Synergistic Processor Unit Model

Wilmer Suarez **|** 109592501  
ESE 545 **|** SPRING 2019

**Table of Contents**

[**I.** **Introduction** 2](#_Toc7794108)

[**II.** **Design Process of Core Architecture** 2](#_Toc7794109)

[**III.** **SPU Core Block Diagram** 3](#_Toc7794110)

[**IV.** **SPU Top Module – Tests & Results** 4](#_Toc7794111)

[**Assembler Test** 4](#_Toc7794112)

[**Cache Miss Test** 4](#_Toc7794113)

[**All Instruction Test** 4](#_Toc7794114)

[**Local Store Fill Test** 4](#_Toc7794115)

[**Dual Instruction – Fetch\_Decode\_Issue\_Execute (No Hazard) Test** 4](#_Toc7794116)

[**Structural Hazard Resolution Test** 4](#_Toc7794117)

[**Data Hazard Resolution by Forwarding (No Stalling) Test** 4](#_Toc7794118)

[**Data Hazard Resolution by Stalling and Forwarding Test** 4](#_Toc7794119)

[**Control Hazard Resolution for Branches Test** 4](#_Toc7794120)

[**Appendix A – ISA** 5](#_Toc7794121)

[**Simple-Fixed 1 Instructions** 5](#_Toc7794122)

[**Simple-Fixed 2 Instructions** 6](#_Toc7794123)

[**Floating Point Instructions** 6](#_Toc7794124)

[**Floating Point Integer Instructions** 7](#_Toc7794125)

[**Byte Instructions** 7](#_Toc7794126)

[**Local Store Instructions** 8](#_Toc7794127)

[**Permute Instructions** 8](#_Toc7794128)

[**Branch Instructions** 9](#_Toc7794129)

[**Control Instructions** 9](#_Toc7794130)

[**Appendix B – Assembler Code** 10](#_Toc7794131)

[**Appendix C – Module Code** 10](#_Toc7794132)

[**Appendix D – Testbench Code** 10](#_Toc7794133)

[**References** 11](#_Toc7794134)

Synergistic Processor Unit Model

Wilmer Suarez **|** 109592501  
ESE 545 **|** SPRING 2019

***Abstract* — This report will describe the design process of a Cell SPU core written in VHDL. The Processor exploits Instruction and Data Level Parallelism. It is a dual-issue, SIMD, RISC, Multimedia Processor that consists of a total of 12 pipeline stages. The sub-set of instructions (ISA) used are presented in** [**Appendix A**](#_Appendix_A_–) **(68 instructions).**

# **Introduction**

The fully synchronous processing core consists of:

A Local SRAM Storage, an Instruction Line Buffer used to hold Instructions read from the Local Store. A 128-entry x128-bit Register File for both, floating-point and integer operations. Two pipes; an “Odd Pipe” with Permute, Local Store, and Branch execution units. An “Even Pipe” with Floating Point, Byte, and two Fixed-Point execution units. Lastly, Forwarding Circuits that forward the instruction results when RAW dependencies arise (Data Hazard).

The SPUs SIMD capability can perform operations on 16 x 8-bit integers, 8 x 16-bit integers, 4 x 32-bit integers, or 4 floating point numbers every cycle. The SPU can complete at most two instructions/cycle; one on each of the execution pipes [[1][2]](#_References). Having a shared Register File “allows the highest level of performance for various workloads with the smallest number of registers.” [[2]](#_References)

Multimedia-based information, and vector processing, is the target of the SPU; performing heavy computations, in parallel, on 128-bit wide data. This data (the instructions) are issued in pairs each cycle. Some taking 2 cycles to complete (simple-fixed instructions) while others up to 7 cycles (integer multiplication) [[2]](#_References).

* [SPU Core Block Diagram](#_SPU_Core_Block)

# **Design Process of Core Architecture**

***Register File***

The Register File is 128 registers x 128 bits. It supports 6 output ports (reads) and 2 inputs (writes). The output sends data directly to the corresponding execution unit latches. Results are only written back to the Register File during the Write Back stage.

The Register Files assures correct readings are made by checking if either of the write back addresses are the same as any of the six read addresses and bypassing the arriving write back data directly to the corresponding output.

***Even Pipe***

* *Simple Fixed 1 Unit*
  + Arithmetic, Logicals, Compares
* *Simple Fixed 2 Unit*
  + Word Shifts & Rotates
* *Single Precision Unit*
  + Single Precision Floating Point & Integer Multiplies
* *Byte Unit*
  + Byte Ops

***Odd Pipe***

* *Permute Unit*
  + Quadword Shifts & Rotates
* *Local Store Unit*
  + Load and Store
* *Branch Unit*
  + Branches

Each pipe produces results for one instruction every cycle. This gives the processor a CPI of ½. 2 Instructions executed each cycle.

All instructions pass through 7 stages prior to writing their result back to the Register File. If an instruction depends on any of the instructions currently in the pipe, the corresponding instruction can be forwarded, using the Forwarding Circuits, after its unit’s latency. The execution unit types are in [Appendix A](#_Appendix_C_–), along with the instructions executed in each unit.

