VCPU-32 System Architecture and Instruction Set Reference

Helmut Fieres Version B.00.06 July, 2024

Table of Content

Introduction

Designers of the seventies and early eighties CPUs almost all used a micro-coded approach with hundreds of instructions. RISC was just starting to enter the stage. Registers were not really generic but often had a special function or meaning and many instructions were rather complicated and the designers felt they were useful. The compiler writers often used a small subset ignoring these complex instructions because they were so specialized. The later eighties and nineties shifted to RISC based designs with large sets of registers, fixed word instruction length, a large virtual memory address range and instructions that are in general much more pipeline friendly. What if some of these principles had found their way earlier into CPU designs?

Welcome to VCPU-32. VCPU-32 is a simple 32-bit CPU with a register-memory model and segmented virtual memory. The design is heavily influenced by Hewlett Packard's PA_RISC architecture, which was initially a 32 bit RISC-style register-register load-store machine. Almost all of the key architecture features come from there. However, other processors, such as the Data General MV8000, the DEC Alpha, and the IBM PowerPc, were also influential or at least investigated for their architectural choices. The original design goal that started this work was to truly understand the design process and implementation tradeoffs for a simple pipelined CPU. While it is not a design goal to build a modern, competitive CPU, the CPU should nevertheless be useful and largely follow established common design practices. For example, instructions should not be invented because they seem to be useful, but rather designed with the assembler and compilers in mind. Instruction set design is also CPU design. Register memory architectures were typically micro-coded complex instruction machines. In contrast, VCPU-32 instructions will be hard coded and are in general "pipeline friendly", avoiding data and control hazards and stalls where possible.

The instruction set design guidelines center around the following principles. First, in the interest of a simple and efficient instruction decoding step, instructions are of fixed length. A machine word is the instruction word length. As a consequence, some address offsets are rather short and addressing modes are required for reaching the entire address range. Instead of a multitude of addressing modes, typically found in the CISC style CPUs, VCPU-32 offers very few addressing modes with a simple base address - offset calculation model. No indirection or any addressing mode that would require to read a memory item for address calculation is part of the architecture.

There will be few instructions in total, however, depending on the instruction several options for the instruction execution are encoded to make an instruction more versatile. For example, a boolean instruction will offer options to negate the result thus offering and "AND" and a "NAND" with one instruction. Wherever possible, useful options such as the one outlined before are added to an instruction, such that it covers a range of instructions typically found on the CPUs looked into. Great emphasis is placed in that such options do not overly complicated the pipeline and only increase the overall data path length slightly.

Modern RISC CPUs are typically load/store CPUs and feature a large number of general registers. Operations takes place between registers. VCPU-32 follows a slightly different route. There is a smaller number of general registers and in addition to computation between registers, a register-memory model is also supported. There are however no instructions that read and write to memory in one instruction cycle as this would complicate the design considerably.

VCPU-32 offers a large address range, organized in segments. Segment Id and offset form a virtual address. In addition, a short form of a virtual address, called a logical address, will select segment register based on the upper two bits of the logical address to form a virtual address. Segments are also associated with access rights and protection identifies. The CPU itself can run in user and privilege mode.

This document describes the overall architecture, instruction set and runtime model for VCPU-32. It is organized into several parts. The first part will give an overview on the architecture. It presents the memory model, registers sets and basic operation modes. The major part of the document then presents the instruction set. Memory reference instructions, immediate instructions, branch instructions, computational instructions and system control instructions are described in detail. These chapters are the authoritative reference of the instruction set.

The runtime environment chapters present the runtime architecture. Although the CPU architecture and instructions allow for a great flexibility how to write programs, a runtime architecture is essential to define the common usage of the instruction set for assembler and compilers.

Architecture Overview

This chapter presents an overview on the overall VCPU-32 architecture. The chapter introduces the memory model, the register set, address translation and trap handling.

A Register Memory Architecture.

VCPU-32 implements a **register memory architecture** offering few addressing modes for the "operand" combined with a fixed instruction word length. For most of the instructions one operand is the register, the other is a flexible operand which could be an immediate, a register content or a memory address from where the data is obtained or placed. The result is placed in the register which is also the first operand. These type of machines are also called two address machines.

```
REG <– REG op OPERAND
```

In contrast to similar historical register-memory designs, there is no operation which does a read and write operation to memory in the same instruction. For example, there is no "increment memory" instruction, since this would require two memory operations and is in general not pipeline friendly. Memory data access is performed on a machine word, half-word or byte basis. Besides the implicit operand fetch in an instruction, there are dedicated memory load / store instructions. In addition to the register-immediate operand and register memory-operand mode, one operand mode supports the three operand model (Rs = Ra OP Rb), specifying two registers and a result register, which allows to perform three address register for computational operations as well. The machine can therefore operate in a memory register model as well as a load / store register model.

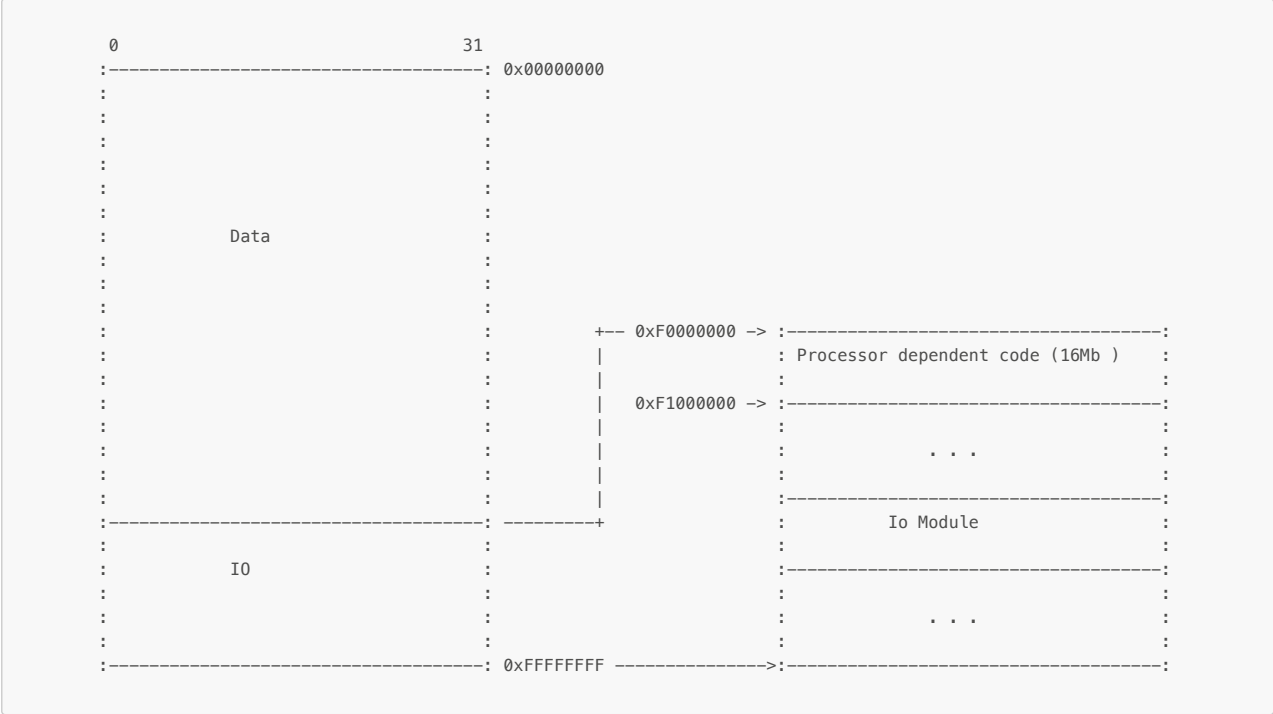
Data Types

VCPU-32 is a big endian 32-bit machine. The fundamental data type is a 32-bit machine word with the most significant bit being left. Bits are numbered from left to right, starting with bit 0 as the MSB bit. Memory access is performed on a word, half-word or byte basis. All addresses are however always expressed as bytes addresses.

```
 MSB                                                 LSB
 0             8             16            24          31
:-------------------------------------------------------:
:  word                                                 :
:-----------------------------:-----------------------:
:  halfword 0                 :  halfword 1            :
:--------------:--------------:--------------:--------:
:  byte 0      :  byte 1      :  byte 2      :  byte 3 :
:--------------:--------------:--------------:--------:
```

Memory and IO Address Model

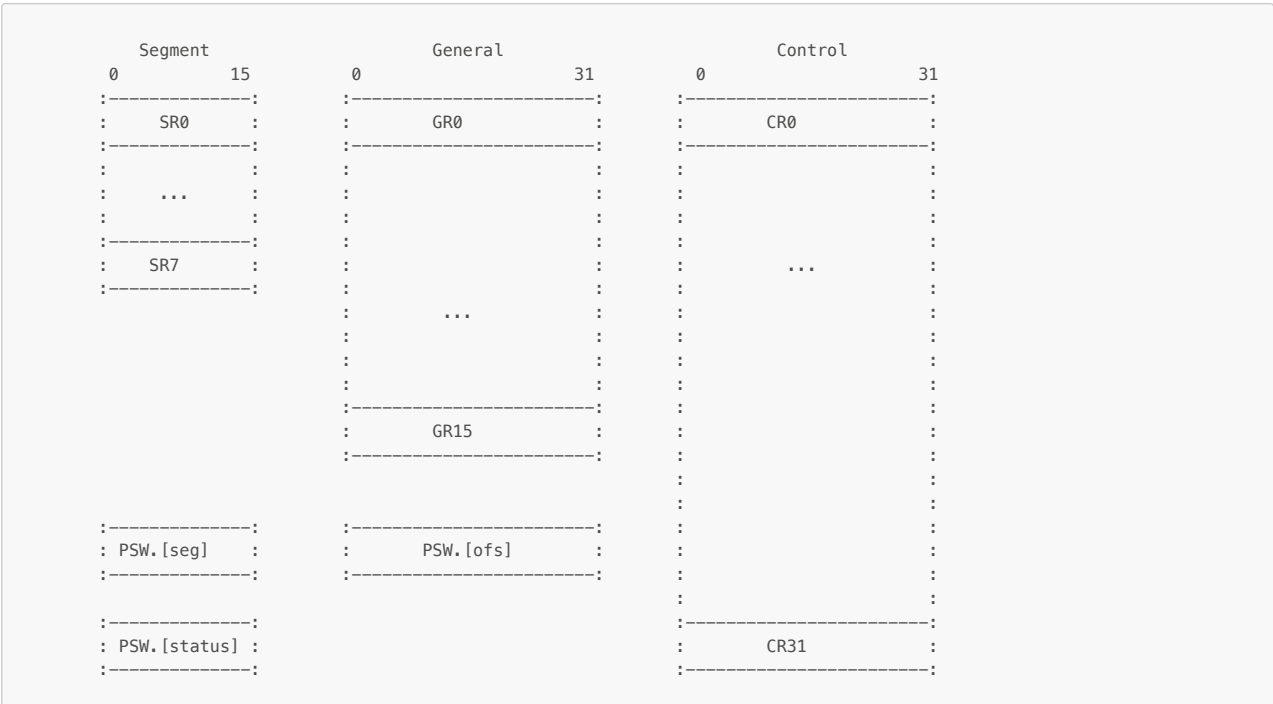
VCPU-32 features a physical memory address range of 32-bits. The picture below depicts the physical memory address layout. The physical address range is divided into a memory data portion and an I/O portion. The entire address range is directly accessible with the absolute load and store instructions.



The I/O address space dedicates a 1/16th of the space to an area that contains the processor dependent code. Typically, this is a set of library functions to manage the processor itself but also provide low-level primitives for the boot process and the operating system. The remainder of the I/O address space is divided into I/O modules, which map the I/O hardware components to a memory addressable location. Both PDC and I/O firmware design is explained in a later part of the document.

General Register Model

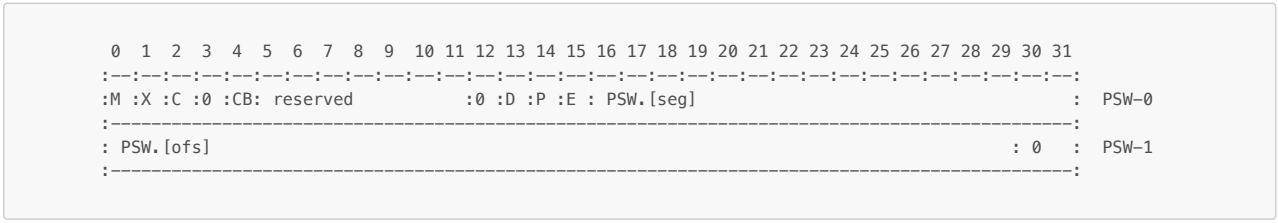
VCPU-32 features a set of registers. They are grouped in **general registers**, **segment registers** and **control registers**. There are sixteen general registers, labeled GR0 to GR15, and eight segment registers, labeled SR0 to SR7. All general registers can do arithmetic and logical operations. The eight segment registers hold the segment part of the virtual address. The control registers, labelled CR0 to CR31, contain system level information such as protection registers and interrupt and trap data registers.



Some general registers have a dedicated use. Register zero will always return a zero value when read and a write operation will have no effect. Although there are only 16 general registers in the CPU, implementing such a register greatly simplifies the instruction design, in that it for example provides a target when the result is not needed. General register one is a scratch register and also serves as an implicit target for some instructions.Segment register zero serves as implicit target for some of the external branching instructions. Other general registers and segment registers may have dedicated purpose defined in the runtime architecture. Hardware only implements dedicated purposes for the above mentioned registers.

Processor State

VCPU-32 features two registers to hold the processor state. The **instruction address** part holds the segment Id and offset of the current instruction being executed. The instruction address is the virtual or absolute memory location of the current instruction. The lower two bits are zero, as instruction words are always word aligned in memory. The **status** part holds the processor status information, such as the carry bit or current execution privilege. The portion is labelled **ST**.



// ??? need a Z bit for single stepping debugging support....cleared after each instruction, when set causes aDebug trap.

// ??? perhaps need a "N" bit for nullification support.

// ??? how would we support data breakpoints ? Would they need a separate mechanism ?

// ??? how about taken branch traps (taken, lower and higher : T H L)

// ??? how about putting also a nullify bit for future enhancements for nullification ? (N)

// ??? we would also need a recovery counter enable bit (R) for the recovery counter

Bits 12 .. 15 of the processor status represent the bit that can be modified by the privileged MST instruction. Setting the other status bits requires the usage of the privileged RFI instruction.

Flag	Name	Purpose
M	Machine Check	When set, checks are disabled.
X	Execution level	When set, the CPU runs in user mode, otherwise in privileged mode.
C	Code Translation	When set, code translation is enabled.
CB	Carry/Borrow	The carry bit for ADD, ADC, SUB and SUBC instructions.
D	Data Translation	When set, data translation is enabled.
P	Protection Check	When set, protection checking is enabled.
E	External Interrupt Enable	When set, an external interrupt are enabled.

Control Registers

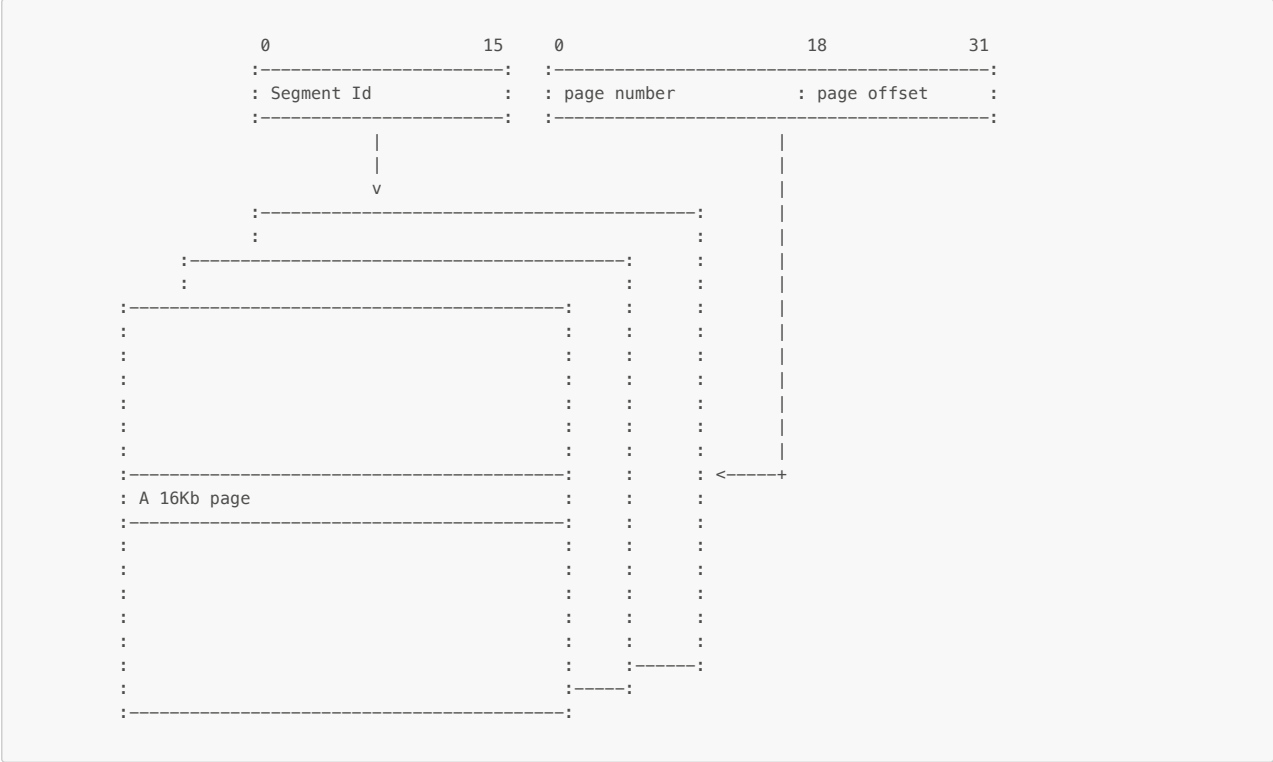
The **control registers** hold information about the processor configuration as well as data needed for the current execution. There are 32 control registers. Examples are the shift amount register, protection ID values, the current trap handling information, or the interrupt mask.

