Capua Environment Reference Manual

V1.0 January 2016

Document history

Date	Note
January 2016	Document creation.
February 2016	Update to architecture graphic and adding Clock and STMP device documentation.

Table of Contents

Introduction

The Capua environment is a virtual core designed to help new comers to assembly programming, and system programming in general, learn the basics associated with such concepts. Capua was developed after witnessing a diminution in the abilities of recent graduated students to evolve in low level programming environment. Therefore, Capua functionalities are stripped down to bare minimum so users can benefit from the simplicity of its design in a learning context.

Capua versus Spartacus

Spartacus is the whole virtual machine while Capua is the core on which the virtual machine (VM) is built. Spartacus provides everything that is required for execution of user code. It provides Capua (the core) with a memory mapped memory controller, a memory array and an IO controller. The whole architecture is based on memory mapped device. At time of this writing, Spartacus VM does not support any form of interruption. This has been left for future development.

Capua

Capua is a *load/store* architecture. This means that only load (**`MEMR`**) and store (**`MEMW`**) instructions can do memory access. Please note that stack related instruction also have access to the memory, however, in a much stricter way. Capua is a 32-bit core with 4 general purpose register, an instruction pointer register and a flag register. Of the 4 GPR, one can be used as a stack pointer register. In that case, that register can no longer be used as a GPR unless its value is, somehow, save in a known memory address before operating on it.

Registers

The general purpose registers are all 32-bit wide. These register are named **`A`**, **`B`**, **`C`** and **`S`**. The **`S`** register is the designated stack pointer register. The other GPR do not have special usage except for the **`MUL`** and **`DIV`** operations. More on this in the appropriate section. The important thing to note about GPRs is that they hold signed 32 bits integers.

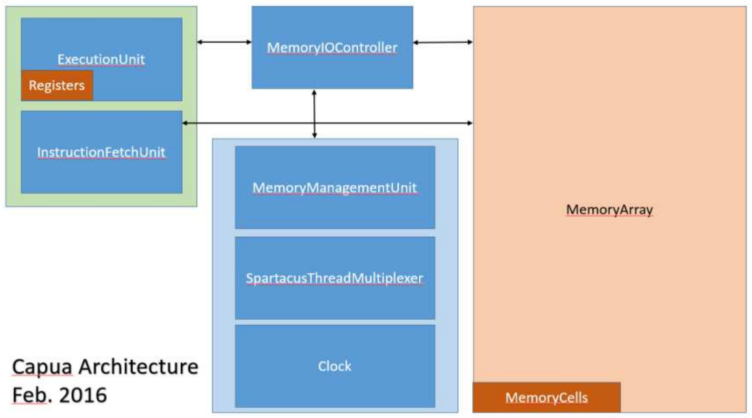
The instruction pointer is named **`I`** and is not user accessible. The flags pointer is named **`FLAGS`** and is also not user accessible. Those two registers are modified by instruction side effect. Please note that, at current time, there are no overflow flags. Therefore, if you require such functionality, you have to provide it yourself in your software.

Memory

At current time, no memory management unit is available on the virtual machine. Memory is, therefore, directly accessed. The start address of the memory is defined in **`Configuration.Configuration.MEMORY_START_AT`**. The default recommended value for this is **`0x40000000`**. The end of memory is defined as **`Configuration.Configuration.MEMORY_END_AT`**. By default, the virtual machine has 1 MB of memory. It is not recommended to change the memory start address as this could result in problems with memory mapped device. Memory mapped device are accessible at addresses bellow memory start address.

General architecture

The following graphic shows the general organisation of the VM design.



As you can see, the architecture is very simple but yet resembles that of a real computer.

Section 1 - Instruction set

General form

The Capua instruction set currently counts 21 different instructions. Most of these instructions can use either registers or immediate values (or a mix of these) in order to execute. Instructions are of variable length and therefore an invalid jump offset will result in an undefined behaviour depending on what is present in the memory where the jump instruction lands. For every instruction being executed, the first 4 bits of the instruction define the instruction form. The following 4 bits define a specific instruction within a form set. The following section elaborates on the various instruction form available in Capua architecture describing their binary properties.

