# Introduction

***Digital Design and Computer Architecture***

Kent Jones

# **MP 1: Microprocessor Language Design**

# 

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| **Exercise 1: Comparing Processors** |  | 20 |
| **Exercise 2: Processor Language Design** |  | 20 |
| **Exercise 3: Assembler for Your Processor** |  | 20 |
| **Exercise 4: Assembly Code and Machine Code** |  | 20 |
| **Exercise 5: Code Walkthrough and Presentation** |  | 20 |
| **TOTAL** |  | 100 |

# 

**For the first half of MP1** you will start by comparing two real world processors. You will compare their intended purpose, their instruction sets, their architecture, performance etc. This step will help you understand other possibilities for instruction sets and instruction set formats. It will also help you understand the architecture behind real world processors. This first part may be more challenging if you pick an obscure processor or a proprietary processor that does not have much information available about it online.

Fill out the answers to your work directly in this document and submit it on blackboard on the due date.

**For the second half of MP1**

1. You will decide on what type of processor you want to target (e.g. general purpose or special purpose dedicated processor: e.g. graphics, a.i., dsp, etc.)
2. You will design a new, original instruction set (i.e. assembly language) for this type of processor that can run non-trivial programs.
3. You will then design the machine code format for your instruction set.
4. You will write an assembler program that uses your new, original, assembly language (for testing purposes).
5. You will create an assembler to convert text based assembly language into a hex code machine format.
6. You will “assemble” (i.e. convert) each of these programs into your machine code.

**Preview of future MP labs:** This lab sets the stage for the next project where you design the ALU and DPU for your original processor. **Your group must create an original processor / control unit / data path unit** and do NOT simply copy an existing processor design that you find out on the Internet.

# Processors of the World Comparison

Compare the hardware architectures and assembly/machine languages for any two different processors (groups of two) and three different processors (groups of 3) of your choice. Decide on the specific application area you want your processor to focus on. Are you interested in general purpose? audio? graphics? encryption? security? We suggest that you pick at least one processer that you are interested in learning more about. Below are some suggestions that have information for them online. Each person in the group is responsible for ONE different processor and will be graded on that individually.

1. Old Style Intel 8080 or 8085 Architecture
   1. <http://en.wikipedia.org/wiki/Intel_8080>
   2. <http://www.intel-vintage.info/intelotherresources.htm#906748189>
2. Old Style Motorola 6502 Architecture
   1. <http://www.visual6502.org/welcome.html>
   2. <http://en.wikipedia.org/wiki/MOS_Technology_6502>
   3. <http://opencores.org/project,t65>
3. Modern General Purpose ARM Architecture
   1. <http://en.wikipedia.org/wiki/ARM_architecture>
   2. <https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf>
4. Modern Application Specific NVIDIA Architecture (Graphics Processor)
5. Modern AMD Architecture (e.g. Ryzen)
6. Other processor of your own choice (other than MIPS) that you can find information for.

#### Comparing Processors

1. Identify the application area of each processor you are comparing (e.g. is it a general purpose or special purpose processor? If special purpose, what is the application area?) Be sure and put the name of the person that did the research for each processor. Each person in the group is responsible for one processor. You may edit each other’s processor reports.   
     
   Tersa:  
   Intel 8086 bit microporcessor  
   General purpose register.  
     
   Amon :  
   NVIDIA Turing GPU  
   Special purpose processor.  
   Graphics processor with real-time ray tracing.
2. Register Set Architecture for each processor.