CR	Name	Purpose
0	**SWR**	System Switch Register. On an optional front panel, a set of switches and LEDs represent the switch register.

CR	Name	Purpose
1	**RCTR**	Recovery Counter. Can be used to implement a watchdog function. (tbd)
2	**SHAMT**	Shift Amount register. This is a 5-bit register which holds the value for variable shift, extract and deposit instructions. Used in instructions that allow to use the value coming from this register instead of the instruction encoded value.
3		reserved
4 - 7	**PID-n**	Protection Id registers. When protection ID checking is enabled, the segment part of a virtual page accessed is checked for matching one of the protection IDs stored in these control registers. There are two protection IDs in each register.
8 -15		reserved
16	**I-BASE-ADR**	Interrupt and trap vector table base address. The absolute memory address of the interrupt vector table. When an interrupt or trap is encountered, the next instruction to execute is calculated by adding the interrupt number shifted by 32 to this address. Each interrupt has eight instruction locations that can be used for this interrupt. The table must be page aligned.
17	**I-PSW-0**	When an interrupt or trap is encountered, this control register holds the current status word and segment portion.
18	**I-PSW-1**	When an interrupt or trap is encountered, control register holds the current instruction address offset.
19 - 21	**I-PARM-n**	Interrupts pass along further information through these control registers.
22	**I-EIM**	External interrupt mask.
23		reserved
24 - 31	**TMP-n**	These control registers are scratch pad registers. Temporary registers are typically used in an interrupt handlers as a scratch register space to save general registers so that they can be used in the interrupt routine handler. They also contain some further values for the actual interrupt. These register will neither be saved nor restored upon entry and exit from an interrupt.

Segmented Memory Model

The VCPU-32 memory model features a segmented memory model**. The address space consists of up to 2^16 segments, each of which holds up to 2^32 bytes in size. Segments are further subdivided into pages with a page size of 16 Kbytes. The concatenation of segment ID and offset form the **virtual address**.



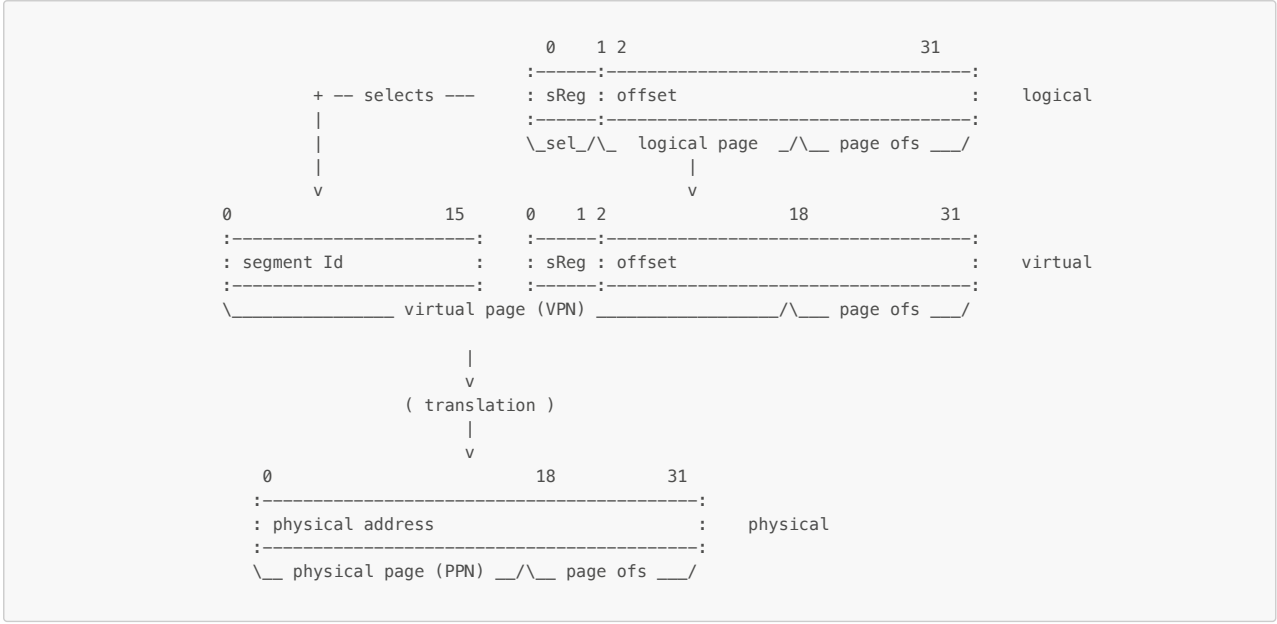
A segment Id of 16-bits allows for up to 65636 segments. Segments should be considered as address ranges from which an operating system allocates virtual memory space for code and data objects. A page size of 16Kb will ease the burden on address translation hardware.

Address Translation

VCPU-32 defines three types of addresses. At the highest level is the **logical address**. The logical address is a 32-bit word, which contains a 2-bit segment register selector and a 30-bit offset. During data access with a logical address the segment selector selects from the segment register set SR4 to SR7 and forms together with the remaining 30-bit offset a virtual address. Since the upper two bits are inherently fixed, the address range in the segment is one of four possible 30-bit quadrants. For example, when the upper two bits are zero, the first quadrant *0x00000000* to *0x3FFFFFFF* of the segment is addressable. When the upper two bits are *0x2*, the reachable address range is *0x80000000* to *0xBFFFFFF*. When the segment registers SR4 to SR7 all would point to the same segment, the entire address range is reachable via a logical address. It is however more likely that these registers point to different segments though.

Instructions that work with an address have a two bit segment select field. A value of zero indicates to use the implicit segment register selection as outlined before, a value of 1 to 3 will select segment register SR1 to SR3. Either way the logical address translation will result in a virtual address.

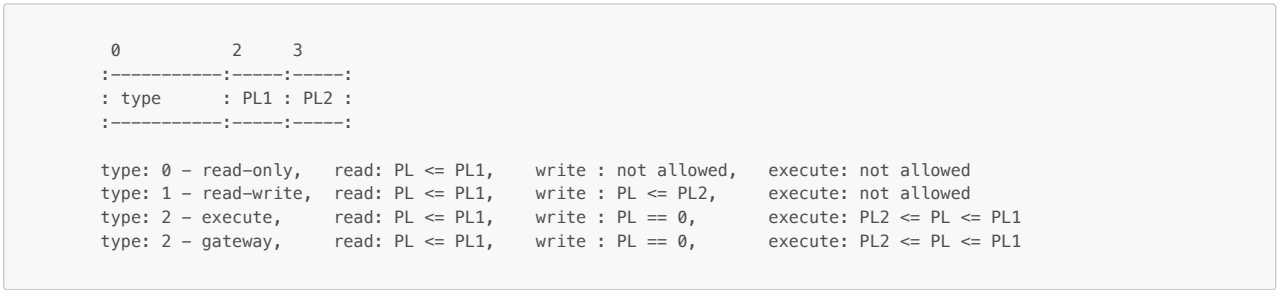
The **virtual address** is the concatenation of a segment and an offset. Together they form a maximum address range of 2^16 segments with 2^32 bytes each. Once the virtual address is formed, the virtual to physical translation process is the same for both logical and virtual addressing mode. The following figure shows the translation process. First a logical address is translated into a virtual address. A virtual address is then translated into a **physical address**.



Address translation is separately enabled for code and data translation. When translation is disabled, the address offset directly maps to the physical address range with the segment part ignore the segment zero has a special role. Hardware must guarantee that a virtual address with a zero segment, e.g. *0x0.0x500* will also directly map to the physical address specified by the offset. The maximum physical memory size that can be reached this way is 4 GBytes. The architecture does however not exclude supporting a larger physical address range. For example, a translation buffer hardware could allow for a larger physical address range beyond the 4GBytes. This part of physical memory is however only reachable when address translation is enabled.

Protection Model

VCPU-32 implements a protection model along two dimensions. The first dimension is the **access control** and privilege level check. Each page is associated with a **page type**. There are read-only, read-write, execute and gateway pages. Each memory access is checked to be compatible with the allowed type of access set in the page descriptor. There are two **privilege levels**, user and supervisor mode. Access type and privilege level form the access control information field, which is checked for each instruction and data memory access. For read access the privilege level us be least as high as the PL1 field, for write access the privilege level must be as least as high as PL2. An exception are the execute and gateway pages, where a write access is only allowed for privilege code. For execute access the privilege level must be at least as high as PL1 and not higher than PL2. If the instruction privilege level is not within this bounds, a privilege violation trap is raised. If the page type does not match the instruction access type an access violation trap is raised. In both cases the instruction is aborted and a memory protection trap is raised.



The second dimension of protection is a **protection ID**. VCPU-32 allows to record a set of segment Id values in the processor control registers. For each access to a segment that has the protection check bit set, one of the protection Id control registers must match the segment Id. If not, a protection violation trap is raised.

Protection Id checking is typically used to form a grouping of access. A good example are stack data segment, which is accessible at user privilege level. Since the CPU features a global virtual memory address space, every every task could access a data stack of another task. With protection Id checking enabled and the segment Id loaded in one of the control registers, only the corresponding user task is allowed to access the stack data segment with a matching protection Id.

Address translation and caching

The previous section depicted the virtual address translation process. While the CPU architects the logical, virtual and physical address scheme, it does not specify how exactly the hardware supports the translation process. A very common model is to have for performance reasons instruction and data translation buffers(TLB) for caching translation results and data caches for caching memory data. The CPU architecture does not specify a particular model of TLB and cache implementation. Split TLBS and unified TLBs, L1 and L2 caches and other schemes are models to consider. This section just outlines the common architecture and instructions to manage these CPU components.

Translation Lookaside Buffer

For performance reasons, the virtual address translation result is stored in a translation look-aside buffer. The TLB is essentially a copy of the page table entry information that represents the virtual page currently loaded at the physical address. Depending on the hardware implementation, there can be a combined TLB or a separate instruction TLB and data TLB. A TLB is essentially a cache for address translations. The virtual page number, VPN, is stored along with attributes and the physical page number, PPN, in the TLB hardware. The architecture allows for a physical address range that is larger than the 4Gb main memory described before. For example, a virtual page can map to a physical page beyond the 4Gb range that is however not directly addressable with absolute addressing modes. Physical pages beyond the 4Gb range can therefore only be reached with address translation enabled. The current architecture allows for a 34-bit physical address space, which is a 16GB memory space.

```
TLB fields (conceptual example):

       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
      :--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
      :V :T :D :B :P : 0      : AR        : VPN-H                                                      :
      :--------------------------------------------------------------------------------------------:
      : VPN-L                            : PPN                                                        :
      :--------------------------------------------------------------------------------------------:
```

Field	Purpose
V	the entry is valid.
T	page reference trap. If the bit is set, a reference to the page results in a trap.
D	dirty trap. If the bit is set, the first write access raises a trap.
B	data reference trap. If the bit is set, data access raises a trap.
P	segment Id checking enabled for the page.
AR	The access rights data for the page.
VPN	the virtual page number.
PPN	the physical page number.

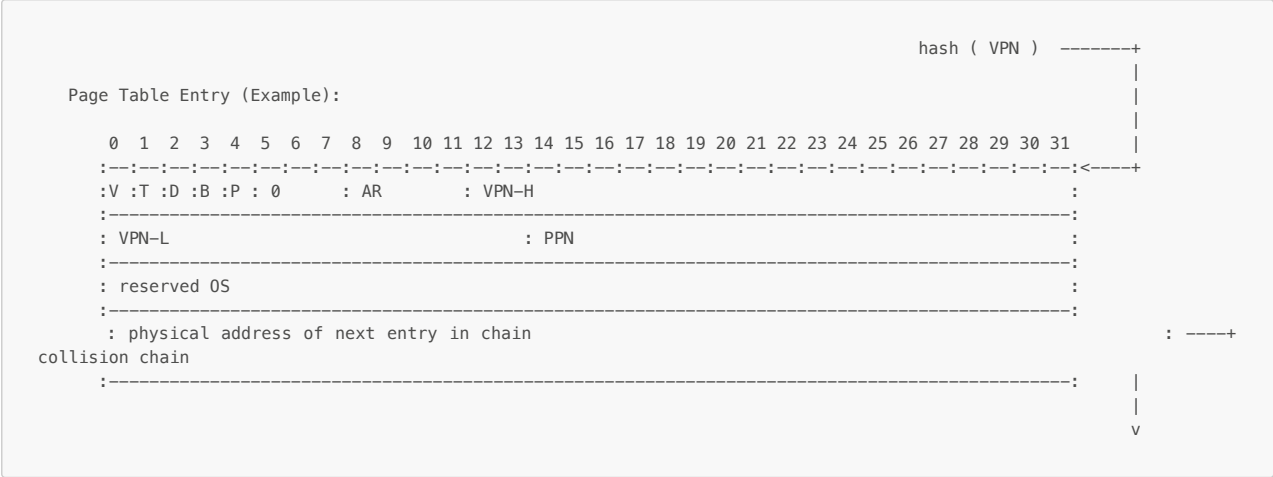
When address translation is disabled, the respective TLB is bypassed and the address represents a physical address. Also, protection ID checking is disabled. The D, T, B only apply to a data TLB. When the processor cannot find the address translation in the TLB, a TLB miss trap will invoke a software handler. The handler will walk the page table for an entry that matches the virtual address and update the TLB with the corresponding physical address and access right information, otherwise the virtual page is not in main memory and there will be a page fault to be handled by the operating system. In a sense the TLB is the cache for the address translations found in the page table. Again, the implementation of the TLB is hardware dependent.

Caches

VCPU-32 is a cache visible architecture, featuring a separate instruction and data cache. While hardware directly controls the cache for cache line insertion and replacement, the software has explicit control on flushing and purging cache line entries. There are instructions to flush and purge cache lines. To speed up the entire memory access process, the cache uses the virtual address to index into the arrays, which can be done in parallel to the TLB lookup. When the physical address tag in the cache and the physical page address in the TLB match, there will be a cache hit. The architecture will not specify the size of cache lines or cache organization. Typically a cache line will hold four to eight machine words, the cache organization is in the simplest case a single set direct mapped cache. The CPU architecture does not specify a particular cache and layer model. The CPU architecture just specifies instructions to handle cache operations, such as delete or flushing a cache.

Page Tables

The CPU architecture does not specify the page table model. Physical memory is organized in pages with a size of 16 Kbytes. There are several models of page tables possible. One model is to use an hashed page table model, in which each physical page allocated has an entry in the page table. During a TLB miss, a hash function computes an index into the hashed page table to the first page table entry and then walks the collision chain for a matching entry. A possible implementation of the page table entry is shown below.



Each page table entry contains the virtual page number, the physical pages number, the access rights information, a set of status flags and the physical address of the next entry in the page table. On a TLB miss, the page table entry contains all the information that needs to go into the TLB entry. For a fast lookup index computation, page table entries should have a power of 2 size, typically 4 or 8 words. The reserved fields could be used for the operating system memory management data.

Field	Purpose
V	the entry is valid.
T	page reference trap. If the bit is set, a reference to the page results in a trap.
D	dirty trap. If the bit is set, the first write access to the TLB raises a trap.
B	data reference trap. If the bit is set, access to the data page raises a trap.
P	segment Id checking enabled for the page.
AR	The access rights data for the page.
VPN	the virtual page number.
PPN	the physical page number.
next PTE	the physical address of the next page table entry, zero if there is none.

Locating a virtual page in the page table requires to first index into the page table and perhaps follow the chain of page table entries. The following code fragment depicts a possible hash function.

```
Hash function ( Example ):

const uint segShift        = 4;      // the number of bits to shift the segment part. This specifies how many
                                     // consecutive pages will result in consecutive hash values.
const uint pageOffsetBits  = 14;      // number of bits for page offset, page size of 16K
const uint hashTableMask   = 0x3FF;   // a hash table size, memory size dependent, must be a power of two.

uint hash_index ( uint16_t segment, uint32_t offset ) {

  return((( segment << segShift ) ^ ( offset >> pageOffsetBits )) & hashTableMask );
}
```

Control Flow

Control flow is implemented through a set of branch instructions. They can be classified into unconditional and conditional instruction branches.