Form 1 – Mnemonic (`0b1111`)

Form 1 describes instructions that do not use registers or immediate values. The **`NOP`** and **`RET`** instructions are example of this form. Since form 1 instructions only need a form and an instruction identifier, these instructions are 1-byte long.

0000	0000
Form	Instruction

Form 2 – Mnemonic Register (**`0b0111`**)

This form describes instructions that use a single register. The **`NOT`** and **`PUSH`** (when used with a register) are examples of this form. As with all instruction forms, the first byte describe the form and instruction codes. In the case of Form 2 instruction, the second byte of the instruction is used to specify the register being used in the operation.

0000	0000	0000 0000
Form	Instruction	Register

Form 3 – Mnemonic Immediate (**`0b1000`**)

This form is similar to Form 2 instructions. However, it uses an immediate instead of a register. The **`PUSH`** instruction is available in Form 3 as well as Form 2. Form 3 instructions are 5 bytes long.

0000	0000	0000 0000 0000 0000 0000 0000 0000 0000
Form	Instruction	Immediate (4 bytes)

Form 4 – Mnemonic Immediate Register (**`0b0110`**)

This form groups instructions like **`MOV`**, **`SHR`** and **`XOR`**. The immediate value in this form act as a source while the register is the destination. The operation is applied on the destination register value by using the immediate provided in the instruction. Instruction of this form are 6 bytes long.

0000	0000	0000 0000 0000 0000 0000 0000 0000 0000	0000 0000
Form	Instruction	Immediate (4 bytes)	Dest. register

Form 5 – Mnemonic Width Immediate Register (`0b0000`)

Instruction of this form are specifically used for memory access. Depending on the instruction being used, the immediate might be a memory address (when reading from memory) or a value (when writing to it). The register also has to hold appropriate data depending on the use case. The width field allow for specifying the data length that one wishes to either read of write to or from memory. Instruction of this form have the particularity that, for instruction encoding optimization, the destination register and immediate value are inverted in the encoded instruction. This allows these instructions to be 6 bytes long.

0000	0000	0000	0000	0000 0000 0000 0000 0000 0000 0000 0000
Form	Instruction	Width	D. Reg.	Immediate (4 bytes)

Form 6 – Mnemonic Width Register Register (**`0b0001`**)

Instructions of this form are, like Form 5, used for memory access. The difference is simply that the immediate is replaced by an additional register. Also, this form does not require inversion of field. These instructions are 2 bytes long.

0000	0000	0000	00	00
Form	Instruction	Width	S. Reg.	D. Reg

Form 7 – Mnemonic Width Register Immediate (**`0b0010`**)

This form is similar in its use as Form 5 and 6. The encoding for this instruction is identical to Form 5 encoding. However, the fields are in the right order.

0000	0000	0000	0000	0000 0000 0000 0000 0000 0000 0000 0000
Form	Instruction	Width	S. Reg.	Immediate (4 bytes)

Form 8 – Mnemonic Width Immediate Immediate (**`0b0011`**)

This form is also used in similar way as previous forms (5, 6, 7) but uses immediate values for both source and destination. Those instructions are 10 bytes long. This is the longest possible instruction type on this architecture.

0000	0000	0000	0000 0000 0000 0000 0000 0000 0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Form	Instruction	Width	S. Imm. (4 bytes)	D. Imm. (4 bytes)

Form 9 – Mnemonic Flag Immediate (`0b0100`)

This form is used for instructions that uses instruction modifier (flags). The **`JMP`** instruction is an example of this form. Instructions of this form are 6 bytes long.

0000	0000	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Form	Instruction	Flag	Immediate

Form 10 – Mnemonic Flag Register (`0b0101`)

This form is similar to Form 9. The difference is that it uses a register instead of an immediate value. The **`JMP`** instruction is also available in this form. Instructions of this form are 2 bytes long.

0000	0000	0000	0000
Form	Instruction	Flag	Immediate

Form 11 – Mnemonic Register Register (`0b1001`)

This is the last instruction form available under this architecture. The MOV instruction is available under Form 11. Instruction of this form are 2 bytes long.