Table 1A: Processor A Register Set: Define the major registers and their names. Make a table similar to Table 6.1 MIPS Register Set on page 300 of your book

8 types of registers:

|  |  |
| --- | --- |
| **Name** | **Use** |
| General Purpose Registers: |  |
| AX  It is of 16 Bit, AH (8 Bit) and AL (8 Bit) | Accumulator. Arithmetical and logical instructions. |
| BX  It is of 16 Bit, BH (8 Bit) and BL (8 Bit) | Base register. Stores values of the offset |
| CX  It is of 16 Bit, CH (8 Bit) and CL (8 Bit) | Counter register. Loops and rotation |
| DX  It is of 16 Bit, DH (8 Bit) and DL (8 Bit) | Data register. Multiplication an input/output port addressing |
| Pointer and Index Registers: |  |
| SP  (16 Bits) | Stack pointer: Pointer to top of stack It offsets address relative to stack segment |
| BP  (16 Bits) | Base pointer. Pointer to base address. Used in accessing parameters passed by the stack, offsets address relative to stack segment |
| SI  (16 Bits) | Source index register. Source string/index pointer. Used in the pointer addressing of data and as a source in some string related operation, offsets relative to the data segment. |
| DI  (16 Bits) | Destination index register. Destination string/index pointer. Used in the pointer addressing of data and as a destination in some string related operation, offsets relative to extra segment. |
| Segment Registers: |  |
| CS | Code segment |
| DS | Data Segment |
| SS | Stack Segment |
| ES | Extra Segment |
| Other Registers: |  |
| Flags | Flags |
| IP | Instruction Pointer |

Table 1A: Processor B Register Set Table: Define the major registers and their names. Make a table similar to Table 6.1 MIPS Register Set on page 300 of your book  
  
Nvidia GPU:   
State Spaces

|  |  |  |
| --- | --- | --- |
| **State Space** | **Use** | **Access** |
| .reg | Registers. | r/w |
| .sreg | Special registers. Read only. | ro |
| .const | Read only memory. | ro |
| .global | Global memory. Shared by all threads. | r/w |
| .local | Local memory. Specific to each thread. | r/w |
| .param | Kernel parameters or function parameters. |  |
| .shared | Shared memory. | r/w |
| .tex | Global texture memory. | ro |

NVIDIA PTX supports virtual registers.

NVIDIA PTX uses state spaces rather than actual registers. Registers are not directly addressable. The .reg state space is used to create virtual registers that work as variables.

Special Registers  
Read only special registers (that are actually variables?).

|  |  |
| --- | --- |
| **Special Register** | **Use** |
| %tid | Thread ID within a CTA. |
| %ntid | Number of thread IDs per CTA. |
| %laneid | Lane ID. |
| %warpid | Warp ID. |
| %nwarpid | Number of warp IDs. |
| %ctaid | CTA ID within a grid. |
| %nctaid | Number of CTA IDs per grid. |
| %smid | SM ID. |
| %nsmid | Number of SM IDs. |
| %gridid | Grid ID. |
| %lanemask\_eq, %lanemask\_le, %lanemask\_lt, %lanemask\_ge, %lanemask\_gt | 32-bit mask with bits set in relation to lane number. |
| %clock, %clock\_hi, %clock64 | Unsigned cycle counter. |
| %pm0…%pm7 , %pm0\_64…%pm7\_64 | Performance monitoring counters. |
| %envreg0…%envreg31 | Driver-defined registers. |
| %globaltimer , %globaltimer\_lo , %globaltimer\_hi | Global nanosecond timer. |
| %total\_smem\_size | Total size of shared memory. |
| %dynamic\_smem\_size | Size of shared memory allocated dynamically. |

\* CTA :  A cooperative thread array (CTA) is a set of concurrent threads that execute the same kernel program.   
  
References:

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

1. Machine Code Format and Operation for four different types of instructions – Pick four instructions from each processor to compare.

Table 2A: Processor A Instructions and Binary Format (see appendix B and figs 6.10, 6.12 for examples)

* 1. A conditional instruction: show the binary format and what the instruction does.
  2. A mathematical instruction: show the binary format and what the instruction does
  3. A jump instruction: show the binary format and what the instruction does
  4. Some other instruction of your choice: show the binary format and what the instruction does

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Syntax** | **Use** |
| AAA | MOV AX, 15;  AAA; | Adjust after Addition |
| IMUL | IMUL BL; |  |
| JAE | JAE label; | Jump if first operand is Above or Equal to second operand |
| INC | MOV AL, 4 INC AL ; AL = 5 | Increment |