Unconditional Branches. Unconditional branches fetch the next instruction address from the computed branch target. The address computation can be relative to the current instruction address (instruction relative branch) or relative to an address register (base register relative). For a segment local branch, the instruction address segment part will not change. Unconditional branches are also used to jump to a subroutine and return from there. The branch and link instruction types save the return point to a general register. External calls are quite similar, except that they always branch to an absolute offset in the external segment. Just like the branch instruction, the return point can be saved, but this time to a segment is additionally stored in SR0.

Conditional Branches. VCPU-32 features one conditional branch instructions. These instruction compares two register values for a certain condition. If the condition is met a branch to the target address is performed. The target address is always a local address and the offset is instruction address relative. Conditional branches adopt a static prediction model. Forward branches are assumed not taken, backward branches are assumed taken.

Privilege change

Modern processors support the distinction between several privilege levels. Instructions can only access data or execute instructions when the privilege level matches the privilege level required. Two or four levels are the most common implementation. Changing between levels is often done with a kind of trap operation to a higher privilege level software, for example the operating system kernel. However, traps are expensive, as they cause a CPU pipeline to be flush when going through a trap handler.

VCPU-32 follows the PA-RISC route of using gateways for privilege promotion. Im contrast, VCPU-32 implements a two privilege level model with **user** and **priv** mode. A special branch instruction will raise the privilege level when the instruction is executed on a gateway page. The branch target instruction continues execution with the privilege level specified by the gateway page and the "X" bit in the status register is set. Return from a higher privilege level to a code page with lower privilege level, will automatically set the lower privilege level.

Each instruction fetch compares the privilege level of the page where the instruction is fetched from with the status register "P" bit. A higher privilege level on an execution page results in a privilege violation trap. A lower level of the execute page and a higher level in the status register will in an automatic demotion of the privilege level.

Interrupts and Exceptions

Interrupts and exceptions are events that occur asynchronously to the instruction execution. Depending on the type, they occur during the execution of an instruction or are checked in between instructions. Example for the former is the arithmetic overflow tap, example for the latter is an external interrupt. Depending on the interrupt or exception type, the instruction is restarted or execution continues with the next instruction.

```
General flow:

- save PSW-0 and PSW-1 to the I-PSW-0 and I-PSW-1 control registers.
- save interrupted instruction to I-TEMP0.
- copy few GRs to shadow GRs( CRs ) to have room to work ( also SRs ? Tbd. )
- set I-PSW-0 to zero, this will turn address translation and protection checking off, putting the CPU in privileged
mode.
- set I-PSW-1 to the interrupt vector table I-BASE-ADR plus the trap index
- start execution of the trap handler
```

The control register I-BASE-ADR holds the absolute address of the interrupt vector table. Upon an interrupt, the current instruction address and the instruction is saved to the respective control registers and control branches then to the interrupt handler in the interrupt vector table. Each entry is just a set of instructions that can be executed.

TrapId	Name	IA	P0	P1	P2
0	**reserved**				
1	**Machine Check**	IA of the current instruction	check type	-	-
2	**Power Failure**	IA of the current instruction	-	-	-

TrapId	Name	IA	P0	P1	P2
3	**Recovery Counter Trap**	IA of the current instruction	-	-	-
4	**External Interrupt**	IA of the instruction to be executed after servicing the interrupt.	-	-	-
5	**Illegal instruction trap**	IA of the current instruction	instr	-	-
6	**Privileged operation trap**	IA of the current instruction	instr	-	-
7	**Privileged register trap**	IA of the current instruction	instr	-	-
8	**Overflow trap**	IA of the current instruction	instr	-	-
9	**Instruction TLB miss**	IA of the current instruction	-	-	-
10	**Non-access instruction TLB miss**	IA of the current instruction	-	-	-
11	**Instruction memory protection trap**	IA of the current instruction	-	-	-
12	**Data TLB miss**	IA of the current instruction	instr	data adr segId	data adr ofs
13	**Non-access data TLB miss**	IA of the current instruction	-	-	-
14	**Data memory access rights trap**	IA of the current instruction	instr	data adr segId	data adr ofs
15	**Data memory protection trap**	IA of the current instruction	instr	data adr segId	data adr ofs
16	**Page reference trap**	IA of the current instruction	instr	data adr segId	data adr ofs
17	**Alignment trap**	IA of the current instruction	instr	data data adr segId	data adr ofs
18	**Break instruction trap**	IA of the current instruction	instr	data data adr segId	data adr ofs
19 .. 31	reserved				

Instruction and Data Breakpoints

A fundamental requirement is the ability to set instruction and data breakpoints. An instruction breakpoint is encountering the BRK instruction in an instruction stream. The break instruction will cause an instruction break trap and pass control to the trap handler. Just like the other traps, the pipeline will be emptied by completing the instructions in flight and flushing all instructions after the break instruction. Since break instructions are normally not in the instruction stream, they need to be set dynamically in place of the instruction normally found at this instruction address. The key mechanism for a debugger is then to exchange the instruction where to set a break point, hit the breakpoint and replace again the break point instruction with the original instruction when the breakpoint is encountered. Upon continuation and a still valid breakpoint for that address, the break point needs to be set after the execution of the original instruction.

Data breakpoints are traps that are taken when a data reference to the page is done. They do not modify the instruction stream. Depending on the data access, the trap could happen before or after the instruction that accesses the data. A write operation is trapped before the data is written, a read instruction is trapped after the instruction took place.

External Interrupts

To be defined ...

Instruction Set Overview

This chapter gives a brief overview on the instruction set. The instruction set is a fixed word length instruction format of one machine word. The instruction set is divided into five general groups of instructions. The **memory reference** group contains the instructions to load and store to virtual and physical memory. The **immediate** group contains the instructions for building an immediate value up to 32 bit. The **branch** group present the conditional and unconditional instructions. The **computational** instructions perform arithmetic, boolean and bit functions such as bit extract and deposit. Finally, the **control** instruction group contains all instructions for managing HW elements such as caches and TLBs, register movements, as well as traps and interrupt handling.

Memory Reference Instructions

Memory reference instruction operate on memory and a general register with a unit of transfer being a word, a half-word or a byte. In that sense the architecture is a typical load/store architecture. However, in contrast to a load/store architecture VCPU-32 load / store instructions are not the only instructions that access memory. Being a register/memory architecture, all instructions with an operand field encoding may access memory as well. There are however no instructions that will access data memory access for reading and writing back an operand in the same instruction. Due to the operand instruction format and the requirement to offer a fixed length instruction word, the offset for an address operation is limited but still covers a large address range.

The **LDw** and **STw** and instructions access virtual memory. The segment selector field allows to use a logical address with implicit selection of SR4..SR7, or an explicit virtual address using SR1..SR3. The instructions use a **W** for word, a **H** for half-word and a **B** for byte operand size. The **LDA** and **STA** instruction implement word access to the physical memory computing a physical address to access.

The **LDR** and **STC** instructions support atomic operations. The LDR instruction loads a value form memory and remember this access. The STC instruction will store a value to the same location that the LDR instructions used and return a failure if the value was modified since the last LDR access. This CPU pipeline friendly pair of instructions allow to build higher level atomic operations on top.

Memory reference instructions can be issued when data translation is either on and off. When address translation is turned off, the memory reference instructions will ignore the segment part and replace it with a zero value. The address is the physical address. It is an architectural requirement that a virtual address with a segment Id of zero maps to an absolute address with the same offset. The absolute address mode instruction also works with translation turned on and off.

With the exception of the load reserved and store conditional instruction, memory reference instructions support a base register modification. When the option is set, the base register for an address computation will be adjusted before or after the data access. A negative offset will be added before the address computation, a positive offset after the address computation.

Immediate Instructions

Several instructions have within the instruction word bit fields for representing immediate values. Nevertheless, fixed length instruction word size architectures have one issue in that there is not enough room to embed a full word immediate value in the instruction. Typically, a combination of two instructions that concatenate the value from two parts is used. There are four instructions in this group.

The **LDIL** instruction will place an 22-bit value in the left portion of a register, padded with zeroes to the right. The register content is then paired with an instruction that sets the right most 10-bit value. The **LDO** instruction that computes an offset is an example of such an instruction. The **ADDIL** instructions add the left side of a register argument to a register. In combination, the two instructions LIDL and ADDIL allow to generate a 32-bit offset value.

The **LDSID** instruction will load the segment ID computed from the upper two bits of a logical address in to a general register.

Branch Instructions

The control flow instructions are divided into unconditional and conditional branch type instructions. The unconditional branch type instructions allow in addition to alter the control flow and to save the return point to a general register. A subroutine call would for example compute the target branch address and store this address + 4 in that register. Upon subroutine return, there is a branch instruction using this value to return via a general register content.

The **B** instruction is the unconditional IA-relative branch instruction within a program segment. It adds a signed offset to the current instruction address. Additionally the instruction saves a return link in a general register. The **BR** instruction behaves similar to the B instruction, except that a general register contains the instruction address relative offset.

The **BV** is an unconditional branch instruction using the segment base relative addressing mode. The branch address for the BV instruction is the segment relative offset computed using a segment relative base stored in a general register, adding a signed offset from another general register.

The **BE** and **BVE** instruction are the inter segment branches. The BE instruction branches to a segment relative address encoded in the instruction. In addition, a signed offset encoded in the instruction that is added to form the target segment offset. The return address segment part is stored in SR0 and the offset in a the general register. The BVE will use a general register containing a logical address to form the target segment relative and external branches may branch to pages with a different privilege level. When branching to a higher privilege level, an privilege execution trap is raised. A branch to a page with a lower privilege level will automatically demote the privilege level. The **GATE** instruction will promote the privilege level to the page on which the GATE instruction resides.

The conditional branch instructions combine a comparison operation with a local instruction address relative branch if the comparison or test condition is met. The **CBR** and **CBRU** instructions will compare two general registers for a condition encoded in the instruction. If the condition is met, an instruction address relative branch is performed. The target address is formed by adding an offset encoded in the instruction to the instruction address. Conditional branches are implemented with a static branch prediction scheme. Forward branches are predicted not taken, backward branches are predicted taken. The decision is predicted during the fetch and decode stage and if predicted correctly will result in no pipeline penalty.

Computational Instructions

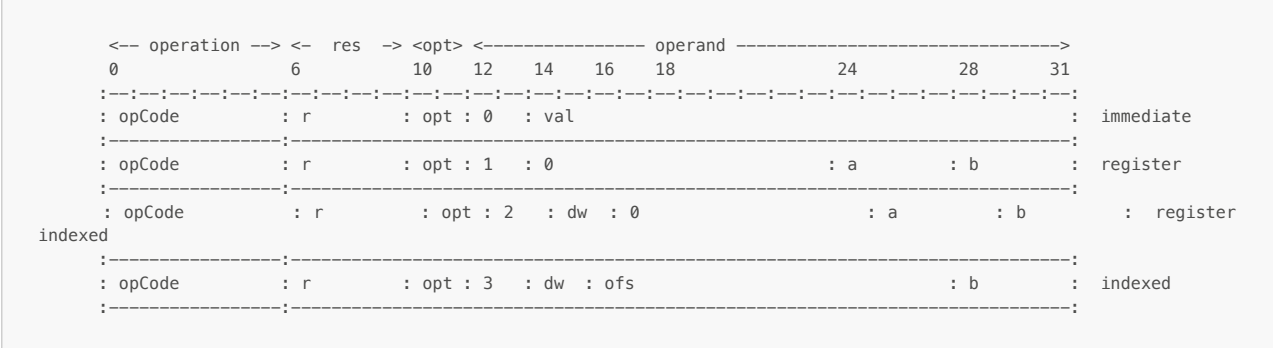
The arithmetic, logical and bit field operations instruction represent the computation type instructions. The computation instructions are divided into the numeric instructions **ADD**, **ADC**, **SUB** and **SBC**, the logical instructions **AND**, **OR**, **XOR** and the bit field operational instructions **EXTR**, **DEP** and **DSR**.

The numeric and logical instructions encode the second operand in the operand field. This allows for immediate values, register values and values accessed via their logical address. The numeric instructions allow for a carry/borrow bit for implementing multi-precision operations as well as the distinction between signed and unsigned operation overflow detection traps. The logical instructions allow to negate the operand as well as the output. The bit field instructions will perform bit extraction and deposit as well as double register shifting. These instruction not only implement bit field manipulation, they are also the base for all shift and rotate operations.

The **CMP** instruction compares two values for a condition and returns a result of zero or one. The **SHLA** instruction is a combination of shifting an operand and adding a value to it. Simple integer multiplication with small values can elegantly be done with the help of this instruction.

Operand Encoding for computational instructions

VCPU-32 features a register memory model, which means that one operand may come from memory. Some computational instructions use the **operand** format to specify the operand in the computation. The **operand mode** field in instructions that use operand modes is used to specify how the second operand is decoded. The instruction generally has the opCode in position 0 .. 5, the result register in position 6 .. 9 and an instruction specific option field in position 10..11. Bits 12..13 encode the operand mode. There are four operand modes, which are immediate value mode, register mode, and two addressing modes. The following figure gives an overview of the instruction layout for instructions with operand encoding.



The **immediate operand mode** supports a signed 18-bit value. The sign bit is in the rightmost position of the field. Depending on the sign, the remaining value in the field is sign extended or zero extended.

The **register modes** specify two registers "a" and "b" for the operation and store the result in "r". An example would be a the ADD instruction that adds the register content of "a" and "b" and stores the result into "r".

The machine addresses memory with a byte address. There are the indexed and register indexed operand mode. Operand mode 2 is the **register indexed address mode**, which will use a base register "b" and add an offset in "a" to it, forming the final byte address offset. The upper two bits select one of the segment registers SR4 .. SR7. The **dw** field specifies the data field width. A value of 0 represent a byte, 1 a half-word and 2 a word and 3 a double word. Option 3 is reserved and not implemented yet. The computed address must be aligned with the size of data to fetch.

The final operand mode is the **indexed address mode**. The offset field is a signed 12 bit field. The sign bit is in the rightmost position of the field. Depending on the sign, the remaining value in the field is sign extended or zero extended. The **dw** field specifies the data field width, just as in operand mode 2.

The operand address for the indexed and register indexed mode is built by adding an signed byte offset to the base register "b". The upper two bits select one of SR4 .. SR7.

System Control Instructions

The system control instructions are intended for the runtime designer to control the CPU resources such as TLB, Cache and so on. There are instruction to load a segment or control registers as well as instructions to store them. The TLB and the cache modules are controlled by instructions to insert and remove data in these two entities. Finally, the interrupt and exception system needs a place to store the address and instruction that was interrupted as well as a set of shadow registers to start processing an interrupt or exception. Most of the instructions found in this group require that the processor runs in privileged mode.

The **MR** instructions is used to transfer data to and from a segment register or a control register. The processor status instruction **MST** allows for setting or clearing an individual an accessible bit of the status part of the processor state.

The **LDPA** and **PRB** instructions are used to obtain information about the virtual memory system. The LDPA instruction returns the physical address of the virtual page, if present in memory. The PRB instruction tests whether the desired access mode to an address is allowed.

The TLB and Caches are managed by the **ITLB**, **PTLB** and **PCA** instruction. The ITLB and PTLB instructions insert into the TLB and removes translations from the TLB. The PCA instruction manages the instruction and data cache and features to flush and / or just purge a cache entry. There is no instruction that inserts data into a cache as this is completely controlled by hardware.

The **RFI** instruction is used to restore a processor state from the control registers holding the interrupted instruction address, the instruction itself and the processor status word. Optionally, general registers that have been stored into the reserved control registers will be restored.

The **DIAG** instruction is a control instructions to issue hardware specific implementation commands. Example is the memory barrier command, which ensures that all memory access operations are completed after the execution of this DIAG instruction. The **BRK** instruction is transferring control to the breakpoint trap handler.

Instruction Set Reference

The instructions described in this chapter contains the instruction word layout, a short description of the instruction and also a high level pseudo code of what the instruction does. Each instruction operation is described in a C-style pseudo code operation. The pseudo code uses a register notation describing by their class and index. For example, GR[5] labels general register 5, SR[2] represents the segment register number 2 and CR[1] the control register number 1.

In addition, some register also have dedicated names, such as for example the shift amount control register, labelled "shamt". A subfield of the instruction is selected by adding a "." and the field name in brackets. For example the carry bit in the PSW register is described as "PSW.[C]". The pseudo code routines used in the instruction descriptions are listed in the appendix.

Additional options for an instruction will be specified by a "." followed by a sequence of characters. Each character is one of the defined options for the instruction. As an example, the AND instruction can negate the result. The assembler notation would be "AND.N ...". The order of the individual characters does not matter. The defined options are listed as a list of characters.

Reserved fields in an instruction word are future enhancements and should be filled with a zero. The instruction explicitly shows the numeric value of these fields. Any other value at that place will result in an undefined behavior.

ADD

Adds the operand to the target register.

