value held in MAR. Values may be transferred from the MDR to memory (Write) or from Memory to the MDR (Read).

To write a value into memory you do the following:

```
RTL               Description

MDR<-00FF         Place data in MDR
MAR<-0010         Place address in MAR
M[MAR]<-[MDR]     Update memory
```

Note the sequence of operations performed when writing data into memory. The address and data values are loaded into the MAR and MDR respectively and a write cycle is performed.

A memory read is as follows:

```
RTL               Description

MAR<-0010         Place address in MAR
MDR<-[M[MAR]]     Read memory
                  MDR now contains [M[0010]]
```

The memory map is shown in figure A.1. **This diagram requires updating.**

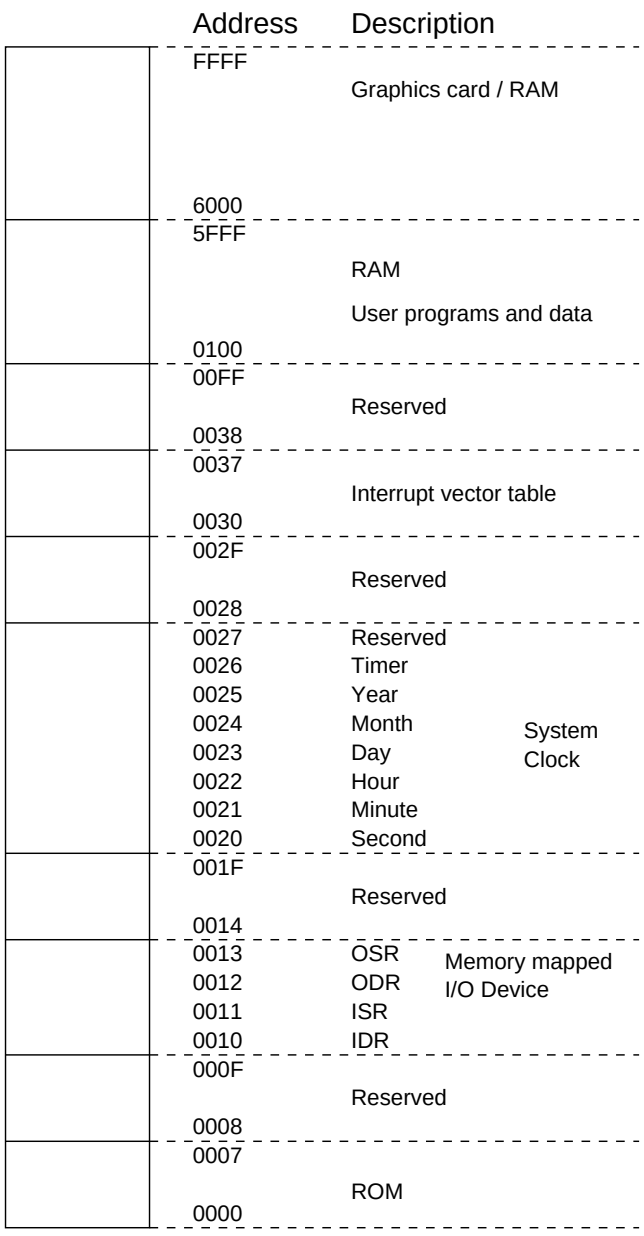

Figure A.1 ***The Sedici memory map***

When accessing or writing to memory, addresses are not wrapped.

All peripherals have default locations within the memory map, but their locations are configurable.

A.3.1 Memory Mapped I/O

Handshaking needs to be used in order to perform I/O.

To write a character to the screen, first check that the OSR port is set to 1, if it's not go into a loop until it is. Only then write the character to the ODR.

To read a character from the keyboard, keep checking until the ISR is set to 1, only then should you read the character from the IDR.

Here is a piece of code that shows handshaking for both input and output:

TODO

A.3.2 Current System Time

Additionally, Sedici has a system clock device. The current date and time is accessible from `$00E8` to `$00ED`. No handshaking is required, simply access the particular memory location for the required date/time value. You cannot write to these memory locations, although no errors are raised if you try.

A.4 Firmware explained

A detailed description of each assembler instruction in the standard firmware.

A.4.1 MOVE instructions

Bla bla.