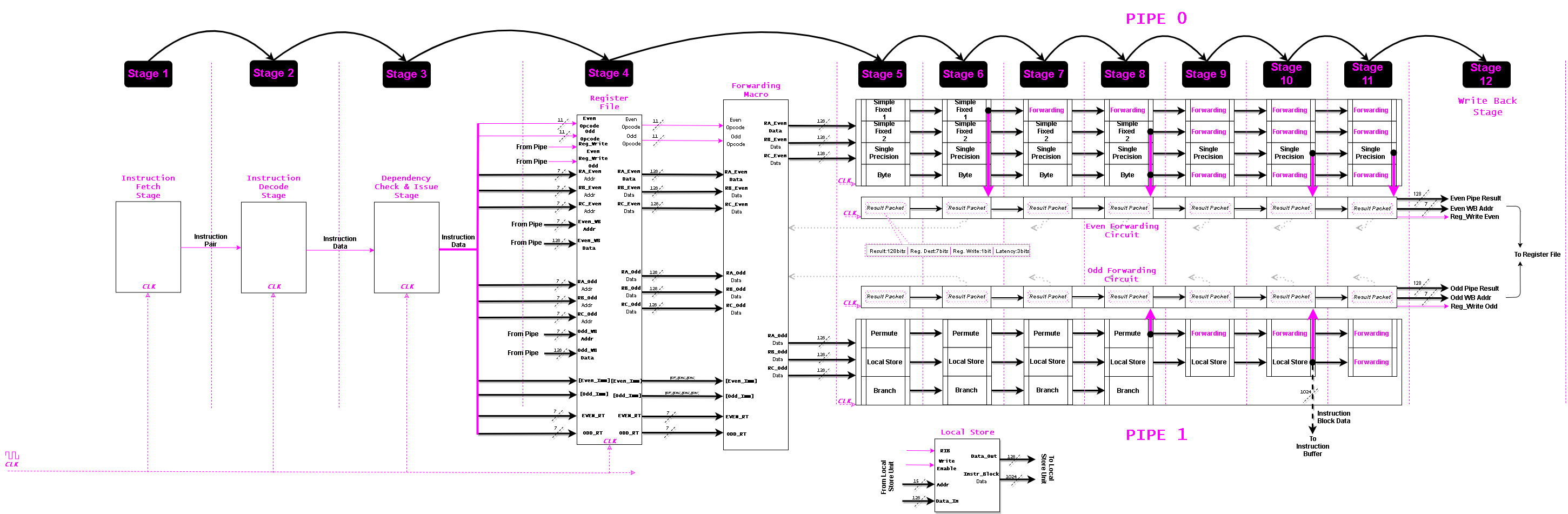
***Forwarding Circuits/Macro:***

If the instruction in the Register File is dependent on an instruction in either pipe, the Forwarding Circuits cycle through the pipes to retrieve the result and replace the dependent operand.

***Local Store***

The Local Store is a 32kB, single port SRAM. The first half is structured to store the current program code (starting at address 0). The second half – starting at address 16,384 – stores the program data. 16-Bytes of data can be read or written by Load & Store instructions, respectively. The Local Store also outputs 128-Byte instruction blocks (32 instructions) to fill the Instruction Line Buffer (ILB). The ILB signals the system to fetch the next instruction block after a miss when trying to fetch the next instruction pair.

# **SPU Core Block Diagram**



# **SPU Top Module – Tests & Results**

## **Assembler Test**

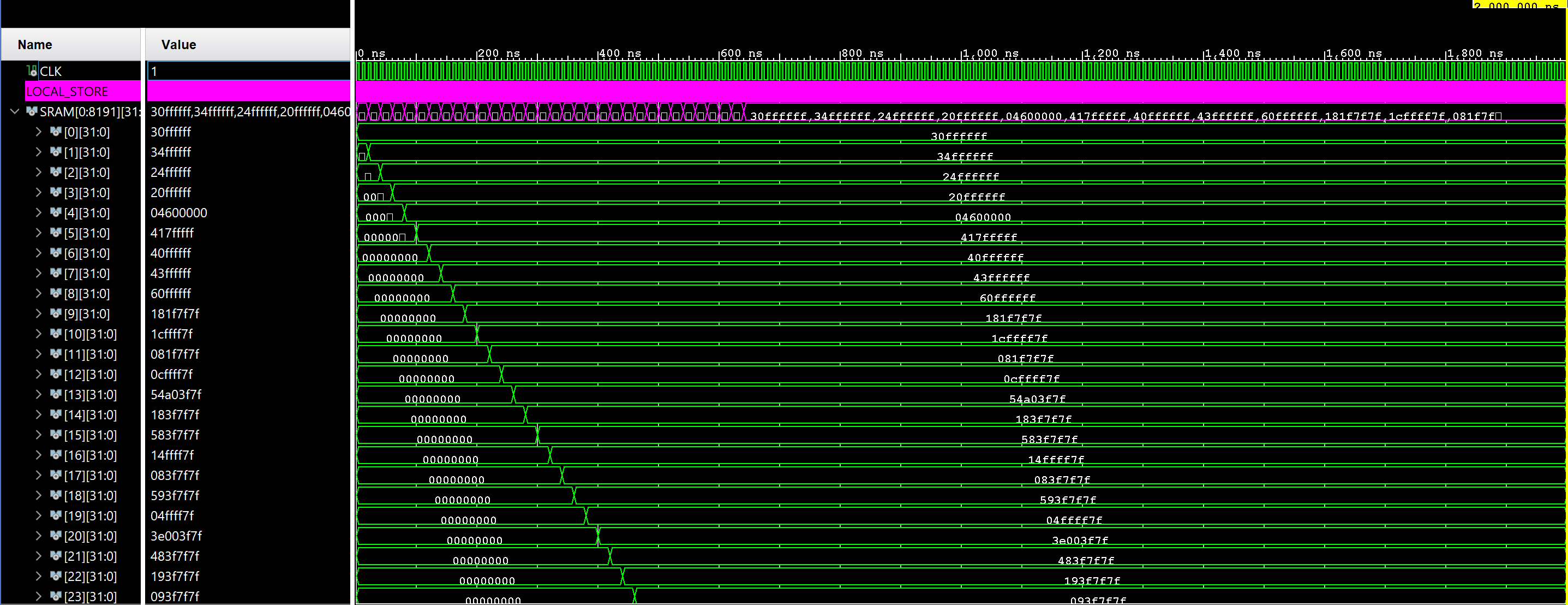
Shows the result from the assembler of an example assembly file, containing all 68 instructions.