Format

```
ADDw [.<opt> ] r, operand        w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: ADDw    ( 0x10 ): r         :L :O : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The **ADD** instruction fetches the operand and adds it to the general register "r". The "L" bit set specifies an unsigned addition. If the "O" bit is set, a numeric overflow will cause an overflow trap. See the section on operand encoding for the defined operand modes. The operations will set the carry/borrow bits in the processor status word.

Operation

```
switch( opMode ) {

   case 0: tmpA <- GR[instr.[r]]; tmpB <- lowSignExtend( opArg, 18 ); break;

   case 1: tmpA <- GR[instr.[a]]; tmpB <- GR[instr.[b]]; break;

   case 2:
   case 3: {

      seg <- operandAdrSeg( instr );
      ofs <- operandAdrOfs( instr );
      len <- operandBitLen( instr );

      tmpA <- GR[instr.[r]];
      tmpB <- zeroExtend( memLoad( seg, ofs, len ), len );

   } break;
}

res <- tmpA + tmpB;

if ( instr.[O] && overflow ) overflowTrap( );
else {

   GR[instr.[r]] <- res;
   if ( ! instr.[L] ) PSW[C/B] <- carry/borrow Bits;
}
```

Exceptions

- Overflow trap
- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

None.

ADC

Adds the operand and carry bit to the target register.

Format

```
ADCw [.<opt> ] r, operand        w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: ADCw    ( 0x11 ): r         :L :O : operand                                                     :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The **ADC** instruction fetches the operand and adds it along with the carry bit of the processor status word to the general register "r". The "L" bit set specifies an unsigned addition. If the "O" bit is set, a numeric overflow will cause an overflow trap. See the section on operand encoding for the defined operand modes. The operations will set the carry/borrow bits in the processor status word.

Operation

```
switch( opMode ) {

   case 0: tmpA <- GR[instr.[r]]; tmpB <- lowSignExtend( opArg, 18 ); break;

   case 1: tmpA <- GR[instr.[a]]; tmpB <- GR[instr.[b]]; break;

   case 2:
   case 3: {

      seg <- operandAdrSeg( instr );
      ofs <- operandAdrOfs( instr );
      len <- operandBitLen( instr );

      tmpA <- GR[instr.[r]];
      tmpB <- zeroExtend( memLoad( seg, ofs, len ), len );

   } break;
}

res <- tmpA + tmpB + PWS.[C/B];

if ( instr.[O] && overflow ) overflowTrap( );
else {

   GR[instr.[r]] <- res;
   if ( ! instr.[L] ) PSW[C/B] <- carry/borrow Bits;
}
```

Exceptions

- Overflow trap
- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

The **ADC** is typically used to implement multi-precision addition.

ADDIL

Adds an immediate value left aligned into the target register.

Format

```
ADDIL r, val
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: ADDIL   ( 0x02 ): r         : val                                                               :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The add immediate left instruction loads a 22-bit immediate value padded with zeroes on the right left aligned to the general register "r" and place the result in general register one. Any potential overflows are ignored.

Operation

```
GR[0] = GR[instr.[r]] + ( instr.[val] << 10 );
```

Exceptions

None.

Notes

The ADDIL instruction is typically used to produce a 32bit address offset in combination with the load and store instruction. The following example will use a 32-bit offset for loading a value into general register one. GR11 holds a logical address. The ADDIL instruction will add the left 22-bit portion padded with zeroes to the right to GR11 and store the result in the scratch register R1. The instruction sequence

```
ADDIL  R11, L%ofs
LDW    R3, R%ofs(R1)
```

AND

Performs a bitwise AND of the operand and the target register and stores the result into the target register.

Format

```
ANDw [.<opt>] r, operand         w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: ANDw    ( 0x14 ): r         :N :C : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The instruction fetches the data specified by the operand and performs a bitwise AND of the general register "r" and the operand fetched. The result is stored in general register "r". The "N" bit negates the result making the AND a NAND operation. The "C" allows to complement (1's complement) the operand input, which is the "b" register. Using both "C" and "N" is an undefined operation. See the section on operand encoding for the defined operand modes.

Operation

```
switch( opMode ) {

   case 0: tmpA <- GR[instr.[r]]; tmpB <- lowSignExtend( opArg, 18 ); break;

   case 1: tmpA <- GR[instr.[a]]; tmpB <- GR[instr.[b]]; break;

   case 2:
   case 3: {

      seg <- operandAdrSeg( instr );
      ofs <- operandAdrOfs( instr );
      len <- operandBitLen( instr );

      tmpA <- GR[instr.[r]];
      tmpB <- zeroExtend( memLoad( seg, ofs, len ), len );

   } break;
}

if (( instr.[C]) && ( ~ ( instr.[C]))) tmpB <- ~ tmpB;
res <- tmpA & tmpB;

if ( instr.[N]) res <- ~ res;
GR[instr.[r]] <- res;
```

Exceptions

- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

Complementing the operand input allows to perform a bit clear in a register word by complementing the bit mask stored in the operand before performing the AND. Typically this is done in a program in two steps, which are first to complement the mask and then AND to the target variable. The C option allows to do this more elegantly in one

step.

B

Perform an unconditional IA-relative branch with a static offset and store the return address an a general register.

Format

```
B r, ofs
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: B       ( 0x20 ): r         : ofs                                                               :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The branch instruction performs a branch to an instruction address relative location. The target address is formed by shifting the low sign extended offset by 2 to the left and adding it to the current instruction offset. The virtual target address is built from the instruction address segment and offset. If code translation is disabled, the target address is the absolute physical address. The current instruction address offset + 4 is returned in general register "r". If code translation is disabled, the return value is the absolute physical address.

Operation

```
GR[instr.[r]]  <- PSW.[ofs] + 4;
PSW.[ofs]      <- PSW.[ofs] + lowSignExt(( instr.[ofs] << 2 ), 24 );
```

Exceptions

- Taken branch trap.

Notes

Using general register zero as the return link register will result in a branch instruction without saving the return link.

BE

Perform an unconditional external branch.

Format

```
BE r, ofs (a, b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BE      ( 0x24 ): r         : ofs                                   : a         : b         :
:-----------------:-------------------------------------------------------------------------:
```

Description

The branch external instruction branches to a segment relative location in another code segment. The target address is built from the segment register in field "a" and the base register "b" to which the low sign extended offset is added. If code translation is disabled, the offset is the absolute physical address and the "a" field ignored. The return offset is stored in "r", the return segment Id in SR0.

Since the BE instruction is a segment base relative branch, a branch to page with a different privilege level is possible. A branch from a lower level to a higher level result in an instruction protection trap. A branch from a higher privilege to a lower privilege level results in adjusting the privilege level in the status register. Otherwise, the privilege level remains unchained.

Operation

```
SR[0]          <- PSW.[seg];
GR[instr.[r]]  <- PSW.[ofs] + 4;

PSW.[seg] <- GR[instr.[a]].[16..31];
PSW.[ofs] <- GR[instr.[b]] + lowSignExt(( instr.[ofs] << 2 ), 16 );
```

Exceptions

- Taken branch trap
- Instruction memory protection trap

Notes

None.

BR

Perform an unconditional IA-relative branch with a dynamic offset stored in a general register and store the return address in a general register.

Format

```
BR r, b
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BR      ( 0x22 ): r         : 0                                                   : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The branch register instruction performs an unconditional IA-relative branch. The target address is formed adding the low sign extended content of register "b" shifted by two bits to the left to the current instruction address. If code translation is disabled, the target address is the absolute physical address. The current instruction address offset + 4 is returned in general register "r".

Operation

```
GR[instr.[r]]  <- PSW.[ofs] + 4;
PSW.[ofs]      <- PSW.[ofs] + ( GR[instr.[b]] << 2 );
```

Exceptions

- Taken branch trap

Notes

Using general register zero as the return link register will result in a branch instruction without saving the return link.

BRK

Trap to the debugger subsystem.

Format

```
BRK info1, info2
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BRK     ( 0x00 ): info1     : 0               : info2                                          :
:-----------------:--------------------------------------------------------------------------:
```

Description

The BRK instruction raises a debug breakpoint trap and enters the debug trap handler. The "info1" and "info2" field are passed to the debug subsystem. An exception is the BRK instruction with both info fields being zero. BRK 0, 0 is treated as a NOP instruction.

Operation

```
debugBreakpointTrap( instr.[info1], instr.[info2] );
```

Exceptions

- debug breakpoint trap

Notes

The instruction opCode for BRK is the opCode value of zero. A zero instruction word result is treated as a NOP. As an alternative option the the BRK 0, 0 could be implemented as an "illegal" instruction trap to detect instruction references to a zero content memory. The current VCPU-32 simulator interprets a zero instruction word as a NOP. To be decided.

BV

Perform an unconditional branch using a base and offset general register for forming the target branch address.

Format

```
BV r,(b)
BV (b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BV      ( 0x23 ): r         : 0                                                   : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The branch vectored instruction performs an unconditional branch to the base address in register in "b". The result is interpreted as an instruction address in the current code segment. This unconditional jump allows to reach the entire code address range. If code translation is disabled, the resulting offset is the absolute physical address. In addition, the current instruction offset + 4 is stored in "r".

Since the BV instruction is a segment base relative branch, a branch to page with a different privilege level is possible. A branch from a lower level to a higher level results in an instruction protection trap. A branch from a higher privilege to a lower privilege level results in the privilege level adjusted in the status register. Otherwise, the privilege level remains unchained.

Operation

```
GR[instr.[r]]   <- PSW.[ofs] + 4;
PSW.[ofs]       <- GR[instr.[b]];
```

Exceptions

- Taken branch trap
- Instruction memory protection trap

Notes

The **BV** instruction is typically used as a procedure return. The **B** instruction can leave a return link in a general register, which can directly be used by this instruction to return to the location after the **B** instruction.

BVE

Perform an unconditional external branch using a logical address and save the return address.

Format

```
BVE r, (a, b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BVE     ( 0x25 ): r         : 0                                     : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The branch and link vectored external instruction branches to an absolute location in another code segment. The target address is formed by adding general register "a" to the base register "b". The segment register is selected based on the upper two bits of general register "b". The return link is stored in general register "r", the return segment Id in SR0.

Since the BVE instruction is a segment base relative branch, a branch to page with a different privilege level is possible. A branch from a lower level to a higher level result in an instruction protection trap. A branch from a higher privilege to a lower privilege level results in adjusting the privilege level in the status register. Otherwise, the privilege level remains unchained.

Operation

```
SR[0]          <- PSW.[seg];
GR[instr.[r]]  <- PSW.[ofs] + 4;

PSW.[ofs] <- GR[instr.[b]] + GR[instr.[a]];
PSW.[seg] <- segSelect( PSW.[ofs] );
```

Exceptions

- Taken branch trap
- Instruction memory protection trap

Notes

None.

CBR, CBRU

Compare two registers and branch on condition.

Format

```
CBR .<cond> a, b, ofs
CBRU .<cond> a, b, ofs
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: CBR     ( 0x26 ):cond : ofs                                          : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: CBRU    ( 0x27 ):cond : ofs                                          : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The CBR and CUBR instructions compare general registers "a" and "b" for the condition specified in the "cond" field. The CBR instruction compares signed values, the CUBR unsigned values. If the condition is met, the target branch address is the low sign extended offset "ofs" shifted by two to the left and added to the current instruction address, otherwise execution continues with the next instruction. The conditional branch is predicted taken for a backward branch and predicted not taken for a forward branch.

Branch condition

The condition field is encoded as follows:

```
0 : EQ ( a == b )                          2 : NE  ( a != b )
1 : LT ( a < b, Signed )                   3 : LE ( a <= b, Signed )
```

Operation

```
switch( cond ) {

   case 0: res <- ( GR[instr.[a]] ==  GR[instr.[b]]);  break;
   case 1: res <- ( GR[instr.[a]] <   GR[instr.[b]]);  break;
   case 2: res <- ( GR[instr.[a]] !=  GR[instr.[b]]);  break;
   case 3: res <- ( GR[instr.[a]] <=  GR[instr.[b]]);  break;
}

if ( res ) PSW.[ofs] <- PSW.[ofs] + lowSignExt(( instr.[ofs] << 2 ), 18 );
else       PSW.[ofs] <- PSW.[ofs] + 4;
```

Exceptions

None.

Notes

Often a comparison is followed by a branch in an instruction stream. VCPU-32 therefore features conditional branch instruction that offers the combination of a register evaluation and branch depending on the condition specified. The condition field does not encode a greater or greater-equal condition. This can easily be done by reversing the registers and inverting the comparison condition.

CMP, CMPU

Compares a register and an operand and stores the comparison result in the target register.

Format

```
CMPw .cond> r, operand         w = B|H|W
CMPUw .cond> r, operand        w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: CMPw    ( 0x17 ): r         :cond : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
: CMPUw   ( 0x18 ): r         :cond : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The compare instructions will compare two operands for the condition specified in the "cond" field. A value of one is stored in "r" when the condition is met, otherwise a zero value is stored. A typical code pattern would be to issue the compare instruction with the comparison result in general register "r", followed by a conditional branch instruction. See the section on operand encoding for the defined operand modes. The **CMP** operates on signed values, the **CMPU** operates on unsigned values.

Comparison condition

The compare condition are encoded as follows.

```
0 : EQ ( r == operand )                          2 : NE ( r != operand )
1 : LT ( r <  operand, Signed )                  3 : LE ( r <= operand, Signed )
```

Operation

```
switch( opMode ) {

   case 0: tmpA <- GR[instr.[r]]; tmpB <- lowSignExtend( opArg, 18 ); break;

   case 1: tmpA <- GR[instr.[a]]; tmpB <- GR[instr.[b]]; break;

   case 2:
   case 3: {

      seg <- operandAdrSeg( instr );
      ofs <- operandAdrOfs( instr );
      len <- operandBitLen( instr );

      tmpA <- GR[instr.[r]];
      tmpB <- zeroExtend( memLoad( seg, ofs, len ), len );

   } break;
}

switch ( cond ) {

   case 0: res <- tmpA ==  tmpB; break;
   case 1: res <- tmpA <   tmpB; break;
   case 2: res <- tmpA !=  tmpB; break;
   case 3: res <- tmpA <=  tmpB; break;
}

GR[instr.[r]] <- res;
```

Exceptions

- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Alignment trap

Notes

The condition field does not encode a greater or greater-equal condition. This can easily be done by reversing the registers and inverting the comparison condition.

CMR

Test a general register for a condition and conditionally move a register value to the target register.

Format

```
CMR [.<cond>] r, a, b
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: CMR     ( 0x09 ): r         :cond       : 0                           : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The conditional move instruction will test register "b" for the condition. If the condition is met, general register "a" is moved to "r".

Comparison condition codes

```
0 : EQ ( b == 0 )                           2 : NE ( b != 0 )
1 : LT ( b < 0, Signed )                    3 : LE ( b <= 0, Signed )
4 : GT ( b > 0, Signed )                    5 : GE ( b >= 0, Signed )
6 : HI ( b > 0, Unsigned )                  7 : HE ( b >= 0, Unsigned )

... 8 .. 15 to be defined: w.g. ODD, EVEN, Left half zero, right half zero, etc.
```

Operation

```
switch( instr.[cond] ) {

   case 0: res <- ( GR[instr.[b]] ==  0 ); break;
   case 1: res <- ( GR[instr.[b]] <   0 ); break;
   case 2: res <- ( GR[instr.[b]] !=  0 ); break;
   case 3: res <- ( GR[instr.[b]] <=  0 ); break;
   case 4: res <- ( GR[instr.[b]] >   0 ); break;
   case 5: res <- ( GR[instr.[b]] >=  0 ); break;
   case 6: res <- ( GR[instr.[b]] >U  0 ); break;
   case 7: res <- ( GR[instr.[b]] >=U 0 ); break;

   ... add more cases ...

   default: illegalInstrTrap( );
}

if ( res ) GR[instr.[r]] <- GR[instr.[a]];
```

Exceptions

None.

Notes

In combination with the CMP instruction, simple instruction sequences such as "if (a < b) c = d" could be realized without pipeline unfriendly branch instructions.