MOVE addr,R

Bla bla.

A.5 Using the Sedici Assembler

A.5.1 Introduction

The Sedici assembler isn't quite a true assembler, more of a (not so) simple text processor. It takes an assembler source file and a firmware description file and attempts to do it's best to produce a machine code file that can be used

by the Sedici microprocessor system. The assembler actually knows nothing about assembly code, it tries to make a best guess as to a instruction being used - only the mnemonic is used to figure out what to do with the instruction.

A.5.2 A first example

In this section I'll include a simple source file and assembler it.

A.5.3 Directives

USE

The format of a USE directive is as follows:

```
        USE library
```

Each library is pre-processed prior to the assembly proper - and if a library doesn't exist then the assembly is halted before it's begun.

EQU

The format of a EQU directive is as follows:

```
label    EQU data
```

The label is necessary (and must begin the line, i.e. the first letter of the label must be in column 1), and `data` is a 16-bit number. For example,

```
IDR    EQU $E0
```

DC.B

The format of a DC.B directive is as follows:

```
{label}    DC.B data
```

Where `data` is a set of strings (indicated by apostrophes) or 8-bit values, each separated by a comma. For example:

```
message    DC.B 'Hello',$0a,$0d,0
```

Apostrophes and asterisks can also be included in your stings provided that they are ‘escaped’, i.e. preceded by a
character, like in this example:

```
message    DC.B ’My name is \’Fred\’ and here is an asterisk \*’,$0a,$0d,0
```

DC.W

The format of a DC.W directive is as follows:

```
{label}    DC.W data
```

Where `data` is a set of 16-bit values, each separated by a comma. For example:

```
stuff     DC.W $1234,$00FF
```

Entries can also be a reference to something in the symbol table. An example is:

```
        ORG  $36              * Interrupt vector 6 is set to
        DC.W #timerhandler    *   the timerhandler routine
```

DS.W

The format of a DS.W directive is as follows:

```
{label}    DS.W data
```

Basically, this is a way of defining a storage area in your program. `label` can be an optional symbol placed in the symbol table to point to the start of the storage area. `data` can be either a 16-bit number or can be a reference to a symbol in the symbol table (i.e. defined by an EQU somewhere in your program). Examples are:

```
limit    EQU  $3
message  DS.W #limit
```

or

```
limit2   DS.W ^10
```

FRM

The format of an FRM directive is as follows:

```
FRM firmwarefile
```

The `firmwarefile` is the name of a firmware file - it is used within the assembly process. For most programs you'll just want to use the standard firmware and so wouldn't need this instruction.

ORG

The format of an ORG directive is as follows:

```
ORG data
```

The `data` must be a 16-bit number - it is used to set the origin for the program.

LPC

The format of an LPC directive is as follows:

```
LPC data
```

The `data` must be a 16-bit number - it is used to pass an instruction through to the processor to set the PC to a specific value - that held in `data`.

A.5.4 Assembler instructions

The assembler has to deal with all manner of instructions - some that haven't even been written yet. That is quite some undertaking.

Currently the assembler program understands the following addressing modes:

- Register - e.g. R
- Register Indirect - e.g. (R)
- Immediate - e.g. `#F123`
- Direct - e.g. `$1234`
- Indirect - e.g. `($1234)`

The assembler can cope with multiple levels of indirection (basically it ignores brackets), it is up to you, the instruction writer, to ensure it does what it is supposed to do - the assembler merely (merely? It's not easy) figures out which opcode a mnemonic within a program matches.

How to write firmware for use with the assembler

Bla bla.

For example, if creating a new arithmetic or logical instruction - ensure that the result is in the last register listed in the mnemonic. For example, `ADD (address),#data,R` should always end up storing the result of the operation in R. So something like `ADD R0,R1` where the result is stored in R1 is a no-no. Keep to this simple rule (as the default instruction set does) and your users won't come a cropper (or at least on where they expect the result to be stored).

A.5.5 Libraries

io.lib

The basic I/O library. To be ported from routines found in the old JASP advancedio.lib.

math.lib

Not written yet.

float.lib

Not written yet.

graphics.lib

Not written yet.

A.6 Using the Sedici Compiler

A.6.1 Introduction

It's not been written yet.

Bibliography