|  |  |
| --- | --- |
| **INPUT** | **OUTPUT** |
| lqa 127, 0xFFFF  lqd 127, 0x3FF(127)  stqd 127, 0x3FF(127)  stqa 127, 0xFFFF  rib  ilhu 127, 0xFFFF  il 127, 0xFFFF  ila 127, 0x3FFFF  iohl 127, 0xFFFF  a 127, 126, 125  ai 127, 126, 0x3FF  sf 127, 126, 125  sfi 127, 126, 0x3FF  clz 127, 126  and 127, 126, 125  andc 127, 126, 125  andi 127, 126, 0x3FF  or 127, 126, 125  orc 127, 126, 125  ori 127, 126, 0x3FF  orx 127, 126  xor 127, 126, 125  nand 127, 126, 125  nor 127, 126, 125  ceq 127, 126, 125  ceqi 127, 126, 0x3FF  cgt 127, 126, 125  cgti 127, 126, 0x3FF  clgt 127, 126, 125  clgti 127, 126, 0x3FF  mpy 127, 126, 125  mpyu 127, 126, 125  mpyi 127, 126, 0x3FF  mpyui 127, 126, 0x3FF  mpya 127, 126, 125, 127  mpyh 127, 126, 125  mpyhhu 127, 126, 125  cntb 127, 126  avgb 127, 126, 125  absdb 127, 126, 125  sumb 127, 126, 125  shlqbii 127, 126, 125  shlqbyi 127, 126, 125  rotqbyi 127, 126, 125  rotqbii 127, 126, 125  shli 127, 126, 125  rothi 127, 126, 125  roti 127, 126, 125  br 0xFFFF  bra 0xFFFF  bi 127  brnz 127, 0xFFFF  brz 127, 0xFFFF  biz 127, 126  binz 127, 126  fa 127, 126, 125  fs 127, 126, 125  fm 127, 126, 125  fma 127, 126, 125, 127  fnms 127, 126, 125, 127  fms 127, 126, 125, 127  fceq 127, 126, 125  fcmeq 127, 126, 125  fcgt 127, 126, 125  fcmgt 127, 126, 125  stop  lnop  NOP | 30FFFFFF  34FFFFFF  24FFFFFF  20FFFFFF  04600000  417FFFFF  40FFFFFF  43FFFFFF  60FFFFFF  181F7F7F  1CFFFF7F  081F7F7F  0CFFFF7F  54A03F7F  183F7F7F  583F7F7F  14FFFF7F  083F7F7F  593F7F7F  04FFFF7F  3E003F7F  483F7F7F  193F7F7F  093F7F7F  781F7F7F  7CFFFF7F  481F7F7F  4CFFFF7F  581F7F7F  5CFFFF7F  789F7F7F  799F7F7F  74FFFF7F  75FFFF7F  CFFFFEFE  78BF7F7F  79DF7F7F  56803F7F  1A7F7F7F  0A7F7F7F  4A7F7F7F  3F7F7F7F  3FFF7F7F  3F9F7F7F  3F1F7F7F  0F7F7F7F  0F9F7F7F  0F1F7F7F  327FFF80  307FFF80  35003F80  217FFFFF  207FFFFF  25003F7F  25203F7F  589F7F7F  58BF7F7F  58DF7F7F  EFFFFEFE  DFFFFEFE  FFFFFEFE  785F7F7F  795F7F7F  585F7F7F  595F7F7F  00000000  00200000  40200000 |

## 

## **Local Store Fill Test**

Opens a waveform showing the result of filling up the Local Store with instructions. The example file show [above](#_OUTPUT) is used to demonstrate.



The ***SRAM*** signal is shown being populated (one instruction per cycle). In normal operation of the processor, the Local Storage is filled with all instructions simultaneously.

## **Cache Miss Test**

This test populates the Instruction Cache with instructions (32) from the Local Store, when a Cache miss occurs. Again, the example file [above](#_OUTPUT) is used to demonstrate.



Firstly,***@ 35-50ns***the ***LS\_RIB\_OUT\_EOP***signal is set to let the Local Store know that the Read Instruction Block (RIB) instruction has arrived and to return a 128-Byte Block of instructions (32 4-Byte instructions). After 6 cycles (the latency for accessing the local store),***@ 110ns*** the ***CACHE*** signal is seen to be populated by the instructions sent to it. The Cache is a 128-Byte Direct-mapped Cache (2 Instructions per entry (8-Bytes), with 16 entries). After 1 cycle, ***@ 120ns*** the ***HIT*** signal is set, signifying a Cache Hit and is ready to begin processing Instruction pairs.

## **Dual Instruction – Fetch\_Decode\_Issue\_Execute (No Hazard) Test**

This test demonstrates the typical Dual Fetch 🡪 Decode 🡪 Issue 🡪 Execute cycle. No type of Hazards are introduced.

## **Structural Hazard Resolution Test**

## **Data Hazard Resolution by Forwarding (No Stalling) Test**

The forwarding was evaluated next using the following test cases:

1. When one operand has a dependency
2. When multiple operands have the same dependency
3. When multiple operands have different dependencies
4. Case 1-3 with both Even and Odd Instructions

## **Data Hazard Resolution by Stalling and Forwarding Test**

## **Control Hazard Resolution for Branches Test**

# **Appendix A – ISA**

## **Simple-Fixed 1 Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Add Word** | *a rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Add Word Immediate** | *ai rt, ra, value* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Subtract from Word** | *sf rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Subtract from Word Immediate** | *sfi rt, ra, value* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Count Leading Zeros** | *clz rt, ra* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **AND** | *and rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **AND with Complement** | *andc rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **AND Word Immediate** | *andi rt, ra, value* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **OR** | *or rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **OR with Complement** | *orc rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **OR Word Immediate** | *ori rt, ra, value* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **OR Across** | *orx rt, ra* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Exclusive OR** | *xor rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **NAND** | *nand rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **NOR** | *nor rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Compare Equal Word** | ceq rt, ra, rb | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Compare Equal Word Immediate** | *ceqi rt, ra, value* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Compare Greater Than Word** | *cgt rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Compare Greater Than Word Immediate** | *cgti rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Compare Logical Greater Than Word** | *clgt rt, ra, rb* | **Even** | **Simple Fixed 1** | **2** | **2** |
| **Compare Logical Greater Than Word Immediate** | *clgti rt, ra, value* | **Even** | **Simple Fixed 1** | **2** | **2** |