```
Example:

C-code:     if ( a < b ) c = d

Assumption: ( a -> R2, b -> R6,c -> R1, d -> R4 )

CMP.LT R1,R2,R6  ; compare R2 < R6 and store a zero or one in R1
CMR    R1,R4,R1  ; test R1 and store R4 when condition is met
```

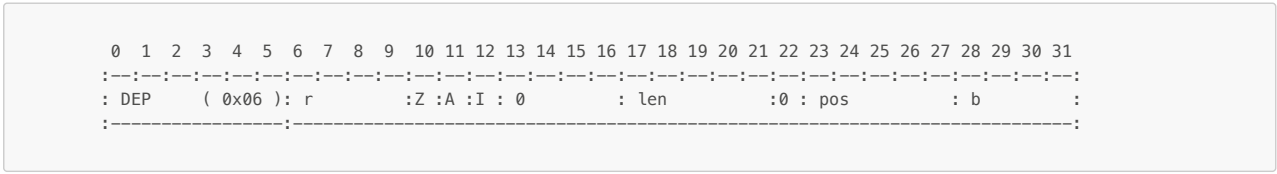
The CMR instruction is a rather specialized instruction. It highly depends on a good peephole optimizer to detect such a situation. The instruction sequence might also be a candidate for a compare and move instruction

DEP

Performs a bit field deposit of the value extracted from a bit field in reg "B" and stores the result in the targetReg.

Format

```
DEP [.<opt>] r, b, pos, len
```



Description

The instruction deposits the bit field of length "len" in general register "b" into the general register "r" at the specified position. The "pos" field specifies the rightmost bit for the bit field to deposit. The "len" field specifies the bit size if the field to deposit. The "Z" bit clears the target register "r" before storing the bit field, the "I" bit specifies that the instruction bits 28..31 contain an immediate value instead of a register. If the "A" bit is set, the shift amount control register is used for obtaining the position value.

Operation

```
if ( instr.[Z] ) GR[instr.[r]] <- 0;

if ( instr.[A] ) tmpPos <- SHAMT.[27..31];
else             tmpPos <- instr.[pos];

if ( instr.[I] ) tmpB <- instr.[28..31];
else             tmpB <- GR[instr.[b]];

deposit( GR[instr.[r]], tmpB, tmpPos, instr.[len] );
```

Exceptions

None.

Notes

The VCPU-32 instruction set does not have dedicated instructions for left and right shift operations. They can easily be realized with the EXTR and DEP instructions. Refer to the chapter for synthetic instructions.

DIAG

Issues commands to hardware specific components and implementation features.

Format

```
DIAG r, a, b, info
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: DIAG    ( 0x3E ): r         : info      : 0                           : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The DIAG instruction sends a command to an implementation hardware specific components. The instruction accepts two argument registers "a" and "b" and returns a result in "r". The meaning of the "info" field and the input and output arguments are processor implementation dependent and described in the respective processor documentation.

Operation

```
if ( ! PSW.[P] ) privilegedOperationTrap( );

... perform the requested operation using instr.[info], GR[instr.[a]] and  GR[instr.[b]]
```

Exceptions

- privileged operation trap

Notes

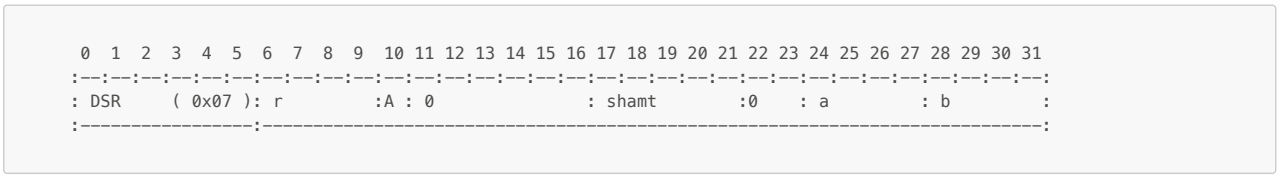
None.

DSR

Performs a right shift of two concatenated registers for shift amount bits and stores the result in the target register.

Format

```
DSR [.A] r, b, a, shAmt
```



Description

The double shift right instruction concatenates the general registers specified by "b" and "a" and performs a right shift operation of "shamt" bits. register "b" is the left part, register "a" the right part. The lower 32 bits of the result are stored in the general register "r". The "shamt" field range is from 0 to 31 for the shift amount. If the "A" bit is set, the shift amount is taken from the shift amount control register.

Operation

```
if ( instr.[A] )  tmpShAmt <- SHAMT.[27..31];
else              tmpShAmt <- instr.[shamt];

GR[instr.[r]] <- rshift( cat( GR[instr.[b]], tmp ), tmpShamt );
```

Exceptions

None.

Notes

The VCPU-32 instruction set does not have dedicated instructions for shift and rotate operations. They can easily be realized with the DSR instructions. Refer to the chapter for synthetic instructions.

EXTR

Performs a bit field extract from a general register and stores the result in the targetReg.

Format

```
EXTR [.<opt>] r, b, pos, len
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: EXTR    ( 0x05 ): r         :S :A :0             : len          :0 : pos          : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The instruction performs a bit field extract specified by the position and length instruction data from general register "b". The "pos" field specifies the rightmost bit of the bitfield to extract. The "len" field specifies the bit size if the field to extract. The extracted bit field is stored right justified in the general register "r". If set, the "S" bit allows to sign extend the extracted bit field. If the "A" bit is set, the shift amount control register is used for obtaining the position value instead of the instruction field.

Operation

```
if ( instr.[A] ) tmpPos <- SHAMT.[27..31];
else             tmpPos <- instr.[pos];

GR[instr.[r]] <= extract( GR[instr.[b]], tmpPos, instr.[len] );

if ( instr.[S] ) signExtend( GR[instr.[r]], instr.[len] );
else             zeroExtend( GR[instr.[r]], instr.[len] );
```

Exceptions

None.

Notes

The VCPU-32 instruction set does not have dedicated instructions for left and right shift operations. They can easily be realized with the EXTR and DEP instructions. Refer to the chapter for synthetic instructions.

GATE

Perform an instruction address relative branch and change the privilege level.

Format

```
GATE r, ofs
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: GATE    ( 0x21 ): r         : ofs                                                              :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The GATE instruction computes the target address by shifting the low sign extended offset by 2 bits adding it to the current instruction address offset. If code translation is enabled, the privilege level is changed to the privilege field in the TLB entry for the page from which the GATE instruction is fetched. The change is only performed when it results in a higher privilege level, otherwise the current privilege level remains. If code translation is disabled, the privilege level is set to zero. The privilege level of the gateway instruction is deposited in GR "r". Execution continues at the target address with the new privilege level.

Operation

```
tmpOfs = PSW.[ofs] + lowSignExt(( instr.[ofs] << 2 ), 24 );

if ( PSW.[C] ) {

   searchInstructionTlbEntry( PSW.[seg], tmpOfs, &entry );

   if ( entry.[PageType] == 3 ) PSW.[P] <- entry.[PL1];

} else ST.[P] <- 0;

PSW.[ofs]      <- tmpOfs;
GR[instr.[r]]  <- entry.[PL1];
```

Exceptions

- Instruction memory protection trap

Notes

The access rights field in of the gateway page defines the potential privilege change when executing the gateway instruction. PL1 can be seen as an entry privilege level. If the current privilege level is lower than PL1 a trap is raised. PL2 is the promotion level. If the current privilege level is lower than the PL2 level, it will be raised to the PL2 level. Note that zero is the highest level and one is the lowest level.

Access Right PL1	Access Right PL2	current PrivLevel	new PrivLevel
0	0	0	0
0	x	1	trap
1	1	1	1
1	0	x	0
1	1	0	0

ITLB

Inserts a translation into the instruction or data TLB.

Format

```
ITLB [.<opt>] r, (a, b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: ITLB    ( 0x3B ): r         :T                                      : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The ITLB instruction inserts a translation into the instruction or data TLB. The virtual address is encoded in "a" for the segment register and "b" for the offset.

The "T" bit specifies whether the instruction or the data TLB is addressed. A value of zero references the instruction TLB, a value of one refers to the data TLB. The "r" register contains the TLB information data.

Argument Word Layout

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
:V :T :D :B :P : 0      : AR        : PPN                                                        :
:-------------------------------------------------------------------------------------------------:
```

Operation

```
if ( ! PSW.[P] ) privilegedOperationTrap( );

if ( instr.[T] ) {

   if ( ! searchDataTlbEntry( SR[a], GR[instr.[b]], &entry ))
      allocateDataTlbEntry( SR[a], GR[instr.[b]], &entry );
}
else {

   if ( ! searchInstructionTlbEntry( SR[a], GR[instr.[b]], &entry ))
      allocateInstructionTlbEntry( SR[a], GR[instr.[b]], &entry );
}

entry.[T]      <- GR[instr.[r]].[T];
entry.[D]      <- GR[instr.[r]].[D];
entry.[B]      <- GR[instr.[r]].[B];
entry.[P]      <- GR[instr.[r]].[P];
entry.AR       <- GR[instr.[r]].AR;
entry.[PPN]    <- GR[instr.[r]].[PPN];
entry.[V]      <- 1;
```

Exceptions

- Privileged operation trap

Notes

None.

LD

Loads a memory value into a general register using a logical address.

Format

```
LDw [.M] r, ofs([s,] b)          w = B|H|W
LDw [.M] r, a([s,] b)            w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDw     ( 0x30 ): r         :0 :M : seg : dw  : ofs                              : b         :
:-----------------:-------------------------------------------------------------------------------:
: LDw     ( 0x30 ): r         :1 :M : seg : dw  : 0                     : a        : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The load instruction will load the operand into the general register "r". The offset is computed by adding the sign extended or the general register "a" offset to "b". The "seg" field selects the segment register. A zero will use the upper two bits of the computed address offset to select among SR4..SR7. Otherwise SR1..SR3 are selected. The "dw" field specifies the data length. From 0 to 3 the data length is byte, half word, word and double. The double option is reserved for future use. The computed offset must match the alignment size of the data to fetch. The "M" bit indicates base register increment. If set, a negative value in the "ofs" field or negative content "a" will add the offset to the base register before the memory access, otherwise after the memory access.

Operation

```
if ( instr.[10] ) {

   if ( instr.[M] ) {

      if ( GR[instr.[a]] < 0 ) tmpOfs = GR[instr.[b]] + GR[instr.[a]];
      else                     tmpOfs = GR[instr.[b]];
   }
   else tmpOfs = GR[instr.[b]] + GR[instr.[a]];
}
else {

   if ( instr.[M] ) {

      if ( lowSignExtend( ofs, 12 ) < 0 ) tmpOfs = GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
      else                                tmpOfs = GR[instr.[b]];
   }
   else tmpOfs = GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
}

len = dataLen( instr.[seg] );

if ( instr.[seg] == 0 ) seg = segSelect( tmpOfs );
else                    seg = instr.[seg];

GR[instr.[r]] <- zeroExtend( memLoad( seg, tmpOfs, len ), len );

if ( instr.[10] ) {

   if ( instr.[10] ) GR[instr.[b]] <- GR[instr.[b]] + GR[instr.[a]];
   else              GR[instr.[b]] <- GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
}
```

Exceptions

- Illegal instruction trap
- DTLB miss/data page fault

- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

The load instruction allows to load a half word or byte. The result is zero sign extended. When the memory location represents a signed value, a sign-extend needs to be performed explicitly.

```
LDH r5, 100(R4)
EXTR.S r5, r5, 31, 16
```

LDA

Loads the memory content into a general register using an absolute address.

Format

```
LDA [.M] r, ofs(b)
LDA [.M] r, a(b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDA     ( 0x32 ): r         :0 :M : 0   : 2   : ofs                              : b         :
:-----------------:-------------------------------------------------------------------------------:
: LDA     ( 0x32 ): r         :1 :M : 0   : 2   : 0                      : a        : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The load absolute instruction will load the content of the physical memory address into the general register "r". The absolute 32-bit address is computed by adding the signed offset or the general register "a" to general register "b". The "M" bit indicates base register increment. If set, a negative value in the "ofs" field or negative content "a" will add the offset to the base register before the memory access, otherwise after the memory access. The LDwA instructions is a privileged instructions.

Operation

```
if ( ! ST.[ PRIV ] ) privilegedOperationTrap( );

if ( instr.[10] ) {

   if ( instr.[M] ) {

      if ( GR[instr.[a]] ) offset = GR[instr.[b]] + GR[instr.[a]];
      else                 offset = GR[instr.[b]];
   }
   else offset = GR[instr.[b]] + GR[instr.[a]];
}
else {

   if ( instr.[M] ) {

      if ( lowSignExtend( ofs, 12 ) < 0 ) offset = GR[instr.[b]] + lowSignExtend( ofs, 12 );
      else                                offset = GR[instr.[b]];
   }
   else offset = GR[instr.[b]] + lowSignExtend( inatr.[ofs], 12 );
}


GR[instr.[r]] <- memLoad( 0, offset, 32 );

if ( instr.[10] ) GR[instr.[b]] <- GR[instr.[b]] + GR[instr.[a]];
else              GR[instr.[b]] <- GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
```

Exceptions

- Privileged operation trap
- Data alignment trap

Notes

None.

LDIL

Loads an immediate value left aligned into the target register.

Format

```
LDIL r, val
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDIL    ( 0x01 ): r         : val                                                               :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The load immediate left instruction loads a 22-bit immediate value left aligned into the general register "r", padded with zeroes on the right.

Operation

```
GR[instr.[r]] = val << 10;
```

Exceptions

None.

Notes

The LDIL instruction will place an 22-bit value in the left portion of a register, padded with zeroes to the right. The register content is then paired with an instruction that sets the right most 10-bit value. The LDO instruction that computes an offset is an example of such an instruction. The instruction sequence

```
LDIL  r10, L%val
LDO   r2, R%val(r10)
```

will load the left hand side of the 32-bit value "val" into R10 and use the LDO instruction to add the right side part of "val"". The result is stored in R2.

LDO

Loads an offset into general register.

Format

```
LDO r, ofs(b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDO     ( 0x03 ): r         : ofs                                                 : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The LDO instruction loads an offset into general register "r". The offset is computed by adding the low sign extended offset value to the general base register "b". Memory is not referenced.

Operation

```
GR[instr.[r]] <- GR[instr.[b]] + lowSignExt( instr.[ofs], 18 );
```

Exceptions

None.

Notes

The LDO instruction can also be used to load an immediate value into a register, when using general register zero as the base. The following example load the value -100 into general register R2.

```
LDO r2, -100(r0)
```

LDPA

Load the physical address for a virtual address.

Format

```
LDPA r, a([s,] b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDPA    ( 0x39 ): r         : 0   : seg : 0                          : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The LDPA instruction returns the physical address for a logical or virtual address into register "r" if the page is main memory. The logical address is formed by adding general register "a" to general register "b" and using the segment selector "seg" for selecting the segment. If the page is physical memory, the offset is returned in "r", otherwise a zero is returned. The result is ambiguous for a virtual address that translates to a zero address. LDPA is a privileged operation.

Operation

```
if ( ! PSW.[P] ) privilegedOperationTrap( );

tmpOfs <- GR[instr.[b]] + GR[instr.[a]];

if ( instr.[seg] == 0 ) seg = segSelect( tmpOfs );
else                    seg = instr.[seg];

GR[instr.[r]] <- loadPhysAdr( seg, tmpOfs );
```

Exceptions

- privileged operation trap
- non-access data TLB miss / page fault

Notes

None.

LDR

Loads the operand into the target register from the address and marks that address.

Format

```
LDR r, ofs([s,] b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDR     ( 0x34 ): r         : 0   : seg : 2   : ofs                               : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The LDR instruction is used for implementing semaphore type operations. The first part of the instruction behaves exactly like the LDW instruction where a virtual address is computed. Next, the memory content is loaded into general register "r". The second part remembers the address and will detect any modifying reference to it. The LDR instruction supports only word addressing.

Operation

```
tmpOfs = GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );

if ( tmpOfs.[30.31] != 0 ) dataAlignmentTrap( );

if ( instr.[seg] == 0 ) seg = segSelect( tmpOfs );
else                    seg = instr.[seg];

GR[instr.[r]] <- memLoad( seg, tmpOfs, 32 );

lrValid = true;
lrArg   = GR[instr.[r]];
```

Exceptions

- illlegal instruction trap
- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

The "remember the access part" is highly implementation dependent. One option is to implement this feature in the L1 data cache. Under construction...

LSID

Load the segment identifier for a logical address.

Format

```
LSID r, b
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LSID    ( 0x04 ): r         : 0                                                   : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The LSID instruction returns the segment identifier of the segment encoded in the logical address in general register "b".

Operation

```
GR[instr.[r]] <- SR[ segSelect( GR[instr.[b]] ) ];
```

Exceptions

None.

Notes

None.

MR

Copy data between a general register and a segment or control register.

Format

```
MR r, s
MR s, r
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: MR      ( 0x0A ): r         :D :M : 0                                     : s               :
:-----------------:-------------------------------------------------------------------------:
```

Description

The move register instructions copy data from segment or control register "s" to a general register "r" and vice versa. The "D" field indicates the copy direction. A value of one will copy general register "r" to segment or control register "s". A value of zero copies "r" from segment or control register "s". The "M" field identifies the register type. A value of one refers to a control register, otherwise segment register. Setting a value for a privileged segment or control register is a privileged operation.

Operation

```
if ( instr.[D] ) {

   if ( instr.[M] )  CR[ instr.[27..31]] <- GR[instr.[r]];
   else              SR[ instr.[29..31]] <- GR[instr.[r]];

} else {

   if ( instr.[M] )  GR[instr.[r]] <- CR[ instr.[27..31]];
   else              GR[instr.[r]] <- SR[ instr.[29..31]];
}
```

Exceptions

- privileged operation trap.

Notes

None.

MST

Set and clears bits in the processor status word.

Format

```
MST b
MST.S val
MST.C val
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: MST     ( 0x0B ): r         :mode : 0                                             : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The MST instruction sets the status bits 28.. 31 of the processor status word. The previous bit settings are returned in "r". There are three modes. Mode 0 will replace the user status bits 28..31 with the bits 28..31 in general register "b". Mode 1 and 2 will interpret the 4-bit field bits 28..31 as the bits to set or clear with a bit set to one will perform the set or clear operation. Modifying the status register is a privileged instruction.