0000	0000	0000	0000
Form	Instruction	S. Register	D. Register

Instructions description

The syntax for the Capua assembly language is based on the source destination notation. When 2 elements, either immediate of registers, are present in an instruction, the first is always the source and the second one is always the destination. Following is the full instruction listing. After each instruction description, a table showing the various available forms for an instruction is presented.

ADD

This instruction allows for integer addition. The following table shows the different variant of the add instruction using examples and noting the associated form.

Form	Example
4	`ADD #0xFF $A`
11	`ADD $B $A`

AND

The **`AND`** instruction is used for binary operation. It is a binary comparison and follows normal **`AND`** truth table:

```
1 and 1 = 1

1 and 0 = 0

0 and 1 = 0

0 and 0 = 0
```

Obviously this verification is extended to the full length of the compared elements (32 bits).

Form	Example
4	`AND #0xFF $A`
11	`AND $B $A`

CALL

You must fulfill the instruction prerequisite to use the **`CALL`** instruction. Doing otherwise will result in bad things, usually a memory error. Before using this instruction, you have to make sure that the stack pointer S register points to a valid and available region of memory. The call instruction will cause the address of the instruction following the **`CALL`** instruction to be pushed on top of the stack in order for this one to be used as return address.

Form	Example
3	`CALL #0x40000010`
2	`CALL $A`

CMP

The compare instruction is the only one that allows for **`FLAGS`** register modification. The **`FLAGS`** register is, at current time, only 3 bits wide. The top bit (**`0b100`**) will be set when the source element of the comparison is equal to the destination element. The middle bit (**`0b010`**) is set when source is lower that destination and the rightmost bit (**`0b001`**) is set when source is higher than destination.

Form	Example
4	`CMP #0xFF $A`
11	`CMP $A $B`

DIV

The div instruction allows for division. The source element will be divided by the destination element. The result of the division is placed in register **`A`** and the rest of the division is placed in register **`B`**

Form	Example
11	`DIV $C $B`

JMP

The jump instruction allows to skip parts of the code either unconditionally or conditionally by looking at the **`FLAGS`** register value. This instruction is special in the sense that it use a flag indicator to allow the user to select the conditions where the jump should be taken. Next is a listing of the possible conditions

`<>` Jump is always taken.

`<E>` or **`<Z>`** Jump is taken if 0b100 is set (E Flag).

`<L>` Jump is taken if 0b010 is set (L Flag).

`<H>` Jump is taken if 0b001 is set (H Flag).

These flags indicators can be combined to form complex conditions. For example, the **`<LE>`** indicator would allow the jump to happen if the E flag is set or if the L flag is set. Note that the immediate or register used by this instruction must hold a valid memory address when code can be fetched. Otherwise, bad things happen.

Form	Example
9	`JMP <E> #0x40000010`
10	`JMP <LH> $B`

JMPR

The **`JMPR`** instruction is identical to the **`JMP`** instruction with a major difference. The immediate or register used by the **`JMPR`** instruction must hold a relative offset to the instruction pointer I where the jump should result. For example, using the immediate value **`#0xFF`** will cause the execution unit to start fetching instruction **`0xFF`** bytes after the **`JMPR`** instruction.

Form	Example
9	`JMPR <E> #0x10`
10	`JMPR <LH> $B`

MEMR

The **`MEMR`** instruction allows for memory read operation. **`MEMR`** uses a width indicator that allows the user to read 1 to 4 bytes from memory into a register. The source can either be a register or an immediate value but must be a valid memory address. The width indicator is defined by using the **`[`** and **`]`** characters.

Form	Example
5	`MEMR [4] #0x40000000 $B`
6	`MEMR [4] $A $B`

MEMW

The **`MEMW`** instruction allows for memory write operation. Like the **`MEMR`** instruction, it uses a width indicator. Because of its nature, the **`MEMW`** instruction is available in more form that the **`MEMR`** instruction. The destination can be a register or an immediate but must be a valid memory address.