## **Simple-Fixed 2 Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Shift Left Word Immediate** | *shli rt, ra, value* | **Even** | **Simple Fixed 2** | **3** | **4** |
| **Rotate Halfword Immediate** | *rothi rt, ra, value* | **Even** | **Simple Fixed 2** | **3** | **4** |
| **Rotate Word Immediate** | *roti rt, ra, value* | **Even** | **Simple Fixed 2** | **3** | **4** |

## **Floating Point Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Floating Add** | *fa rt, ra, rb* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Subtract** | *fs rt, ra, rb* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Multiply** | *fm rt, ra, rb* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Multiply and Add** | *fma rt, ra, rb, rc* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Negative Multiply and Subtract** | *fnms rt, ra, rb, rc* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Multiply and Subtract** | *fms rt, ra, rb ,rc* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Compare Equal** | *fceq rt, ra, rb* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Compare Magnitude Equal** | *fcmeq rt, ra, rb* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Compare Greater Than** | *fcgt rt, ra, rb* | **Even** | **Single Precision FP** | **6** | **6** |
| **Floating Compare Magnitude Greater Than** | *fcmgt rt, ra, rb* | **Even** | **Single Precision FP** | **6** | **6** |

## **Floating Point Integer Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Multiply** | *mpy rt, ra, rb* | **Even** | **FP Integer** | **7** | **7** |
| **Multiply Unsigned** | *mpyu rt, ra, rb* | **Even** | **FP Integer** | **7** | **7** |
| **Multiply Immediate** | *mpyi rt, ra, value* | **Even** | **FP Integer** | **7** | **7** |
| **Multiply Unsigned Immediate** | *mpyui rt, ra, value* | **Even** | **FP Integer** | **7** | **7** |
| **Multiply and Add** | *mpya rt, ra, rb, rc* | **Even** | **FP Integer** | **7** | **7** |
| **Multiply High** | *mpyh rt, ra, rb* | **Even** | **FP Integer** | **7** | **7** |
| **Multiply High High Unsigned** | *mpyhhu rt, ra, rb* | **Even** | **FP Integer** | **7** | **7** |

## **Byte Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Count Ones in Bytes** | *cntb rt, ra* | **Even** | **Byte** | **3** | **4** |
| **Average Bytes** | *avgb rt, ra, rb* | **Even** | **Byte** | **3** | **4** |
| **Absolute Differences of Bytes** | *absdb rt, ra, rb* | **Even** | **Byte** | **3** | **4** |
| **Sum Bytes into Halfwords** | *sumb rt, ra, rb* | **Even** | **Byte** | **3** | **4** |

## **Local Store Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Load Quadword**  **(a–form)** | *lqa rt, symbol* | **Odd** | **LS** | **6** | **6** |
| **Load Quadword**  **(d-form)** | *lqd rt, symbol(ra)* | **Odd** | **LS** | **6** | **6** |
| **Store Quadword**  **(d-form)** | *stqd rt, symbol(ra)* | **Odd** | **LS** | **6** | **6** |
| **Store Quadword**  **(a-form)** | *stqa rt, symbol* | **Odd** | **LS** | **6** | **6** |
| **Read Instruction Block** | *rib* | **Odd** | **LS** | **6** | **6** |
| **Immediate Load Half Word Upper** | *ilhu rt, symbol* | **Odd** | **LS** | **6** | **6** |
| **Immediate Load Word** | *il rt, symbol* | **Odd** | **LS** | **6** | **6** |
| **Immediate load Address** | *ila rt, symbol* | **Odd** | **LS** | **6** | **6** |
| **Immediate OR Halfword Lower** | *iohl rt, symbol* | **Odd** | **LS** | **6** | **6** |

## **Permute Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Shift Left Quadword by Bits Immediate** | *shlqbii rt, ra, value* | **Odd** | **Permute** | **3** | **4** |
| **Shift Left Quadword by Bytes Immediate** | *shlqbyi rt, ra, value* | **Odd** | **Permute** | **3** | **4** |
| **Rotate Quadword by Bytes Immediate** | *rotqbyi rt, ra, value* | **Odd** | **Permute** | **3** | **4** |
| **Rotate Quadword by Bits Immediate** | *rotqbii rt, ra, value* | **Odd** | **Permute** | **3** | **4** |

## **Branch Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Branch Relative** | *br symbol* | **Odd** | **Branch** | **3** | **4** |
| **Branch Absolute** | *bra symbol* | **Odd** | **Branch** | **3** | **4** |
| **Branch Indirect** | *bi ra* | **Odd** | **Branch** | **3** | **4** |
| **Branch If Not Zero Word** | *brnz rt, symbol* | **Odd** | **Branch** | **3** | **4** |
| **Branch If Zero Word** | *brz rt, symbol* | **Odd** | **Branch** | **3** | **4** |
| **Branch Indirect If Zero** | *biz rt, ra* | **Odd** | **Branch** | **3** | **4** |
| **Branch Indirect If Not Zero** | *binz rt, ra* | **Odd** | **Branch** | **3** | **4** |

## **Control Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Name** | **Assembly Code** | **Lane**  **(Execution Pipe)** | **Unit** | **Number of Stages (Pipeline Depth)** | **Instruction Latency** |
| **Stop and Signal** | *stop* | **-** | **-** | **-** | **-** |
| **Nop (Load)** | *lnop* | **Odd** | **-** | **-** | **-** |
| **Nop (Execute)** | *nop* | **Even** | **-** | **-** | **-** |

**Total Instructions: 68**

# **Appendix B – Assembler Code**

***include/***

#include <stdio.h>

#include <stdlib.h>

#include "assembler.h"

#include "sort.h"

#include "isa.h"

/\*

\* Program reads, from a file passed to stdin, Cell

\* SPU ASM Instructions and, outputs to a file, the binary

\* data in the format associated with each instruction.