Mode

```
0 - copy status bits using general register "b" ( bits 28..31 )
1 - set status bits using the bits in "b" ( bits 28..31 )
2 - clears status bits using the bits in "b" ( bits 28..31 )
3 - undefined.
```

Operation

```
if ( ! PSW.[P] ) privilegedOperationTrap( );

GR[instr.[r]] <- cat( 0.[0..27], ST.[28..31] );

switch( mode ) {

   case 0: {

       ST.[28..31] <- GR[instr.[b]].[28..31];

   } break;

   case 1: {

      if ( instr.[28] ) PSW.[12] <- 1;
      if ( instr.[29] ) PSW.[13] <- 1;
      if ( instr.[30] ) PSW.[14] <- 1;
      if ( instr.[31] ) PSW.[15] <- 1;

   } break;

   case 2: {

      if ( instr.[28] ) PSW.[12] <- 0;
      if ( instr.[29] ) PSW.[13] <- 0;
      if ( instr.[30] ) PSW.[14] <- 0;
      if ( instr.[31] ) PSW.[15] <- 0;

    } break;

   default: illegalInstructionTrap( );
}
```

Exceptions

- privileged operation trap

Notes

None.

OR

Performs a bitwise OR of the operand and the target register and stores the result into the target register.

Format

```
ORw[.<opt>] r, operand         w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: ORw      ( 0x15 ): r         :N :C : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The instruction fetches the data specified by the operand and performs a bitwise OR of the general register "r" and the operand fetched. The result is stored in general register "r". The N bit negates the result making the AND a NAND operation. The C allows to complement (1's complement) the operand input, which is the "b" register. Using both C and N is an undefined operation. See the section on operand encoding for the defined operand modes.

Operation

```
switch( opMode ) {

   case 0: tmpA <- GR[instr.[r]]; tmpB <- lowSignExtend( opArg, 18 ); break;

   case 1: tmpA <- GR[instr.[a]]; tmpB <- GR[instr.[b]]; break;

   case 2:
   case 3: {

      seg <- operandAdrSeg( instr );
      ofs <- operandAdrOfs( instr );
      len <- operandBitLen( instr );

      tmpA <- GR[instr.[r]];
      tmpB <- zeroExtend( memLoad( seg, ofs, len ), len );

   } break;
}

if (( instr.[C]) && ( ~ ( instr.[C]))) tmpB <- ~ tmpB;
res <- tmpA | tmpB;

if ( instr.[N]) res <- ~ res;
GR[instr.[r]] <- res;
```

Exceptions

- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

Using general register zero as one operand will result in OR-ing a zero with a general register, which is a copy of a register to general register "r".

PCA

Flush and / or remove cache lines from the cache.

Format

```
PCA [.<opt>] a (b)
PCA [.<opt>] a (s, b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: PCA     ( 0x3D ): 0         :T :M : seg :F : 0                       : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The PCA instruction flushes or purges a cache line from the instruction or data cache. The "seg" field selects the segment register for forming the virtual address. A zero will use the upper two bits of the computed address offset to select among SR4..SR7. Otherwise SR1..SR3 are selected.

The "T" bit indicates whether the instruction or the data cache is addressed. A value of zero references the instruction cache. An instruction cache line can only be purged, a data cache line can also be written back to memory when dirty and then optionally purged.

The "M" bit indicates base register increment. If set, a negative value in general register "a" will form the address by adding the content to the base register "b" before the cache operation, otherwise after the cache operation.

The "F" bit will indicate whether the data cache is to be purged without flushing it first to memory. If "F" is one, the entry is first flushed and then purged, else just purged. The "F" bit has no meaning for an instruction cache.

Operation

```
 if ( instr.[M] ) {

    if ( GR[instr.[a]] < 0 ) tmpOfs = GR[instr.[b]] + GR[instr.[a]];
    else                     tmpOfs = GR[instr.[b]];
 }
 else tmpOfs = GR[instr.[b]] + GR[instr.[a]];

 if ( instr.[seg] == 0 ) seg = segSelect( tmpOfs );
 else                    seg = instr.[seg];

if ( instr.[T] ) {

    if ( instr.[F] ) flushDataCache( seg, tmpOfs );
    purgeDataCache( seg, tmpOfs );

 } else purgeInstructionCache( seg, tmpOfs );

 if ( instr.[M] ) GR[instr.[b]] <- GR[instr.[b]] + GR[instr.[a]];
```

Exceptions

- Privileged operation trap
- Non-access ITLB trap
- Non-access DTLB trap

Notes

None.

PRB

Probe data access to a virtual address.

Format

```
PRB [.<opt>] r, ([s,] b)
PRB [.<opt>] r, ([s,] b), a
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: PRB     ( 0x3A ): r         :W :I : seg : 0                          : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The PRB instruction determines whether data access at the requested privilege level. If the probe operation succeeds, a value of one is returned in "r", otherwise a zero is returned. The "seg" field selects the segment register for forming the virtual address. A zero will use the upper two bits of the computed address offset to select among SR4..SR7. Otherwise SR1..SR3 are selected.

The "W" bit specifies whether a read or write access is requested. A value of zero is a read access, a value of one a read/write access.

The "I" bit designates the privilege level to use for the probe operation. A value of zero specifies that the privilege level is stored register "a". A value of one interprets bit 27 as the privilege level.

If protection checking is enabled, the protection Id is checked as well. The instruction performs the necessary virtual to physical data translation regardless of the processor status bit for data translation.

Operation

```
if ( instr.[seg] == 0 ) seg = segSelect( GR[instr.[b]] );
else                    seg = instr.[seg];

if ( ! searchDataTlbEntry( SR[seg], ofs, &entry )) {

   if ( instr.[I] ) {

      if      ((   instr.[W] ) && ( writeAccessAllowed( entry, instr.[27] )))  GR[instr.[r]] <- 1;
      else if (( ! instr.[w] ) && ( readAccessAllowed( entry, instr.[27] )))   GR[instr.[r]] <- 1;
      else                                                                     GR[instr.[r]] <- 0;

   } else {

      if      ((   instr.[W] ) && ( writeAccessAllowed( entry, GR[instr.[a]] )))    GR[instr.[r]] <- 1;
      else if (( ! instr.[W] ) && ( readAccessAllowed( entry, GR[instr.[a]] )))     GR[instr.[r]] <- 1;
      else                                                                          GR[instr.[r]] <- 0;
   }

} else nonAccessDataTlbMiss( );
```

Exceptions

- non-access data TLB miss trap

Notes

None.

PTLB

Removes a translation entry from the TLB.

Format

```
PTLB [.<opt>] a (b)
PTLB [.<opt>] a (s, b)
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: PTLB    ( 0x3C ): 0         :T :M : seg : 0                         : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The PTLB instruction removes a translation from the instruction or data TLB by marking the entry invalid. The "seg" field selects the segment register for forming the virtual address. A zero will use the upper two bits of the computed address offset to select among SR4..SR7. Otherwise SR1..SR3 are selected.

The "T" bit indicates whether the instruction or the data TLB is addressed. A value of zero references the instruction TLB, a value of one refers to the data TLB.

The "M" bit indicates base register increment. If set, a negative value in general register "a" will form the address by adding the content to the base register "b" before the TLB operation, otherwise after the TLB operation.

Operation

```
if ( ! PSW.[P] ) privilegedOperationTrap( );

if ( instr.[M] ) {

   if ( GR[instr.[a]] < 0 ) tmpOfs = GR[instr.[b]] + GR[instr.[a]];
   else                     tmpOfs = GR[instr.[b]];
}
else tmpOfs = GR[instr.[b]] + GR[instr.[a]];

if ( instr.[seg] == 0 ) seg = segSelect( tmpOfs );
else                    seg = instr.[seg];

if ( instr.[T] ) {

   if ( searchDataTlbEntry( SR[seg], tmpOfs, &entry ))
      purgeDataTlbEntry( SR[seg], tmpOfs, &entry );
}
else {

   if ( searchInstructionTlbEntry( SR[seg], tmpOfs, &entry ))
      purgeInstructionTlbEntry( SR[seg], tmpOfs, &entry );
}

if ( instr.[M] ) GR[instr.[b]] <- GR[instr.[b]] + GR[instr.[a]];
```

Exceptions

- Privileged operation trap

Notes

None.

RFI

Restore the processor state and restart execution.

Format

```
RFI
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: RFI     ( 0x3F ): 0                                                                           :
:-----------------:---------------------------------------------------------------------------:
```

Description

The RFI instruction restores the instruction address segment, instruction address offset and the processor status register from the control registers I-PSW-0 and I-PSW-1.

Operation

```
if ( ! PSW.[P] ) privilegedOperationTrap( );

PSW-0 <- I-PSW-0;
PSW-1 <- I-PSW-1;
```

Exceptions

- privileged operation trap

Notes

The RFI instruction is also used to perform a context switch. Changing from one task to another is accomplished by loading the control registers **I-PSW-0** and **I-PSW-1** and then issue the RFI instruction. Setting bits other than the system mask is generally accomplished by constructing the respective status word and then issuing an RFI instruction to set them.

There could be an option to also restore some of the general a segment registers from a set of shadow registers to which they have been saved when the interrupt occurred. The current implementation does not offer this option.

SBC

Subtracts the operand and the carry/borrow bit from the target register.

Format

```
SBC[.<opt>] r, operand        w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: SBCw    ( 0x13 ): r         :L :O : operand                                                     :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The instruction fetches the operand and subtracts it along with the carry bit from the general register "r". The "L" bit set specifies an unsigned subtraction. If the "O" bit is set, a numeric overflow will cause an overflow trap. See the section on operand encoding for the defined operand modes. The operations will set the carry/borrow bits in the processor status word.

Operation

```
switch( opMode ) {

   case 0: tmpA <- GR[instr.[r]]; tmpB <- lowSignExtend( opArg, 18 ); break;

   case 1: tmpA <- GR[instr.[a]]; tmpB <- GR[instr.[b]]; break;

   case 2:
   case 3: {

      seg <- operandAdrSeg( instr );
      ofs <- operandAdrOfs( instr );
      len <- operandBitLen( instr );

      tmpA <- GR[instr.[r]];
      tmpB <- zeroExtend( memLoad( seg, ofs, len ), len );

   } break;
}

res <- tmpA - tmpB + PSW.[C/B];

if ( instr.[O] && overflow ) overflowTrap( );
else {

   GR[instr.[r]] <- res;
   PSW[C/B] <- carry/borrow Bits;
}
```

Exceptions

- Overflow trap
- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

The **SBC** is typically used to implement multi-precision subtraction.

SHLA

Performs a combined shift left and add operation and stores the result into the target register.

Format

```
SHLA [.<opt>] r, a, b, smAmt
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: SHLA    ( 0x08 ): r         :L :O : 0                    : sa  : 0   : a         : b         :
:-----------------:-------------------------------------------------------------------------:
```

Description

The shift left and add instruction will shift general register "a" left by the bits specified in the "shamt" field, add the content of general register "b" to it and store the result in general register "r". This combined operation allows for an efficient multiplication operation for multiplying a value with a small integer value.

The "L" bit indicates that this is an operation on unsigned quantities.

The "O" bit is set to raise a trap if the instruction either shifts beyond the number range of general register "r" or when the addition of the shifted general register "a" plus the value in general register "b" results in an overflow.

Operation

```
GR[instr.[r]] <- ( GR[instr.[a]] << instr.[sa] ) + GR[instr.[b]];

if ( instr.[O] ) && ( ovl ) overflowTrap( );
```

Exceptions

- Overflow trap

Notes

None.

ST

Stores a general register value into memory using a logical address.

Format

```
STw [.M] ofs ([s,] b), r         w = B|H|W
STw [.M] a ([s,] b), r           w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: STw     ( 0x31 ): r         :0 :M : seg : dw  : ofs                              : b         :
:-----------------:-------------------------------------------------------------------------------:
: STw     ( 0x31 ): r         :1 :M : seg : dw  : 0                    : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The store instruction will store the data in general register "r" to memory. The offset is computed by adding the sign extended offset of general register "a" to "b". The "seg" field selects the segment register. A zero will use the upper two bits of the computed address offset to select among SR4..SR7. Otherwise SR1..SR3 are selected.

The "dw" field specifies the data length. From 0 to 3 the data length is byte, half word, word and double. The double option is reserved for future use. The computed offset must match the alignment size of the data to fetch.

The "M" bit indicates base register increment. If set, a negative value in the "ofs" field or negative content "a" will add the offset to the base register before the memory access, otherwise after the memory access.

Operation

```
if ( instr.[10] ) {

   if ( instr.[M] ) {

      if ( GR[instr.[a]] < 0 ) tmpOfs = GR[instr.[b]] + GR[instr.[a]];
      else                     tmpOfs = GR[instr.[b]];
   }
   else tmpOfs = GR[instr.[b]] + GR[instr.[a]];
}
else {

   if ( instr.[M] ) {

      if ( lowSignExtend( ofs, 12 ) < 0 ) tmpOfs = GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
      else                                tmpOfs = GR[instr.[b]];
   }
   else tmpOfs = GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
}

len = dataLen( instr.[len] );

if ( instr.[seg] == 0 ) seg = segSelect( tmpOfs );
else                    seg = instr.[seg];

memStore( SR[seg], tmpOfs, GR[instr.[r]], len );

if ( instr.[10] ) {

   if ( instr.[10] ) GR[instr.[b]] <- GR[instr.[b]] + GR[instr.[a]];
   else              GR[instr.[b]] <- GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
}
```

Exceptions

- Illegal instruction trap
- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap
- Overflow trap

Notes

None.

STA

Stores a general register value into memory using an absolute physical address.

Format

```
STA [.M] ofs (b), r
STA [.M] a (b), r
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: STA     ( 0x33 ): r         :0 :M : 0   : 2   : ofs                              : b         :
:-----------------:-------------------------------------------------------------------------------:
: STA     ( 0x33 ): r         :1 :M : 0   : 2   : 0                    : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The store absolute instruction will store the target register into memory using a physical address. The absolute 32-bit address is computed by adding the signed offset or general register "a" to general register "b". The "M" bit indicates base register increment. If set, a negative value in the "ofs" field or negative content "a" will add the offset to the base register before the memory access, otherwise after the memory access. The STWA instructions are privileged instructions.

Operation

```
if ( ! ST.[ PRIV ] ) privilegedOperationTrap( );

 if ( instr.[10] ) {

   if ( instr.[M] ) {

      if ( GR[instr.[a]] < 0 ) tmpOfs = GR[instr.[b]] + GR[instr.[a]];
      else                     tmpOfs = GR[instr.[b]];
   }
   else tmpOfs = GR[instr.[b]] + GR[instr.[a]];
}
else {

   if ( instr.[M] ) {

      if ( lowSignExtend( ofs, 12 ) < 0 ) tmpOfs = GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
      else                                tmpOfs = GR[instr.[b]];
   }
   else tmpOfs = GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
}

memStore( 0, tmpOfs, GR[instr.[r]], 32 );

GR[instr.[b]] <- GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );

if ( instr.[10] ) {

   if ( instr.[10] ) GR[instr.[b]] <- GR[instr.[b]] + GR[instr.[a]];
   else              GR[instr.[b]] <- GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );
}
```

Exceptions

- Privileged operation trap
- Data alignment trap

Notes

None.

STC

Conditionally store a value to memory.

Format

```
STC ofs ( [s,] b),  r
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: STC     ( 0x35 ): r         : 0   : seg : 2   : ofs                               : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The STC conditional instruction will store a value in "r" to the memory location specified by the operand address. The first part of the instruction behaves exactly like the STW instruction where a virtual address is computed. The store is however only performed when the data location has not been written to since the last load reference instruction execution for that address. If the operation is successful a value of zero is returned otherwise a value of one. See the section on operand encoding for the defined operand mode encoding. Only OpMode 3 with word data length is allowed.

Operation

```
if (( lrValid ) && ( lrVal == GR[instr.[r]])) {

   tmpOfs = GR[instr.[b]] + lowSignExtend( instr.[ofs], 12 );

   if ( tmpOfs.[30.31] != 0 ) dataAlignmentTrap( );

   if ( instr.[seg] == 0 ) seg = segSelect( tmpOfs );
   else                    seg = instr.[seg];

   memStore( SR[seg], tmpOfs, GR[instr.[r]], 32 );
   GR[instr.[r]] <- 0;

} else GR[instr.[r]] <- 1;
```

Exceptions

- Illegal instruction tap
- Data TLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

The "check the access part" is highly implementation dependent. One option is to implement this feature in the L1 data cache. Under construction...

SUB

Subtracts the operand from the target register.