Form	Example
8	`MEMW [4] #0xFF #0x40000000`
5	`MEMW [4] #0xFF $A`

7	`MEMW [4] $B #0x40000000`
6	`MEMW [4] $B $A`

MOV

The **`MOV`** instruction allows for data displacement between register or for loading a register with an immediate value. This instruction could be use, by example, to set the stack pointer **`S`** to a valid memory region in order for the stack to be usable.

Form	Example
4	`MOV #0x40000200 $S`
11	`MOV $A $B`

MUL

The **`MUL`** instruction allow for integer multiplication between two registers. It is not possible to use the **`MUL`** instruction with an immediate value. Since multiplication can result in numbers that are bigger than 32 bits, the **`B`** register hold the higher 32 bits and the **`A`** register holds the lower 32 bits of the resulting number. The number should read as **`B:A`** after the multiplication.

Form	Example
11	`MUL $A $B`

NOP

The **NOP** instruction is the no operation instruction. It does nothing. Since Capua does not need to be 4 bytes aligned, the **NOP** instruction is typically useless. However, it can be use to fil the memory at boot time in order to ease development.

Form	Example
1	`NOP`

NOT

The **NOT** instruction will inverse the bits of a register. For example, if register **A** is equal to **0x01** before the **NOT** instruction, it will be equal to **0xFFFFFFFE** after the **NOT** instruction. The not instruction can, obviously, only be used with a register.

Form	Example
2	`NOT $A`

OR

The **OR** instruction is a bitwise or operation. It works following the Boolean logic or rules. Here is its truth table

1 or 1 = 1

0 or 1 = 1

1 or 0 = 1

0 or 0 = 0

It will impact all the bits in a register.

Form	Example
4	`OR #0xFF $A`
11	`OR $A $B`

POP

The **`POP`** instruction removes 32 bits from the stack and decreases the stack pointer 4 bytes back. The data that was on the top of the stack will be available in the register specified by the **`POP`** instruction. For the **`POP`** instruction to be safely used, the stack pointer **`S`** must be set to a valid memory address.

Form	Example
2	`POP $A`

PUSH

The **`PUSH`** instruction will add a 32 bits value on the top of the stack and increase the stack pointer **`S`** 4 bytes forward. In order for the **`PUSH`** instruction to be safely used, the stack pointer **`S`** must be set to a valid memory address before using the **`PUSH`** instruction.

Form	Example
3	`PUSH #0xFF`
2	`PUSH $A`

RET

The return instruction is typically used to come back from a call. It will take the element on the top of the stack and set the instruction pointer I value to it. In order to use the **`RET`** instruction, the caller has to make sure that the top of the stack value is a pointer to a region of memory where code can be fetched by the execution unit. Otherwise, bad things will happen.

Form	Example
1	`RET`

SHL

The **`SHL`** instruction mnemonic stand for SHift Left. It allows its user to shift the value of a register **`X`** bits to the left where **`X`** is a number from 1 to 8. The exceeding values are simply lost. As an example, consider the value **`0x80000001`**. Applying a one-bit shift left to this value will result in that value being transformed to **`0x00000002`**

Form	Example
4	`SHL #0x01 $A`
11	`SHL $B $A`

SHR

The **`SHR`** instruction is the same as **`SHL`** instruction except that the shift happens to the right. There for a one-bit right shift applied to the value **`0x80000001`** will result in the value **`0x40000000`**.

Form	Example
4	`SHR #0x01 $A`
11	`SHR $B $A`

SUB

The **`SUB`** instruction allows for subtracting values. It is important to understand that the operation work like this:

`destination = destination - source`

Not the other way around. Beware. It is also important to know that the **`SUB`** instruction only considers 32 bits signed, in the two's complement format, integer numbers.

Form	Example
4	`SUB #0x01 $A`
11	`SUB $B $A`

XOR

The **`XOR`** instruction is a bitwise operation, just like the **`AND`** and **`OR`** instruction. It follows the normal **`XOR`** logic and has the following truth table:

`1 xor 1 = 0`

`1 xor 0 = 1`

`0 xor 1 = 1`

```
0 xor 0 =0
```

It can be used to set a register value to 0.