\*/

*int* main() {

*char* \*instr = (*char* \*)malloc(MAX\_INSTR\_LEN); // Instruction String

*unsigned* \*bdata = NULL; // Binary Data

/\* Sort Instruction Table \*/

sort\_instr();

for(;;) {

/\* Get next Instruction Line \*/

if((getInstr(instr, MAX\_INSTR\_LEN))) break; // EOF reached

/\* Skip comments \*/

if(instr[0] == '#') continue;

/\* Parse the obtained Instruction String \*/

if(parse\_line(instr, &bdata)) {

puts("Empty String!");

free(instr);

return EXIT\_FAILURE;

}

/\* Output the Binary data \*/

output\_data(bdata);

}

/\* Free the instruction line buffer \*/

free(instr);

/\* If End of File reached \*/

if(feof(stdin))

return EXIT\_SUCCESS;

else if(ferror(stdin))

return EXIT\_FAILURE;

return EXIT\_FAILURE;

}

#ifndef ISA\_H

#define ISA\_H

/\* Total number of Instructions \*/

#define TOTAL\_INSTR 68

/\* Instructin Formats \*/

*typedef* *enum* {

RR, // RR-Type

RRR, // RRR-Type

RI7, // RI7-Type

RI10, // RI10-Type

RI16, // RI16-Type

RI18 // RI18-Type

} format;

/\* Binary Instruction Arrangement \*/

*typedef* *union* {

*struct* {

*unsigned* RT: 7; // Operand Register T

*unsigned* RA: 7; // Operand Register A

*unsigned* RB: 7; // Operand Register B

const *unsigned* op: 11; // Unique Instruction Op-code

} RR\_bf;

*struct* {

*unsigned* RA: 7;

*unsigned* RB: 7;

*unsigned* RC: 7; // Operand Register C

*unsigned* RT: 7;

const *unsigned* op: 4;

} RRR\_bf;

*struct* {

*unsigned* RT: 7;

*unsigned* RA: 7;

*unsigned* RI7: 7; // Operand Immediate 7 bits

const *unsigned* op: 11;

} RI7\_bf;

*struct* {

*unsigned* RT: 7;

*unsigned* RA: 7;

*unsigned* RI10: 10; // Operand Immediate 10 bits

const *unsigned* op: 8;

} RI10\_bf;

*struct* {

*unsigned* RT: 7;

*unsigned* RI16: 16; // Operand Immediate 16 bits

const *unsigned* op: 9;

} RI16\_bf;

*struct* {

*unsigned* RT: 7;

*unsigned* RI18: 18; // Operand Immediate 18 bits

const *unsigned* op: 7;

} RI18\_bf;

*unsigned* instr; // Full 32-bit Instruction

} bin\_format;

/\* ISA - Instruction Table \*/

*typedef* *struct* {

const format f; /\* Instruction format \*/

const *char* \*instr\_name; /\* Instruction Name \*/

bin\_format bf; /\* Instruction Binary Arrangement \*/

} isa;

extern isa isa\_instr[TOTAL\_INSTR];

#endif /\* ISA\_H \*/

#ifndef SORT\_H

#define SORT\_H

/\*

\* Details: Sort the Instructions array according to their names.

\*/

*void*

sort\_instr();

#endif /\* SORT\_H \*/

***src/***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include "assembler.h"

#include "isa.h"

#include "sort.h"

/\*

\* Details: Read one line (Instruction & operands) using getline().

\*

\* Parameter(s):

\* - bytes\_read: number of bytes read

\* - size: number of bytes to read

\*

\* Returns:

\* - 0: If line was read successfully

\* - 1: If failure or EOF

\*/

*int*

getInstr(*char* \*instr, *size\_t* size) {

*size\_t* buffer\_size = 0;

/\* Read Instruction line from stdin. \*/

if((buffer\_size = getline(&instr, &size, stdin)) == EOF) return 1;

/\* Get rid of newline at the end of the line read \*/

if(instr[buffer\_size-1] == '\n') {

instr[buffer\_size-1] = '\0';

}

return 0;

}

/\*

\* Details: Output the Instruction binary data to stdout

\*

\* Parameter(s):

\* - instr\_data: Binary data to output

\*/

*void*

output\_data(*unsigned* \*instr\_data) {

printf("%08X\n", \*instr\_data);

}

/\*

\* Details: Comparater function used for qsort and bsearch.

\* Compares the String Instructions names in the isa\_instr table.

\*

\* Parameter(s):

\* - p1: Pointer to 1st isa Struct being compared

\* - p2: Pointer to 2nd isa Struct being compared

\*

\* Returns:

\* - 0: p1 == p2

\* - < 0: The stopping character in p1 was less than the stopping character in p2

\* - > 0: The stopping character in p1 was greater than the stopping character in p2

\*/

*int*

cmp(const *void* \*p1, const *void* \*p2) {

return strcmp(((isa \*)p1)->instr\_name, ((isa \*)p2)->instr\_name);

}

/\*

\* Details: Read the Operand String, convert them

\* to unsigned integer and fill the necessary values

\* needed for the RR format Instruction.

\*

\* Parameter(s):

\* - instr\_ptr: Pointer to current instruction being

\* processed.