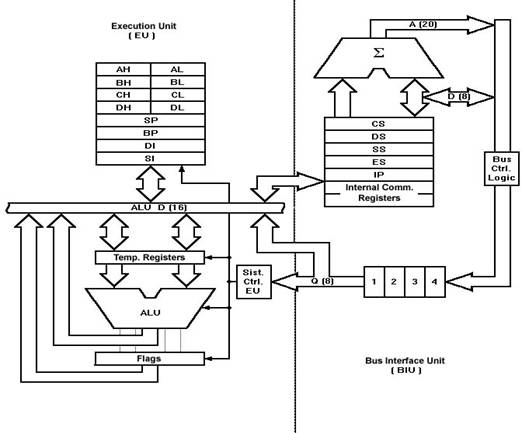
Table 2B: Processor B Instructions and Binary Format (see appendix B and figs 6.10, 6.12 for examples)

* 1. A conditional instruction: show the binary format and what the instruction does.
  2. A mathematical instruction: show the binary format and what the instruction does
  3. A jump instruction: show the binary format and what the instruction does
  4. Some other instruction of your choice: show the binary format and what the instruction does  
       
     NVIDIA GPU:

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Syntax** | **Semantic** | **Use** |
| MOV | MOV a, b; | a <- b | Move reg value. |
| IADD | IADD d, a, b; | d = a + b | Integer addition. |
| BRA | BRA tgt; | pc = tgt | Branch to target. |
| LD | LD d, [a]; | d = \*a | Load value. |

1. High level block diagram of each processor’s design and discussion – Compare and contrast the design of each processor your group chose. You may not be able to find the answer to all of the questions for both processors, do you best. I am expecting short concise answers for each of the questions to which you can find the answers. Each person in the group is responsible for one processor. You may however discuss and edit each other’s reports.

Processor A

* 1. : What does the data path look like for each processor? Find a high level architecture diagram that shows the data path for each processor. Insert these diagrams here: 
  2. What types of memory (register, cache, etc.) does each processor contain or access?

It has segment register, CS.DS,ES,SS all have different purposes. Temorary registers

* 1. How is the ALU connected to the registers (refer to the diagram in part a)?

The ALU is connected to temporary registers, flags and execution Unit

* 1. How are instructions fetched and executed? Is there an instruction cache?

The Bus Interface Unit (BIU) fetches the instruction from the memory and reads the data from the port and the memory as well as writing data to the ports and the memory . There is an instruction queue(IQ), the BIU contains the IQ and it gets up to 6 bytes of next instructions and it stores them in the IQ. The Execution Unit (EU), decodes the instructions from BIU. When the Execution Unit executes instructions the it becomes ready for the next instructions and it then reads the instruction from the IQ. In short BIU fetches, EU executes.

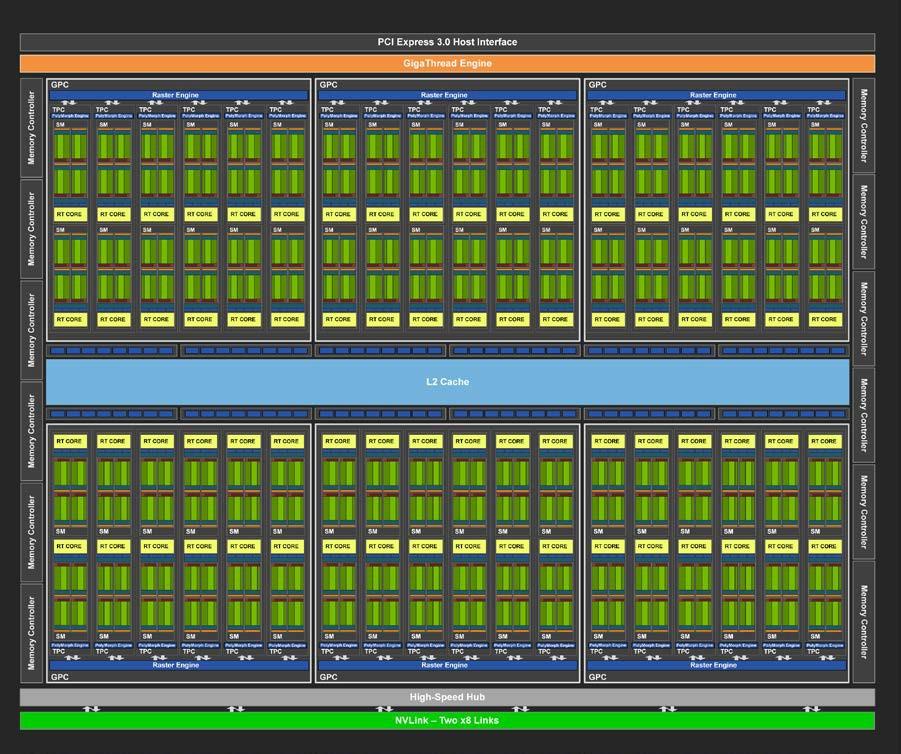
* 1. Does the processor pipeline instructions?