Form	Example
4	`XOR #0x01 $A`
11	`XOR $B $A`

Section 2 - Tool Chain

Capua, being an architecture, needs its own assembler. For this project, 3 tools were developed in order to help software development and testing for the platform.

— Assembler

— Linker

— Debugger

The following sections explain each tool. Please keep in mind that each of these tools has been developed, at first, to allow for Capua to be tested. The second goal of these tools was to allow for software development. Therefore, these tools do present some bugs and, as a developer, you have to be careful to use these tools exactly as this text explains their use. Know that these bugs are being addressed, but reality is, this project is so big and time is so short that not everything can be quickly fixed. At the time of this writing, some features have still not been tested. This manual also serves the purpose of a development guideline.

Development Process Overview

The toolchain provided with Capua allows for complete development process from assembling the code to running it using a debugger. No compiler is provided. The linker allows for multiple files to be linked together. One important thing to understand is that this toolchain aims at providing flat binaries that can run freestanding on Capua environment. Please note that it is possible to use the toolchain to build more complex non freestanding binaries. However, building these will require a hosting environment and the resulting binary file will still result in a fully statically linked binary file. Also, the developer will need to provide some imagination to get these to work… It is, however, possible.

No Bare Metal Experience

Programmers coming to Capua development without any system or bare metal experience, need to understand that developing in a stripped down environment is not free of pitfalls. Nothing is provided for you. You are responsible for your own memory management and in memory binary management. Capua does not provide “segment” in any form. If you mess up, your memory placement and management you will destroy your own code while you are running it. On the bright side, you can also do crazy things like using multiple stacks at the same time and switching between them as you wish. Not that you should ;)

Assembler

Overview

Capua has its own assembler. It's easy to use and the actual implementation code is fairly straightforward. Before you start using the assembler there are a couple of things that you need to be aware of. First, please always remember that the current version of the assembler has been written as a testing tool for Capua. The assembler is good enough to write actual code and assemble it with the assembler but please be extra careful to fully respect the syntax described here since error message might not always be easy to understand. Also, in it's current state, the assembler is very "typo" sensitive. Every part of an instruction need to be separated by a white space. This include end of line comment:

```
MOV $A $B;This is not good enough
MOV $A $B ;This is fine
```

Obviously many of these problems are actually easy fixes. Feel free to jump in and contribute some code. This is, after all, an open source project. For now, because of limited time available, choice has been made to keep these small (but yet annoying) bugs in there since following guidelines will keep a programmer out of trouble. In case of trouble, the best way to fix your problem is to dig into the code and look at it. The code is not all that complicated. The tool chain code base is, however, way messier than the actual VM code. These tools have been developed with the initial intend of replacing them as soon as possible with some more stable tools. When running the assembler, the input file is transformed into a *.o* file. Please note that, the extension has been chosen because of the historic signification of a *.o* file. The file format used by the Capua tool chain has been made (designed would be an overstatement here ;)) with the intent of being able to link multiple files together in order to form a flat binary file. The current format is also very simple to understand and is a mix of XML and binary data.

Capua .o file format

The Capua file format is very simple. There is, however a bug associated with the file format. The bug is obvious if you try to look for it. This could have easily been avoided at development time but since the creation of the tool chain was rushed the bug knowingly made if to this version of the assembler. The file format will be modified in future time to avoid this problem. The bug can be avoided by a careful programmer. Now, because I find it funny, I will not tell what that bug is and let you guys figure this out ;)

The format itself is very simple. Here is the general form:

```
<AssemblySize></AssemblySize>
<ExternalSymbols>
    <refName></refName>
    <refAdd></refAdd>
</ExternalSymbols>
<InternalSymbols>
    <refName>END</refName>
    <refAdd></refAdd>
    ...
</InternalSymbols>
<Text>...</Text>
```

The following table explains every part of the file format in details.