\*/

static *void*

RR\_proc(isa \*instr\_ptr) {

*char* \*token;

/\* Take care of special cases \*/

if(!strcmp(instr\_ptr->instr\_name, "bi")) {

/\* Read Operand RA \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RR\_bf.RA = (*unsigned*)strtol(token, NULL, 0);

return;

} else if(!strcmp(instr\_ptr->instr\_name, "stop") ||

!strcmp(instr\_ptr->instr\_name, "lnop") ||

!strcmp(instr\_ptr->instr\_name, "nop") ||

!strcmp(instr\_ptr->instr\_name, "rib")) {

return; // These Instructions does not have any operands

}

/\* Read Operand RT \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RR\_bf.RT = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand RA \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RR\_bf.RA = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand RB \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RR\_bf.RB = (*unsigned*)strtol(token, NULL, 0);

}

/\*

\* Details: Read the Operand String, convert them

\* to unsigned integer and fill the necessary values

\* needed for the RRR format Instruction.

\*

\* Parameter(s):

\* - instr\_ptr: Pointer to current instruction being

\* processed.

\*/

static *void*

RRR\_proc(isa \*instr\_ptr) {

*char* \*token;

/\* Read Operand RT \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RRR\_bf.RT = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand RA \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RRR\_bf.RA = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand RB \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RRR\_bf.RB = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand RC \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RRR\_bf.RC = (*unsigned*)strtol(token, NULL, 0);

}

/\*

\* Details: Read the Operand String, convert them

\* to unsigned integer and fill the necessary values

\* needed for the RI7 format Instruction.

\*

\* Parameter(s):

\* - instr\_ptr: Pointer to current instruction being

\* processed.

\*/

static *void*

RI7\_proc(isa \*instr\_ptr) {

*char* \*token;

/\* Read Operand RT \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI7\_bf.RT = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand RA \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI7\_bf.RA = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand I7 \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI7\_bf.RI7 = (*unsigned*)strtol(token, NULL, 0);

}

/\*

\* Details: Read the Operand String, convert them

\* to unsigned integer and fill the necessary values

\* needed for the RI10 format Instruction.

\*

\* Parameter(s):

\* - instr\_ptr: Pointer to current instruction being

\* processed.

\*/

static *void*

RI10\_proc(isa \*instr\_ptr) {

*char* \*token;

/\* Take care of special cases \*/

if(!strcmp(instr\_ptr->instr\_name, "lqd") ||

!strcmp(instr\_ptr->instr\_name, "stqd")) {

/\* Read Operand RT \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI10\_bf.RT = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand I10 \*/

if(!(token = strtok(NULL, " ,("))) return;

instr\_ptr->bf.RI10\_bf.RI10 = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand RA \*/

if(!(token = strtok(NULL, " ,()"))) return;

instr\_ptr->bf.RI10\_bf.RA = (*unsigned*)strtol(token, NULL, 0);

return;

}

/\* Read Operand RT \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI10\_bf.RT = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand RA \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI10\_bf.RA = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand I10 \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI10\_bf.RI10 = (*unsigned*)strtol(token, NULL, 0);

}

/\*

\* Details: Read the Operand String, convert them

\* to unsigned integer and fill the necessary values

\* needed for the RI16 format Instruction.

\*

\* Parameter(s):

\* - instr\_ptr: Pointer to current instruction being

\* processed.

\*/

static *void*

RI16\_proc(isa \*instr\_ptr) {

*char* \*token;

/\* Take care of special case \*/

if(!strcmp(instr\_ptr->instr\_name, "br") ||

!strcmp(instr\_ptr->instr\_name, "bra")) {

/\* Read Operand I16 \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI16\_bf.RI16 = (*unsigned*)strtol(token, NULL, 0);

return;

}

/\* Read Operand RT \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI16\_bf.RT = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand I16 \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI16\_bf.RI16 = (*unsigned*)strtol(token, NULL, 0);

}

/\*

\* Details: Read the Operand String, convert them

\* to unsigned integer and fill the necessary values

\* needed for the RI18 format Instruction.

\*

\* Parameter(s):

\* - instr\_ptr: Pointer to current instruction being

\* processed.

\*/

static *void*

RI18\_proc(isa \*instr\_ptr) {

*char* \*token;

/\* Read Operand RT \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI18\_bf.RT = (*unsigned*)strtol(token, NULL, 0);

/\* Read Operand I18 \*/

if(!(token = strtok(NULL, " ,"))) return;

instr\_ptr->bf.RI18\_bf.RI18 = (*unsigned*)strtol(token, NULL, 0);

}

/\*

\* Details: Takes in the given String and convert

\* it to lowercase (regardless of orignal case).

\*

\* Parameter(s):

\* - str: String to make lower case

\*/

static *void*

lowercase(*char* \*str) {

for(;\*str;++str) \*str = tolower(\*str);

}

/\*

\* Details: Parses the instrution line retrieved using strtok

\* and outputs the binary code representing it.

\*

\* Parameter(s):

\* - instr: instruction line String

\* - dbata: Binary data to output

\*

\* Returns:

\* - 0: Successfully parsed String

\* - 1: Error

\*/

*int*

parse\_line(*char* \*instr, *unsigned* \*\*bdata) {

*char* \*token;

/\*

Intermediary isa Structure to hold the name

of the Instruction during search

\*/

isa target;

isa \*instr\_ptr; // Points to Instruction Structure currently being processed

/\* Get Instruction name \*/

if(!(token = strtok(instr, " "))) return 1; // Return Error if Empty String

lowercase(token);

/\* Search for Instruction Structure given the Insruction Name \*/

/\* Find Instruction \*/

target.instr\_name = token;

instr\_ptr = (isa \*)bsearch(&target, isa\_instr, TOTAL\_INSTR,

sizeof(isa), cmp);

/\* Get Instruction Operands accodring to Instruction format \*/

switch(instr\_ptr->f) {

case RR:

RR\_proc(instr\_ptr);

break;

case RRR:

RRR\_proc(instr\_ptr);

break;

case RI7:

RI7\_proc(instr\_ptr);

break;

case RI10:

RI10\_proc(instr\_ptr);

break;

case RI16:

RI16\_proc(instr\_ptr);

break;

case RI18:

RI18\_proc(instr\_ptr);

break;

}

/\* Get the data value \*/

\*bdata = &instr\_ptr->bf.instr;

return 0;

}

#include "isa.h"

/\* ISA - Instruction Table \*/

isa isa\_instr[TOTAL\_INSTR] = {

{.instr\_name = "lqa", .bf.RI16\_bf.op = 0x61, .f = RI16}, // Load Quadword (a-form)

{.instr\_name = "lqd", .bf.RI10\_bf.op = 0x34, .f = RI10}, // Load Quadword (d-form)

{.instr\_name = "stqd", .bf.RI10\_bf.op = 0x24, .f = RI10}, // Store Quadword (d-form)

{.instr\_name = "stqa", .bf.RI16\_bf.op = 0x41, .f = RI16}, // Store Quadword (a-form)

{.instr\_name = "rib", .bf.RR\_bf.op = 0x23, .f = RR}, // Read Instruction Block

{.instr\_name = "ilhu", .bf.RI16\_bf.op = 0x82, .f = RI16}, // Immediate Load Halfword Upper

{.instr\_name = "il", .bf.RI16\_bf.op = 0x81, .f = RI16}, // Immediate Load Word

{.instr\_name = "ila", .bf.RI18\_bf.op = 0x21, .f = RI18}, // Immediate Load Address

{.instr\_name = "iohl", .bf.RI16\_bf.op = 0xC1, .f = RI16}, // Immediate OR Halfword Lower

{.instr\_name = "a", .bf.RR\_bf.op = 0xC0, .f = RR}, // Add Word

{.instr\_name = "ai", .bf.RI10\_bf.op = 0x1C, .f = RI10}, // Add Word Immediate

{.instr\_name = "sf", .bf.RR\_bf.op = 0x40, .f = RR}, // Subtract from Word

{.instr\_name = "sfi", .bf.RI10\_bf.op = 0x0C, .f = RI10}, // Subtract from Word Immediate

{.instr\_name = "clz", .bf.RR\_bf.op = 0x2A5, .f = RR}, // Count Leading Zeros

{.instr\_name = "and", .bf.RR\_bf.op = 0xC1, .f = RR}, // And

{.instr\_name = "andc", .bf.RR\_bf.op = 0x2C1, .f = RR}, // And with Complement

{.instr\_name = "andi", .bf.RI10\_bf.op = 0x14, .f = RI10}, // And Word Immediate

{.instr\_name = "or", .bf.RR\_bf.op = 0x41, .f = RR}, // Or

{.instr\_name = "orc", .bf.RR\_bf.op = 0x2C9, .f = RR}, // Or with Complement

{.instr\_name = "ori", .bf.RI10\_bf.op = 0x4, .f = RI10}, // Or Word Immediate

{.instr\_name = "orx", .bf.RR\_bf.op = 0x1f0, .f = RR}, // Or Across

{.instr\_name = "xor", .bf.RR\_bf.op = 0x241, .f = RR}, // Exclusive Or

{.instr\_name = "nand", .bf.RR\_bf.op = 0xC9, .f = RR}, // Nand

{.instr\_name = "nor", .bf.RR\_bf.op = 0x49, .f = RR}, // Nor

{.instr\_name = "ceq", .bf.RR\_bf.op = 0x3C0, .f = RR}, // Compare Equal Word

{.instr\_name = "ceqi", .bf.RI10\_bf.op = 0x7C, .f = RI10}, // Compare Equal Word Immediate

{.instr\_name = "cgt", .bf.RR\_bf.op = 0x240, .f = RR}, // Compare Greater Than Word

{.instr\_name = "cgti", .bf.RI10\_bf.op = 0x4C, .f = RI10}, // Compare Greater Than Word Immediate

{.instr\_name = "clgt", .bf.RR\_bf.op = 0x2C0, .f = RR}, // Compare Logical Greater Than Word

{.instr\_name = "clgti", .bf.RI10\_bf.op = 0x5C, .f = RI10}, // Compare Logical Greater Than Word Immediate

{.instr\_name = "mpy", .bf.RR\_bf.op = 0x3C4, .f = RR}, // Multiply

{.instr\_name = "mpyu", .bf.RR\_bf.op = 0x3CC, .f = RR}, // Multiply Unsigned

{.instr\_name = "mpyi", .bf.RI10\_bf.op = 0x74, .f = RI10}, // Multiply Immediate

{.instr\_name = "mpyui", .bf.RI10\_bf.op = 0x75, .f = RI10}, // Multiply Unsigned Immediate

{.instr\_name = "mpya", .bf.RRR\_bf.op = 0xC, .f = RRR}, // Multiply and Add

{.instr\_name = "mpyh", .bf.RR\_bf.op = 0x3C5, .f = RR}, // Multiply High

{.instr\_name = "mpyhhu", .bf.RR\_bf.op = 0x3CE, .f = RR}, // Multiply High High Unsigned

{.instr\_name = "cntb", .bf.RR\_bf.op = 0x2B4, .f = RR}, // Count Ones in Bytes

{.instr\_name = "avgb", .bf.RR\_bf.op = 0xD3, .f = RR}, // Average Bytes

{.instr\_name = "absdb", .bf.RR\_bf.op = 0x53, .f = RR}, // Absolute Differences of Bytes

{.instr\_name = "sumb", .bf.RR\_bf.op = 0x253, .f = RR}, // Sum Bytes into Halfwords

{.instr\_name = "shlqbii", .bf.RI7\_bf.op = 0x1FB, .f = RI7}, // Shift Left Quadword by Bits Immediate