Yes, it fetches the next instruction while the current instruction executes

* 1. What is the clock speed of the processor?

It initially shipped at 5 MHz and later reached to 8MHz and 10MHz

Processor B

1. : What does the data path look like for each processor? Find a high level architecture diagram that shows the data path for each processor. Insert these diagrams here:  
     
     
   
2. What types of memory (register, cache, etc.) does each processor contain or access?   
   Register file, data cache, texture cache.
3. How is the ALU connected to the registers (refer to the diagram in part a)?   
   No idea.
4. How are instructions fetched and executed? Is there an instruction cache?   
   Warp scheduler.  
   Schedules operations to run on the points in the streaming multiprocessors.
5. Does the processor pipeline instructions?   
   Yes.
6. What is the clock speed of the processor?   
   Not disclosed. Estimated at 1730 MHz.  
   (https://www.anandtech.com/show/13214/nvidia-reveals-next-gen-turing-gpu-architecture)

#### Processor Language Design

Design the programmer’s view of the architecture for your processor (as a group). Just like in part 1, you will make two tables. The first table will be the registers and the second table the instructions for your processor.

* TABLE 2A: A diagram of the registers (and their purposes) for your custom processor

|  |  |
| --- | --- |
| **Register** | **Purpose** |
| %PC | Pointer to next instruction |
| %loopc | Loop counter |
| %r0 … %r9 | General registers |
| %xsys | System call register. (Like $v0 in mips) |
|  |  |
|  |  |

* TABLE 2B: A list of the instructions that you are going to build into the processor.
  + Don’t start with too many instructions!
  + Have some stretch instructions as well as basic instructions.
  + Each person in the group (after group consultation) must be personally responsible for at least one instruction.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Opcode** | **Binary** | **Syntax** | **Purpose** |
| Arithmetic 0001 | Add | 0001 0001 | Add c, a,b | Add |
| Sub | 0001 0010 | Sub c, a,b | Subtract |
| Mul | 0001 0011 | Mul c, a,b | Multiply |
| div | 0001 0100 | Div c, a,b | Divide |
| Mod | 0001 0101 | Mod c, a,b | Modulus |
| and | 0001 0110 | And c, a,b | And |
| or | 0001 0111 | Or c, a,b | Or |
| Data Control 0010 | ld | 0010 0001 | Ld r, <label> | Load from memory |
| St | 0010 0010 | St r, <label> | Store into memory |
| li | 0010 0011 | Li r, <imm> | Load immediate |
| Conditionals 0011 | jmp | 0011 0001 | jmp <imm> | jump |
| Move  0100 | mov | 0100 0001 | Mov r2, r1 | r2 = r1 |

* *Your processor will need enough instructions to do useful work* You must have enough instructions to be able to run a useful program (e.g. something of the complexity of generating the Fibonacci sequence)
* Check the evaluation rubric for how the number of instructions you implement will be evaluated.

#### Assembler for Your Processor

Create an assembler for your processor. Put your assembler code in your CS401 folder on CS1. Use pair / group programming to develop the assembler. Do not work individually.