Tag	Explanation
AssemblySize	This tag holds the projected size of the final linked version of this file. The size if kept big endian binary encoded inside the tag. The size calculation was put in the assembler since it was easier and faster to put it there. Also, the info is readily available for the linker when this one needs to calculate file wide address across multiple files that needs to be linked together.
ExternalSymbols	This tag holds a list of tags (refName/refAdd couples). The symbols present in the ExternalSymbols list are seen as “global” and, therefore, are available to the linker when linking multiple object together.
InternalSymbols	This tag is the same as ExternalSymbols except that the symbols listed in this tag are not available to the linker when linking multiple files together. This was originally made that way to prevent name collision on the global (external) scale at linking time. The linking process later add the source file name to the symbols so that all

	symbols are available (internal and external) when using the debugger while still avoiding collision.
refName	A refName tag must be inside of an ExternalSymbols or an InternalSymbols tag. It also has to be followed by a refAdd tag. The refName simply holds a texte version of the symbol name. The symbol name is determined by memory reference/naming in the assembly code written by the programmer.
refAdd	This that follows a refName tag and simply indicate the offset where the symbol can be found from the 0 address (relative to the beginning of the current file). Note that the offset is relative to a fully linked file. Not to an object file. That address eventualy replace the symbol name when the file is linked
Text	The text tag holds the assembled binary of the object file. A close look will reveal the presence of symbols name inside the text tag. These are replaced at linking time. The symbols are present in the text section like: ":SymbolName:". This is NOT a perfect (not even a good one) solution but it was fast. (Always keep in mind these tools were originally written to test the execution unit, not to write usable code with them)

This is it! The file format is that simple.

Writing assembly code

All the instructions have already been covered in the previous section. Therefore, this part will concentrate not on the code but on assembler directives. One important thing to note and that has not been explicitly stated up to now, it that, unlike other architectures, the assembly language and assembler do not have the ability to do indirect memory access. You can however get similar behaviour by doing offset calculation, manually, in code (have fun, and, yes it's a pain). The following table list available directives and show how to use these.

Directive	Description
symbolName:	An arbitrary name, by itself, on a line followed by a ":" character indicates a symbol. The assembler will add this in the symbol list and code will be able to reference symbolName anywhere an immediate could be used. Note that to use the symbol name in code, you have to precede the name by a ":". No ":" would be required after the symbol name when using the name in code.For example: **`MOV :stackAddress $A`** Would result in the linked address of :stackAddress being put as an immediate value in the mov instruction displayed. That could also be used in loops: **`JMP <> :loopStart`**
.global symbolName:	Will allow the assembler to add a symbol to the external symbols list. Note that even if the symbol name is followed by ":", you still need to "declare" this symbol on a line of it's own.
;	The ";" character, like in many other assembly language, indicate a comment. These can either be on a line of their own or at the end of a line, after an instruction. Please note that, in case a comment is put after an instruction, a space must separate the end of the instruction from the beginning of the comment (the ";" character).
.dataAlpha	This can be used anywhere as long as it sits on a line of it's own. This directive is followed by a white space and by free

	formed text. The text does not need to be quoted. In fact, it MUST NOT be quoted. The string ends at the end of the line. The assembler will add a 0x00 termination character at the end of the string at assembling time. Usage example
	`testString:`
	`.dataAlpha This is a test string`
	
	Please note that no comment can follow this line.

.dataNumeric	This is the same as .dataAlpha except that it allows the programmer to use 32 bits numeric values. Usage example:
	`testInt:`
	`.dataNumeric 0xFFFFFFFF`

$	Is the register prefix. Every register, when used in an assembly instruction, must be preceded by the "$" character.

#	Is the immediate prefix. Every immediate value (except when using a symbol) must be preceded by the "#" character.
	Multiple variants are possible for immediate values:
	`#0xFF`
	`#255`
	`#-1`
	`#0b11111111`