{.instr\_name = "shlqbyi", .bf.RI7\_bf.op = 0x1FF, .f = RI7}, // Shift Left Quadword by Bytes Immediate

{.instr\_name = "rotqbyi", .bf.RI7\_bf.op = 0x1FC, .f = RI7}, // Rotate Quadword by Bytes Immediate

{.instr\_name = "rotqbii", .bf.RI7\_bf.op = 0x1F8, .f = RI7}, // Rotate Quadword by Bits Immediate

{.instr\_name = "shli", .bf.RI7\_bf.op = 0x7B, .f = RI7}, // Shift Left Word Immediate

{.instr\_name = "rothi", .bf.RI7\_bf.op = 0x7C, .f = RI7}, // Rotate Halfword Immediate

{.instr\_name = "roti", .bf.RI7\_bf.op = 0x78, .f = RI7}, // Rotate Word Immediate

{.instr\_name = "br", .bf.RI16\_bf.op = 0x64, .f = RI16}, // Branch Relative

{.instr\_name = "bra", .bf.RI16\_bf.op = 0x60, .f = RI16}, // Branch Absolute

{.instr\_name = "bi", .bf.RR\_bf.op = 0x1A8, .f = RR}, // Branch Indirect

{.instr\_name = "brnz", .bf.RI16\_bf.op = 0x42, .f = RI16}, // Branch If Not Zero Word

{.instr\_name = "brz", .bf.RI16\_bf.op = 0x40, .f = RI16}, // Branch If Zero Word

{.instr\_name = "biz", .bf.RR\_bf.op = 0x128, .f = RR}, // Branch Indirect If Zero

{.instr\_name = "binz", .bf.RI7\_bf.op = 0x129, .f = RI7}, // Branch Indirect If Not Zero

{.instr\_name = "fa", .bf.RR\_bf.op = 0x2C4, .f = RR}, // Floating Add

{.instr\_name = "fs", .bf.RR\_bf.op = 0x2C5, .f = RR}, // Floating Subtract

{.instr\_name = "fm", .bf.RR\_bf.op = 0x2C6, .f = RR}, // Floating Multiply

{.instr\_name = "fma", .bf.RRR\_bf.op = 0xE, .f = RRR}, // Floating Multiply and Add

{.instr\_name = "fnms", .bf.RRR\_bf.op = 0xD, .f = RRR}, // Floating Negative Multiply and Subtract

{.instr\_name = "fms", .bf.RRR\_bf.op = 0xF, .f = RRR}, // Floating Multiply and Subtract

{.instr\_name = "fceq", .bf.RR\_bf.op = 0x3C2, .f = RR}, // Floating Compare Equal

{.instr\_name = "fcmeq", .bf.RR\_bf.op = 0x3CA, .f = RR}, // Floating Compare Magnitude Equal

{.instr\_name = "fcgt", .bf.RR\_bf.op = 0x2C2, .f = RR}, // Floating Compare Greater Than

{.instr\_name = "fcmgt", .bf.RR\_bf.op = 0x2CA, .f = RR}, // Floating Compare Magnitude Greater Than

{.instr\_name = "stop", .bf.RR\_bf.op = 0x0, .f = RR}, // Stop and Signal

{.instr\_name = "lnop", .bf.RR\_bf.op = 0x1, .f = RR}, // No Operation (Load)

{.instr\_name = "nop", .bf.RR\_bf.op = 0x201, .f = RR} // No Operation (Execute)

};

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "isa.h"

#include "assembler.h"

/\*

\* Details: Sort the Instructions array according to their names.

\*/

*void*

sort\_instr() {

qsort(isa\_instr, TOTAL\_INSTR, sizeof(isa), cmp);

}

#include <stdio.h>

#include <stdlib.h>

#include "assembler.h"

#include "sort.h"

#include "isa.h"

/\*

\* Program reads, from a file passed to stdin, Cell

\* SPU ASM Instructions and, outputs to a file, the binary

\* data in the format associated with each instruction.

\*/

*int* main() {

*char* \*instr = (*char* \*)malloc(MAX\_INSTR\_LEN); // Instruction String

*unsigned* \*bdata = NULL; // Binary Data

/\* Sort Instruction Table \*/

sort\_instr();

for(;;) {

/\* Get next Instruction Line \*/

if((getInstr(instr, MAX\_INSTR\_LEN))) break; // EOF reached

/\* Skip comments \*/

if(instr[0] == '#') continue;

/\* Parse the obtained Instruction String \*/

if(parse\_line(instr, &bdata)) {

puts("Empty String!");

free(instr);

return EXIT\_FAILURE;

}

/\* Output the Binary data \*/

output\_data(bdata);

}

/\* Free the instruction line buffer \*/

free(instr);

/\* If End of File reached \*/

if(feof(stdin))

return EXIT\_SUCCESS;

else if(ferror(stdin))

return EXIT\_FAILURE;

return EXIT\_FAILURE;

}

# **Appendix C – Module Code**

***SPU Core Top Module***

***Register File***

***Local Store***

***Forwarding Macro/Circuits***

***Even & Odd Pipeline***

***Component Library***

***ISA Library***

# **Appendix D – Testbench Code**

***SPU Core Top Module***

# **References**

1. *Thomas Chen, Ram Raghavan, Jason Dale, Eiji Iwata. “Cell Broadband Engine Architecture and its First Implementation,” IBM. November 29, 2005. [Online]* [*https://www.ibm.com/developerworks/library/pacellperf/index.html*](https://www.ibm.com/developerworks/library/pacellperf/index.html)
2. *Flachs et al. (2006) “The Microarchitecture of the Synergistic Processor for a Cell Processor, “IEEE Journal of Solid-State Circuits (Volume: 41, Issue: 1, January 2006) [Online]*

<https://ieeexplore.ieee.org/document/1564346/metrics#metrics>