You have several choices here…

1. Create a simple program that reads your assembly language program word by word and then converts each line to machine code. Here is some possible pseudo code for a simple assembler:

Read in the variable definitions from the top of the program.

For each variable in the definitions list:

Determine the size of memory required

Insert the variable into a dictionary that stores the assigned memory address of the variable.

Compute the memory location for the next variable.

For each line of assembly

NextHexCode = “”

Strip comments from the line of code read

If line of code has a label,

store memory location of the label in a dictionary

Determine the assembly keyword on the line of code

Based on keyword update NextHexCode

If line of code has arguments

Based on argument types (register, variable, immediate) and instruction type update NextHexCode

Write NextHexCode to the output machine code file.

1. Define a grammar for your assembly language and build a recursive descent parser (<https://en.wikipedia.org/wiki/Recursive_descent_parser> )
2. Use the utilities Lexx and Yacc (you will have to learn these on your own)

Include your assembly language code program here (make sure your program includes comments)

// test asm program

.data

var1: 5 7 8 9

var2: 98

.code

li %r1, 56 // li rr, aa

li %r2, 72 // comment...

li %r5, var2

add %r3, %r1, %r2 // add rr, rr, rr

sub %r4, %r2, %r1 // sub rr, rr, rr

jmp 32 // jmp iiiiii

mov %r3, %r1

mov %r4, %r2

#### Assembly Code Program and resulting Machine Code

Write an assembly language program using the language that you designed. Use your assembler from exercise 3 to compile the assembly language into hex based machine code.

Include your machine code program here:  
  
  
23010038

23020048

2305000a

11030102

12040201

31000020

41030001

41040002

#### Exercise 5: Design/Code Walkthrough MP 1 Presentations

On the due date for this design, groups of two will take 5 minutes (max) and present their assembly language designs to the class. You will be graded on whether you present the following items. **Do not use more than 4 or 5 slides to summarize your algorithm.**

Minute 1: What type of processor do you want to build and why?

Minute 2: What instruction set did you decide on and why?

Minute 3: How did you design your assembler? Did you allow for labels and variable definitions? Show your assembly language program and the assembled machine code file.

Minutes 4: Summary.

* What went well? What not so well?
* What issues did you run into in writing an assembler?
* Did you do anything above and beyond the requirements (not required)?

What to Hand In:

* Place all of the required items in this text document (i.e. word / pdf) called MP1\_Language\_Design. Submit this document on blackboard. Also be sure to save a copy in your folder on CS1.
* Be sure to check the evaluation rubric given here.
* In general A work will go above and beyond the basic requirements given in the document and rubric.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CATEGORY** | **Poor or missing attempt** | **Beginning** | **Satisfactory** | **Excellent** |
| **Exercise 1: Comparing Processors** | Missing or extremely poor quality. | Low quality comparison of processors. Only compared 1-2 instructions with limited discussion and questions answered poorly and instructions not followed. | Adequate comparison of both processors architecture, machine code, and assembly language with adequate comparison of 3 instructions and answers to majority of the questions. Both Table I and Table II completed adequately. | Excellent comparison of both processors architecture, machine code, and four or more assembly language instructions and format.  Includes high level circuit design (block diagrams) for both processors.  Includes an excellent comparison and discussion of pros and cons of each processor and answers to questions given in the problem description. |
| **Exercise 2: Assembly Language Design** | Missing or extremely poor quality. | 4 instructions total and 2 points for each part of each table and other answers. | 6 instructions total and 2 points for each part of each table and other answers. | 8 instructions total and 2 points for each part of each table and other answers. |
| **Exercise 3: Assembler** | Missing or extremely poor quality. | Hard coded assembler with few comments. No variables or labels (values are hard coded in the instructions) | Adequate assembler with one of either variables or labels. Adequately commented. | 1 well commented, comprehensive assembler program that supports labels and variables. |
| **Exercise 4: Assembly Language and Machine Code** | Missing or extremely poor quality. | 1 hex listing without assembly language | Adequate assembly language program and associated working machine code. | Excellent assembler with comments, excellent assembly language program |