Format

```
SUB[.<opt>] r, operand        w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: SUBw    ( 0x12 ): r         :L :O : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The instruction fetches the operand and subtracts it from the general register "r". The "L" bit set specifies an unsigned subtraction. If the "O" bit is set, a numeric overflow will cause an overflow trap. See the section on operand encoding for the defined operand modes. The operations will set the carry/borrow bits in the processor status word.

Operation

```
  switch( opMode ) {

   case 0: tmpA <- GR[instr.[r]]; tmpB <- lowSignExtend( opArg, 18 ); break;

   case 1: tmpA <- GR[instr.[a]]; tmpB <- GR[instr.[b]]; break;

   case 2:
   case 3: {

      seg <- operandAdrSeg( instr );
      ofs <- operandAdrOfs( instr );
      len <- operandBitLen( instr );

      tmpA <- GR[instr.[r]];
      tmpB <- zeroExtend( memLoad( SR[seg], ofs, len ), len );

   } break;
}

res <- tmpA - tmpB;

if ( instr.[O] && overflow ) overflowTrap( );
else {

   GR[instr.[r]] <- res;
   PSW[C/B] <- carry/borrow Bits;
}
```

Exceptions

- Overflow trap
- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

None.

XOR

Performs a bitwise XORing the operand and the target register and stores the result into the target register.

Format

```
XORw [.<opt>] r, operand          w = B|H|W
```

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: XORw    ( 0x16 ): r         :N :0 : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The instruction fetches the data specified by the operand and performs a bitwise XOR of the general register "r" and the operand fetched. The result is stored in general register "r". The "N" bit allows to negate the result making the XOR a XNOR operation. See the section on operand encoding for the defined operand modes.

Operation

```
switch( opMode ) {

   case 0: tmpA <- GR[instr.[r]]; tmpB <- lowSignExtend( opArg, 18 ); break;

   case 1: tmpA <- GR[instr.[a]]; tmpB <- GR[instr.[b]]; break;

   case 2:
   case 3: {

      seg <- operandAdrSeg( instr );
      ofs <- operandAdrOfs( instr );
      len <- operandBitLen( instr );

      tmpA <- GR[instr.[r]];
      tmpB <- zeroExtend( memLoad( seg, ofs, len ), len );

   } break;
}

res <- tmpA ^ tmpB;