Short program example

The following table simply shows a working program that will calculate the length of a string.

```
; This will calculate the length of testString
.global start:

start:
JMP <> :stackSetup ;Jump over the string

testString:
.dataAlpha This is a test string

stackSetup:
    MOV :stack $S ;Stack is now usable

codeStart:

    PUSH :testString
    CALL :strlen
    SUB #0x4 $S ;stack reset

end:

;Following is the length calculation

;strlen(stringPointer)
strlen:
    MOV $S $A
    SUB #0x4 $A ;Calculate parameter offset
    MEMR [4] $A $A ;Get parameter in register A
    MOV $A $C ;Keep pointer to string start
lenover:
    MEMR [1] $A $B
    CMP #0x00 $B ;are we at the end of the string?
    JMP <E> :gotlen
    ADD #0x1 $A
    JMP <> :lenover ;not at the end, jump back
gotlen:
    SUB $C $A ;A will hold the len of the string at this
point.
    RET ;return value in register A

; This is at the end of the program.
; No risk of overwriting the program
stack:
```

Help

You can get help about the assembler usage by executing the assembler with the “-h” option.

Linker

Overview

Capua also has its own linker. This linker is intended to be used to assist in the creation of flat binary file. The linker can link multiple files together[1]. Since the binary files produced by the linker are meant to be used "bare metal", no dynamic linking is available. All addresses, after linking, will be hardcoded into the resulting binary file. This means that the linker needs to know, at linking time, the memory address where the binary is to be loaded in memory. Not providing the load address will result in the binary being linked to be loaded at the MEMORY_START_AT address (default is 0x40000000). This is fine for testing purpose since the execution unit starts fetching instructions at that address.

Beware

When linking multiple files together, the order in which you tell the linker to input the files IS OF MAJOR IMPORTANCE. The files will be put into the final binary in the same order as you input them into the linker.

Note about symbols

The linker will output a "*.sym*" file in the same folder as the final binary. That file is simply a symbol and address listing. This file, if available, will be loaded when you run the binary in the debugger and will allow you to use symbol names instead of memory address when executing the

[1] At time of writing, this feature is in the code but has yet to be tested

binary in the context of the debugger. Note that all the symbols present in that file have been modified with the name of their origin file as prefix. The symbol name themselves are also transformed to upper case.

Usage information

The linker is easy to use. For usage information, please execute the linker with the “-h” option.

Debugger

The debugger is really simple. It has almost all the basics features that most debuggers offer. At current time, three big limitations exist. The first is that you can’t modify a register value or an in memory value through the debugger. You can also not “step over” a function call easily. To do so, you would have to but a break point at the return of the function call. The last one is that you can’t simply “reload” the program without relaunching the debugger. All of these features are noted and will be added at some point in the future. Using the debugger is simple and very strait forward. Simply run the debugger with the “-h” option to learn how to launch it. Once launched, the debugger will break right at the beginning of your binary file. You can access the debugger help menu by typing “h” or “help” at the debugger prompt.

Section 3 - Memory Mapped Hardware

The virtual machine offers support for memory mapped hardware. Memory mapped hardware is accessible at specific memory addresses. Different address has different meaning. The way each device behave is specific to a single device. At the time of this writing only the memory management unit is accessible in memory.

Memory Management Unit

The memory management unit is not as complex as what it generally is in real computers. Capua's MMU serves a single purpose; to manage memory access permission. These permissions can be set on a per byte basis. No segments are available in this architecture.

Using the MMU

The MMU is mapped at address **`0x20000000`**. Modifying bytes at that address will cause the MMU to take action based on the content of the bytes modified. The following table explains the layout of the memory for the MMU.

Address	Length (byte)	Purpose
`0x20000000`	4	This holds the memory address where a

		permission modification is to take place. If this is set at value `0x40000000`, then that address is where the MMU will start modifying permissions.
`0x20000004`	1	Length of modification. If this is set, for example, at `0xF`, the MMU, when modifying the permission will modify the permission for the `0xF` bytes following the address that has been written in `0x20000000`.
`0x20000005`	5	The most significant bit of the byte located at that address, if set, will cause, when a write is done to the MMU, the permission modification action to be triggered. The three next bit, are used to set the permission values (read, write or execute). Here is the exact definition for the format of this byte: `0b1000 0000` – Trigger MMU action `0b0100 0000` – If set, add read permission `0b0010 0000` – If set, add write permission `0b0001 0000` – If set, add execute permission

Please note that the MMU will not, reset its own memory, after an action has been triggered. The programmer is responsible for keeping track of the MMU state.

The Clock

Capua also has a clock. The clock could be used, for example, as a source of entropy. The clock is mapped at address **`0x20000100`**. Reading 4 bytes at that address will allow the user to get the time, in seconds, since epoch. Writing at addresses belonging to the clock is not allowed. It will cause a memory error.

Spartacus Thread Multiplexer

When in game mode, an additional device is available at address **`0x20000200`**. That device is the Spartacus Thread Multiplexer. This device allows a player (only available in game mode) to start a new thread even if there is no operating system in place to support such an action. The STM is a special device that has special access to the game environment (residing outside of the VM). Its functioning is very simple. A player who wants to start a new thread simply write the address where the thread should start fetching code to the STM address (**`0x20000200`**). Right after a write to that address, the game environment will generate a new, and empty, context (**`A=0, B=0, C=0, S=0, I=x, FLAGS=0`**, where x is equal to the address written at STM address). That context will then be scheduled by the game environment (interleaving a specific player threads one with another). A user can have a maximum of 3 thread running on the system at the same time. No instructions are available to kill a thread. Killing a thread can be achieved by forcing a system exception. Writing to the STM address when not in game mode will not case an exception. The write will be accepted into STM memory buffer but no further action will be taken. After spawning a new thread, the data that was written to STM address stays there until a new write is done to the STM address. STM address content can also be read.

Section 4 – Step by Step example

This section simply provides step by step instructions in using the tool chain to generate a runnable binary file. The file will then be running in the debugger.

Step 1 – Writing the program

No support here. Simply write a program. If you want to follow along, you can use the program that is provided in the following table. Please note that this example does not use the **S** register as a stack pointer but rather as a general purpose register.

```
; This will calculate Fibonaccie number
.global start:

codeStart:
  MOV #0 $A
  MOV #1 $B

fiboStart:
  MOV $A $C
  ADD $B $C
  MOV $B $A
  MOV $C $B
  MOV $A $S ;Current fibo number will be in $S after this
  JMP <> :fiboStart
  MOV #1 $A
  MOV #2 $B
end:
```

Simply save that program to a file.

Step 2 – Assemble the file

Here, you simply use the **`Assembler.py`** script available at Spartacus root in order to generate a **`.o`** file. The following table shows how to use the **`Assembler.py`** script.

```
C:\Spartacus>python Assembler.py -i fibonaccie.txt
Assembler about to begin, following options will be used
  input file:            fibonaccie.txt
  output file:           fibonaccie.o
Assembler done, output file has been written to fibonaccie.o

C:\Spartacus>
```

Step 3 – Link the file

Just like for the assembly step, here, you simply call the **`Linker.py`** script in order to generate the final binary file. Please note that the linker will also generate a **`.sym`** file. You can get rid of this file if you do not want to use it. The following table show the linking of the previously generated **`.o`** file.

```
C:\Spartacus>python Linker.py -i fibonaccie.o -o fibonaccie.bin
Linker about to begin, following options will be used
  input file:            ['fibonaccie.o']
  symbols file:          fibonaccie.sym
  output file:           fibonaccie.bin
Linker done, output file has been written to fibonaccie.bin

C:\Spartacus>
```

Step 4 – Run the binary in debugger

You are now ready to run the binary file using the debugger. Doing so is, again easy, you simply use the **`Debugger.py`** script, also available at the root of the Spartacus project. The following table shows the **`fibonaccie.bin`** file loaded in the debugger, ready to be executed.

```
C:\Spartacus>python Debugger.py -i fibonaccie.bin
Debug session about to begin, following options will be used
  input file:             fibonaccie.bin
  symbols file:           fibonaccie.sym
Building Capua execution environment
Loading ('fibonaccie.bin',) in memory
Done loading ('fibonaccie.bin',) into memory
Loading symbols from file fibonaccie.sym
Done loading symbols
Debugging session is ready to be used. Have fun!

Next instruction to be executed:
0x40000000 : MOV #0x0 $A
('fibonaccie.bin',):
```

Step 5 – Enjoy!

This is it! For more help, please contact us or have a look at the code!