if ( instr.[N]) res <- ~ res;
GR[instr.[r]] <- res;
```

Exceptions

- DTLB miss/data page fault
- Data memory access rights trap
- Data memory protection ld trap
- Page reference trap
- Data alignment trap

Notes

None.

Synthetic Instructions

The instruction set allows for a rich set of options on the individual instruction functions. Setting a defined option bit in the instruction adds useful capabilities to an instruction with little additional overhead to the overall data path. The same is true for instruction that use general register zero as an argument. For better readability, an assembler could offer synthetic instructions for more convenient usage of the available instructions. Since the assembler generates these instruction they are called synthetic instructions.

There is also the case where the assembler could simplify the coding process by emitting instruction sequences that implement often used code sequences. For example consider the load an immediate value operation. When the immediate value is larger than what the instruction field can hold, a two instruction sequence is used to load an arbitrary 32-bit value. The assembler could offer a generic "load immediate" synthetic instruction and either use a one instruction or two instruction sequence depending on the numeric value. Another example is the case where the offset to a base register is not large enough to reach the data item. A two instruction sequence using the ADDIL instruction could transparently be emitted by the assembler.

Immediate Operations

VCPU-32 offers an immediate field in the immediate instructions and some computational instructions. Since the fields offer a limited numeric range, a large immediate value needs to be loaded from a sequence of instructions.

Synthetic Instruction	Possible Assembler Syntax	Possible Implementation	Purpose
LDI	LDI GRx, val	LDO GRx, val(GR0)	If val is in the 18-bit range.
LDI	LDI GRx, val	LDIL GRx, L%val; LDO GRx, R%val(GRx)	The 32-bit value is stored with a two instruction sequence.
ADDI	ADDI GRx, val		
ADDI	ADDI GRx, val		
SUBI	SUBI GRx, val		
SUBI	SUBI GRx, val		

Register Operations

Synthetic Instruction	Possible Assembler Syntax	Possible Implementation	Purpose
NOP	NOP	OR GR0, GR0	There are many instructions that can be used for a NOP. The idea is to pick one that does not affect the program state.
CLR	CLR GRn	AND GRn, GR0	Clears a general register.
COPY	COPY GRx, GRy	OR GRx, GRy, GR0	Copies general register GRy to GRx.
MR	MR GRx, GRy	OR GRx, GRy, GR0	This overlaps the MR instruction for moving two general registers.
NOT	NOT GRx, GRy	CMP.NE GRx, GRy	Logical NOT. Tests GRy for a non-zero value and returns 0 or 1.
INC	INC [.< opt >] GRx, val	, ADC, SUB [.< opCt >] GRx, val	Increments GRx by "val". Note that "val" can also be a negative number, which then results in a decrement.
DEC	DEC [.< opt >] GRx, val	SUB [.< opt >] GRx, val	Decrements GRx by "val". Note that "val" can also be a negative number, which then results in an increment.
NEG	NEG GRx	SUB [.] GRx, GR0, GRy	Negates a register by subtracting GRy from zero.

Synthetic Instruction	Possible Assembler Syntax	Possible Implementation	Purpose
COM	COM GRx	OR.N GRx, GRx, R0	OR a zero value with GRx and complements the result stored in GRx.
...	...	...	more to come ...

Shift and Rotate Operations

VCPU-32 does not offer instructions for shift and rotate operations. They can easily be realized with the EXTR and DEP instructions.

Synthetic Instruction	Possible Assembler Syntax	Possible Implementation	Purpose
ASR	ASR GRx, shamt	EXTR.S Rx, Rx, 31 - shamt, 32 - shamt	For the right shift operations, the shift amount is transformed into the bit position of the bit field and the length of the field to shift right. For arithmetic shifts, the result is sign extended.
LSR	LSR GRx, shamt	EXTR Rx, Rx, 31 - shamt, 32 - shamt	For the right shift operations, the shift amount is transformed into the bot position of the bit field and the length of the field to shift right.
LSL	LSR GRx, shamt	DEP.Z Rx, Rx, 31 - shamt, 32 - shamt	
ROL	ROL GRx, GRy, shamt	DSR Rx, Rx, shamt	
ROR	ROR GRx, GRy, shamt	DSR Rx, Rx, 32 - shamt	

System Type Instructions

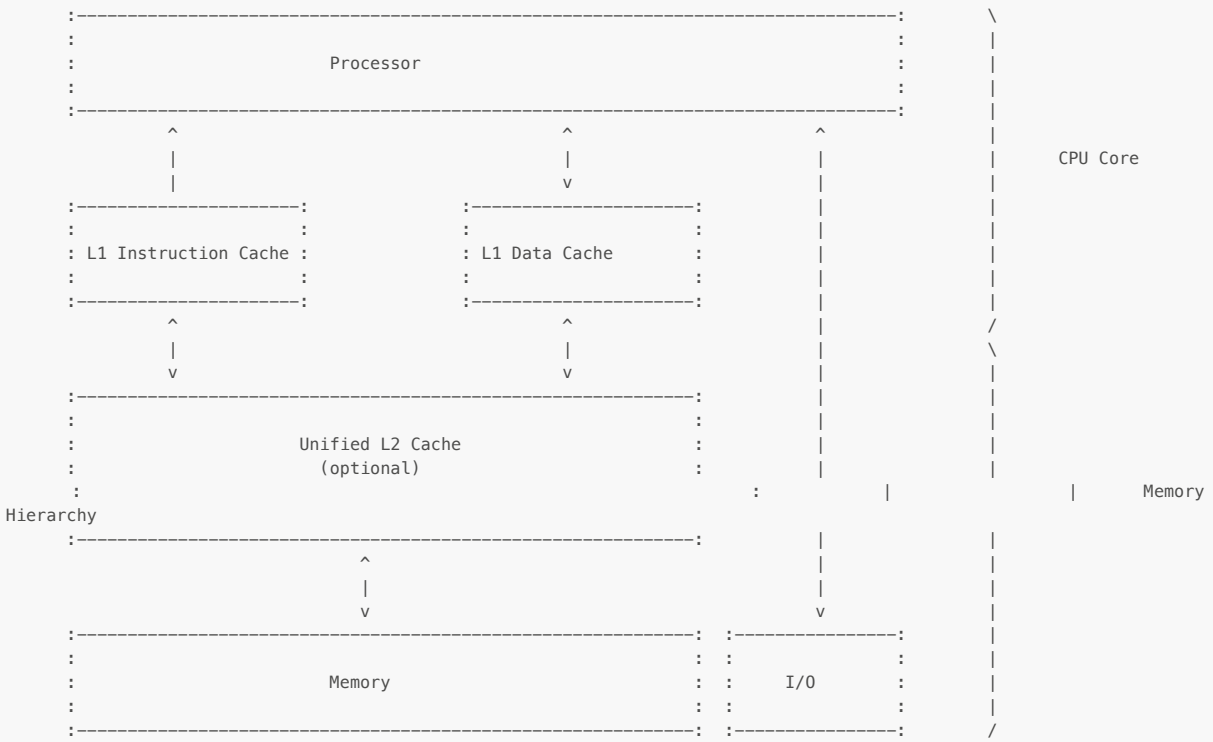
Synthetic Instruction	Possible Assembler Syntax	Possible Implementation	Purpose
PICA	PICA [.M] a([s,] b)	PCA [.M] a([s,] b)	Purge from instruction cache
FDCA	FDCA [.M] a([s,] b)	PCA.TF[M] a([s,] b)	Flush instruction cache
PDCA	PDCA [.M] a([s,] b)	PCA.T[M] a([s,] b)	Purge from data cache
IITLB			Insert in Instruction TLB
PITLB			Purge from Instruction TLB
IDTLB			Insert in Data TLB
PDTLB			Purge from Data TLB

TLB and Cache Models

A key part of the CPU is a cache and a TLB mechanism. The caches bridge the performance gap between a main memory and the CPU processing elements. In modern CPUs, there is even a hierarchy of cache layers. In addition, a virtual memory system needs a way to translate a virtual address to a physical address on each instruction. The translation look-aside buffers are therefore an indispensable component of such systems. Not surprisingly, VCPU-32 has caches and TLBs too.

Instruction and Data Caches

The processor unit, the TLB and caches form the "CPU core". The pipeline design makes a reference to memory during instruction fetch and then optional data access. Since both operations potentially take place for different instructions but in the same cycle, a separate **instruction cache** and **data cache** is a key part of the overall architecture. These two caches are called **L1 caches**. The L1 cache is a virtually indexed and physically tagged cache. The cache lookup and the TLB lookup can therefore be performed in parallel. If the TLB physical address and the cache tag match, there is a cache hit. In an implementation with just L1 caches, both caches will directly interface with the physical memory and IO space. L1 caches typically use a direct mapped indexing methods with one or more sets. They complete a hit within the same cycle.



In addition to the L1 caches, there could be a joint L2 cache to serve both L1 caches. On a simultaneously issued L1 cache request, the instruction cache request has priority. In contrast to the L1 caches, the L2 cache is physically indexed and physically tagged. The L2 cache is also inclusive, which means that a cache line entry in a L1 cache must exist also in the L2 cache. A cache flush operation can thus always assume that there is an entry in the L2 as the target block.

Combined caches

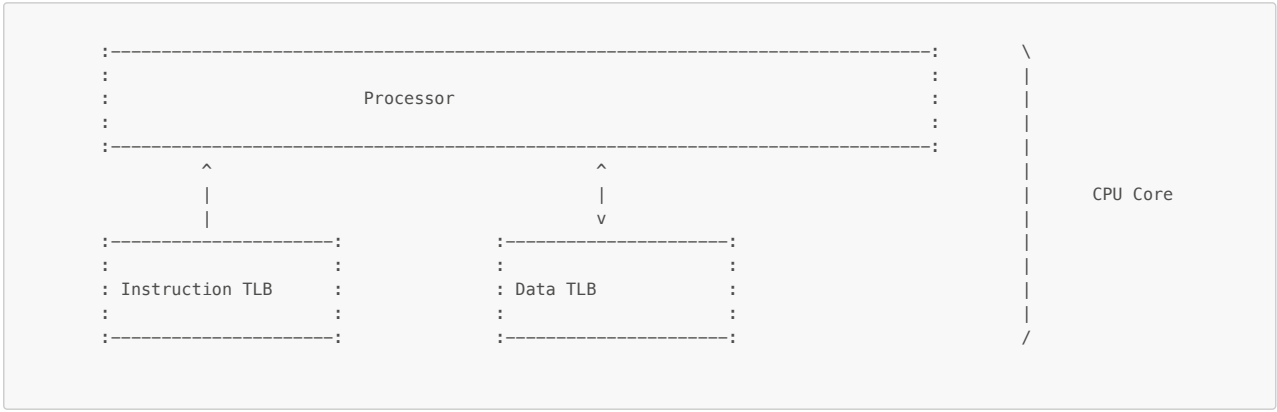
Depending on the CPU pipeline architecture, the L1 caches could also be combined. However, it needs to be ensured that both an instruction and a data cache line with the same index can be hold by the combined cache.

Instructions to manage caches

While cache flushes and deletions are under software control, cache insertions are alway done by hardware. The PCA instruction manages the cache flush and deletion. A cache is typically organized in cache lines, which are the unit of transfer between the layers of the memory hierarchy. The PCA instruction will flush and/or purge the cache line corresponding to the virtual address. A data page can be flushed and then purged or just purged. A code page can only be purged.

Instruction and Data TLBs

Computers with virtual addressing simply cannot work without a form of **translation look-aside buffer** (TLB). For each instruction using a virtual address for instruction fetch and data access a translation to the physical memory address needs to be performed. The TLB is a kind of cache for translations and comparable to the data caches, separate instruction and data TLBs are the common implementation.



TLBs are indexed by a portion of the virtual address. There is the option of a simple direct mapped TLB, set associative TLBs and a fully associative TLBs. Because of the high hardware cost, a fully associative TLB has only few entries versus the mapped models, which typically have a few hundreds of entries.

Combined TLBs

Another implementation option is to combine both TLB units. Both types of translation are kept in one store. Such a TLB should be implemented as two-port TLB because of he simultaneous instruction TLB instruction and data TLB access. Another approach could be to complement a single port joint instruction and data TLB with a small fully associative instruction TLB that holds entries from the unified TLB. An instruction TLB miss has priority over a data TLB miss.

Instructions to manage TLBs

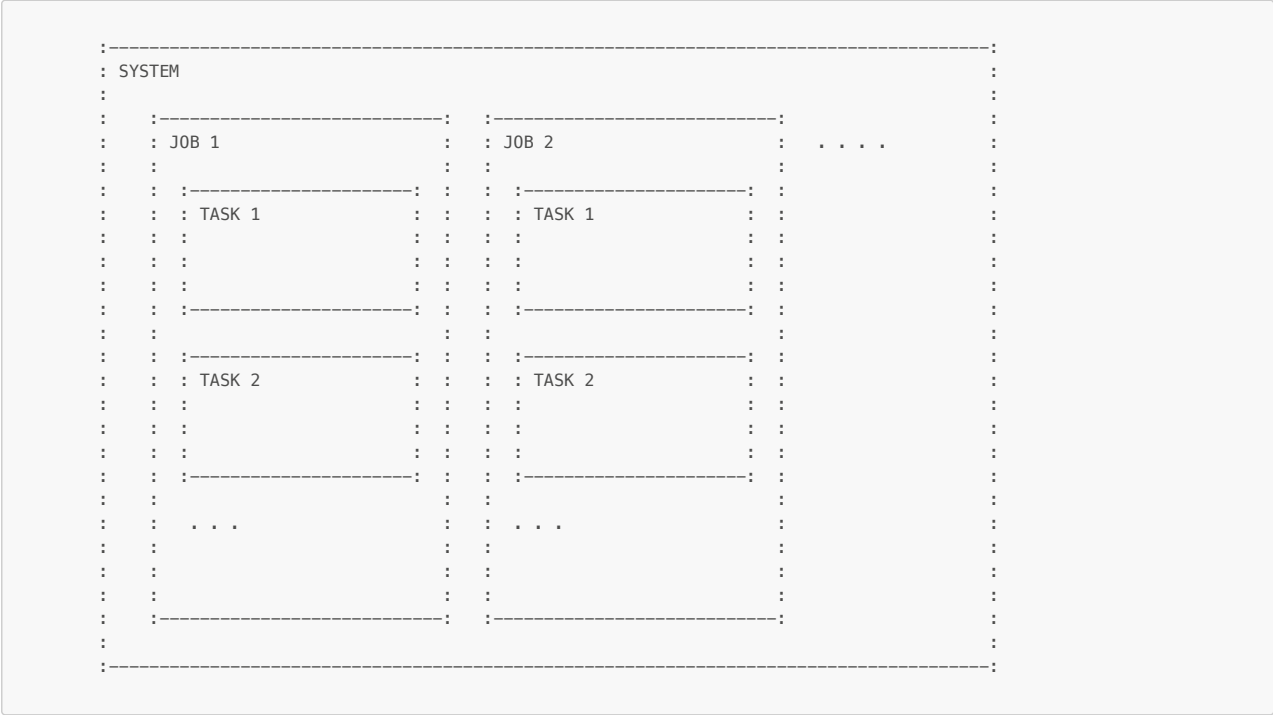
TLBs are explicitly managed by software. The ITLB and PTLB instruction allow for insertion and deletion of TLB entries. The insert TLB instruction will place an entry for the virtual page number along with access rights and the physical page number. The purge TLB instruction removes an entry for the virtual page number.

VCPU-32 Runtime Environment

No CPU architecture with an idea of a runtime environment. Although the instruction set can be used to implement many models of runtime environments, there are common principles that exploit the CPU architecture. In fact, several CPU concepts were selected with a runtime already in mind. This chapter will present the VCPU-32 runtime environment and describe the high level system model, the register usage convention, the execution model and calling convention.

The bigger picture

The following figure depicts a high level overview of a software system for VCPU-32. At the center is the execution thread, which is called a **task**. A task will consist of the current execution object and the task data area. Tasks belong to a **job**. All tasks of a job share a common job data area. At the highest level is the **system**, which contains the global data area for the system.



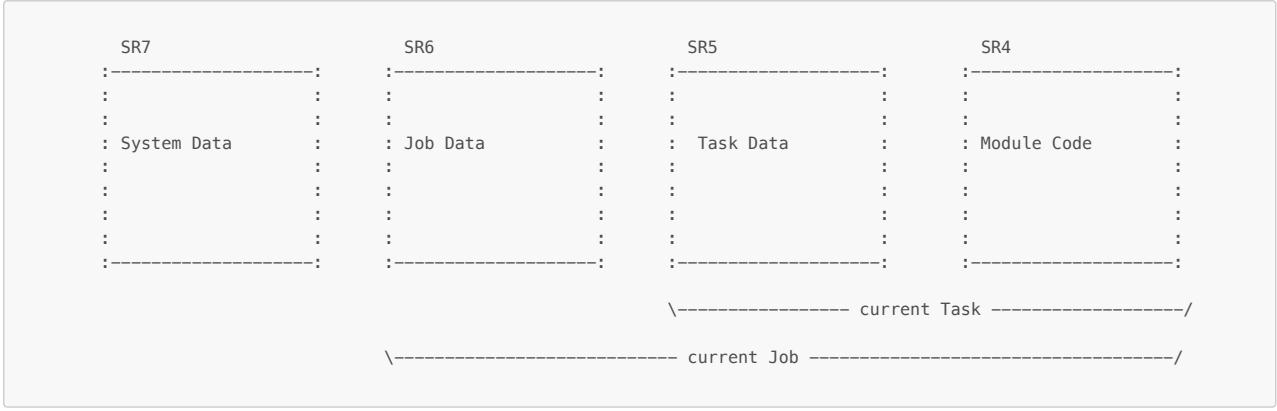
All tasks make calls to system services. At any point in time the CPU is executing a task of a job or a system level functions. If the CPU has more than one core, there are as many tasks active as there are cores in the CPU. A running task accesses the data areas through an address formed by the dedicated segment registers and the offset into the segment. The next section will describe how this high level system model will be mapped to the registers of the CPU.

Register Usage

The CPU features general registers, segment registers and control registers. As described before, all general registers can do computation as well as address computation. Segment registers complement the general register set. A combination of a general register and segment register form a virtual address. To avoid juggling on every access both segment and an index register, the segment register selection field in the respective instructions will either implicitly pick one of the upper for segment registers or explicitly specify on of the segment registers 1 to 3. This scheme allows in most cases to just use the logical address when passing pointers to a function and so on.

Consequently, segment registers 4 to 7 play a special role in the runtime environment. An execution thread, i.e. a **task**, in the runtime environment expects access to three data areas. The outermost area is the **system global area**, which is created at system start and never changed from there on. SR7 is assigned to contain the segment ID of this space. Several executing tasks belonging to the same job are provided with the **job data area**. All threads belonging to the job have access to it via SR6. This register is set every time the execution changes to a task of another job. Finally, in a similar manner, the task local data is pointed to by the SR5 and contains task local data and the stack.

Since the upper two bits of the address offset select the segment register, the maximum size of these areas is 30 bits i.e. one gByte.

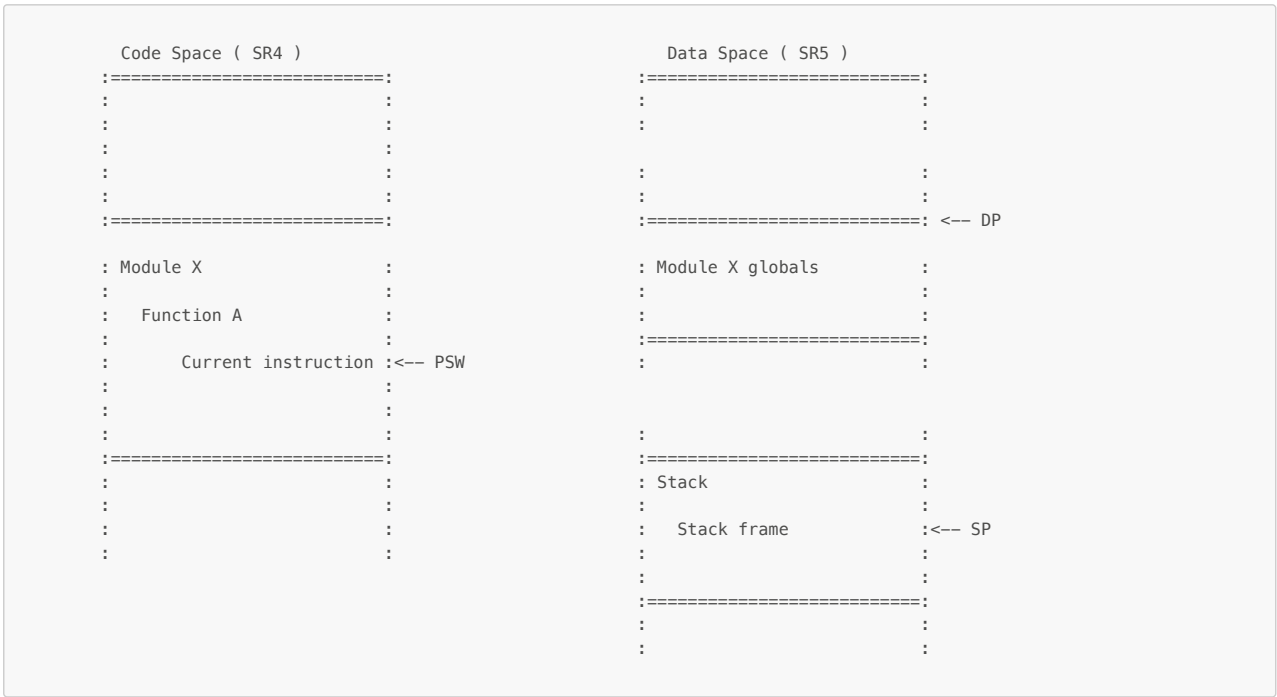


Switching between tasks will set SR5 to the segment containing the task data and stack. In addition, SR6 is set to the job data segment to which the task belongs. Naturally, setting any of these registers is a privileged operation. SR4 will track the current code module so that program literals can be referenced. It will change when crossing a code module boundary. SR7 never changes after system startup.

Task Execution

This section provides a high level overview of a running task execution. A running task will execute instruction pointed to by the virtual address contained in the processor state PSW. Code is organized as a collection of modules, which themselves are a collection of functions. By runtime convention, the code is in the SR4 quadrant and contains instructions and constant data. The constant data can be accessed using the SR4 segment register, which tracks the current code segment.

The SR5 quadrant contains the global data the modules, the task stack and linkage information for inter module calls. The DP register (GR13) point to the global data of the current module and changes on inter-module calls. The SP register (GR15) points to the stack frame of the current function.



For the discussion of the runtime model there are the terms **compilation unit**, **module**, **functions**, **stack** and **stack frame**. At any time the CPU is executing an instruction in a module. A module is a collection of functions and small code snippets, called stubs. A programmer will program a set of functions in a set of modules. A module also contains a global data area, accessible by all functions of that module. A dedicated register will point to the global data area of the current module. Each function will typically also have a stack frame. A stack frame is allocated on function entry and deallocated on function return. The stack frame consist of the local data area and a frame marker for linkage information.

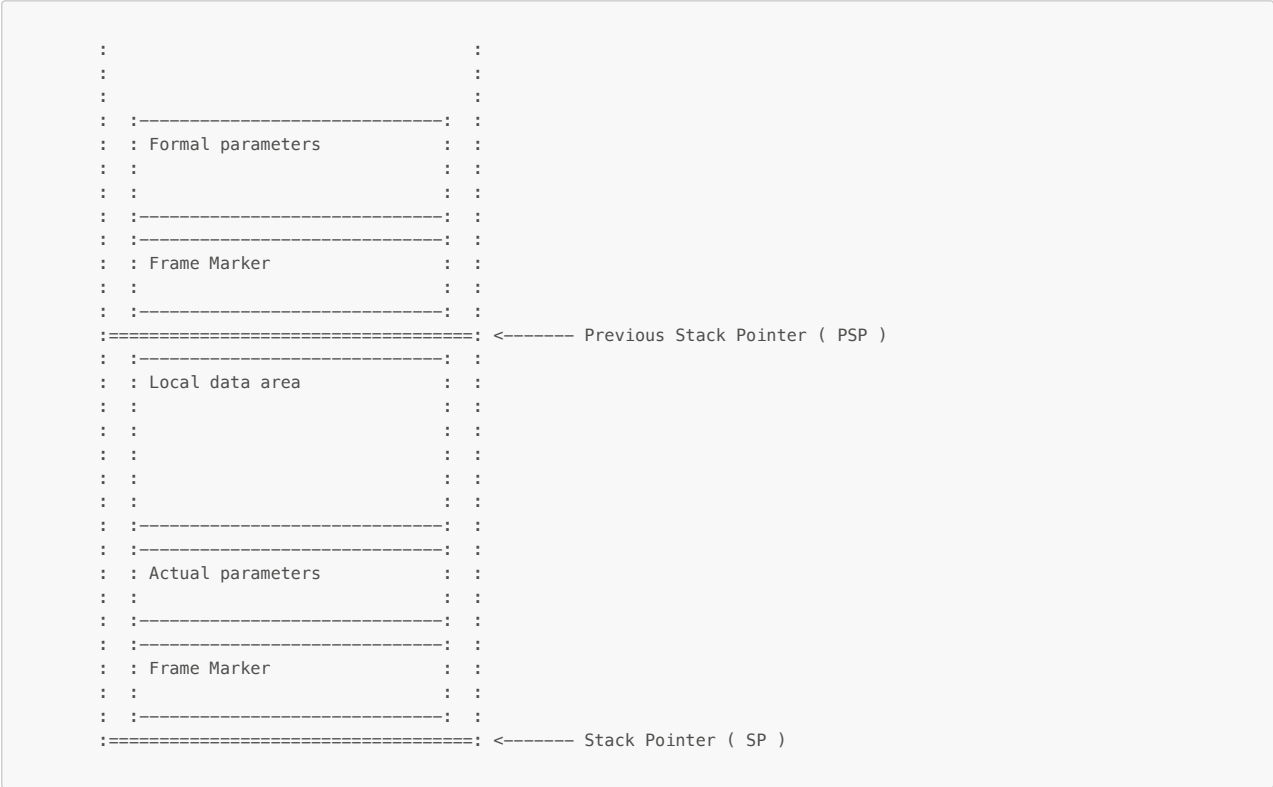
Calling Conventions and Naming

A key portion of the runtime model is the method how function call other functions and pass data to them. The runtime description uses a basic model to describe the calling convention. The calling function is called the **caller**, the target function is called the **callee**.

"Call" transfers control to the callee. **"Entry"** is the point of entry to the callee. **"Exit"** is the point of exit from the caller. **"Return"** is the return point of program control to the caller. In addition, there could be pieces of code that enable more complex calling mechanisms such as library module calls. These pieces are called stubs. Typically, these stubs are added by a linker or loader when preparing the program. The section on external calls will explain stubs in more detail.

Stack and Stack Frames

During program execution functions call other functions. Upon entry, a function allocates a stack frame for its local data, outgoing parameters and a frame maker used for keeping the execution thread data, such as returns links and so on. Generally, there are leaf and non-leaf functions. A leaf function is one that does not make any further calls and perhaps need not to allocate a stack frame.

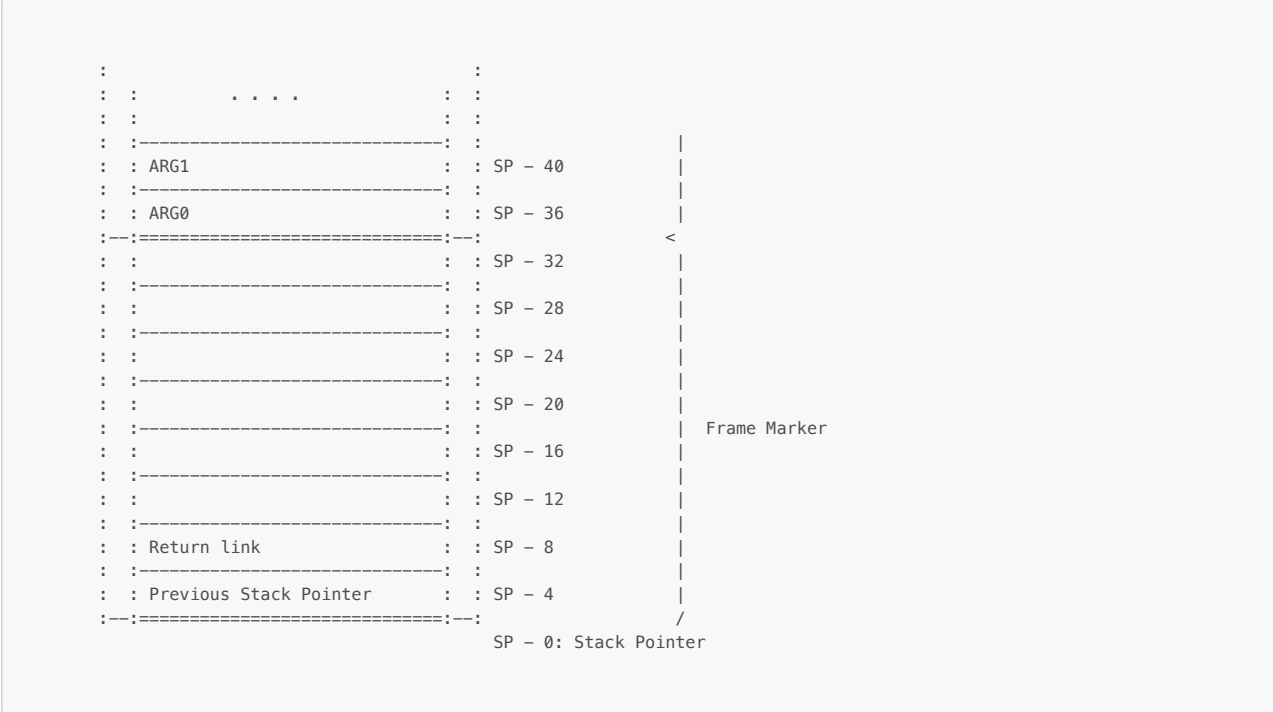


A stack frame is the associated data area for a function. Its size is computed at compile time and will not change. The frame marker and the argument areas are also at known locations. Since the overall frame size is fixed, i.e. computed at compile time, the callee can easily locate the incoming arguments by subtracting its own frame size and the position of the particular argument.

The stack are will grow toward higher memory addresses. The data stack pointer will always points to the location where the next frame will be allocated. Addressing the current frame and the frame marker and parameter area of the previous is SP minus an offset.

Stack Frame Marker

The frame marker is a fixed structure of 8 locations which will contains the data and link information for the call. It used by the caller and the callee to store that kind of data during the calling sequence.



All stack frame locations are addressed relative to the current stack pointer. SP-4 contains the pointer to the previous stack frame. This field is set optional. Since the stack frame size is determined at compile time, the previous stack pointer can be computed by subtracting the frame size from the current stack pointer. SP-8 is the location where the return link can be stored, if necessary. A leaf function may not store the return link in this location.

// ??? **note** under construction. How much space do we need for the calling convention ? // ??? we would need a space to keep the DP value, perhaps a static link for Pascal like languages, and so on.

Register Partitioning

To the programmer general registers and segment registers are the primary registers to work with. Although the registers have with few exceptions no special hardware meaning, the runtime convention assigns to some of the registers a dedicated purpose. The registers are further divided into caller save and callee save registers. A register that needs to be preserved across a procedure call is either saved by the caller or the callee.

```
      General registers                      Segment registers
   :-------------------------------:      :-----------------------------------:
0  :  Zero                          :   0  : Return segment Link               :
   :-------------------------------:      :-----------------------------------:
1  :  Scratch                       :   1  : Caller save, general purpose      :
   :-------------------------------:      :-----------------------------------:
2  :  Callee Save                   :   2  : Caller save, general purpose      :
   :-------------------------------:      :-----------------------------------:
3  :  Callee Save                   :   3  : Callee Save, general purpose      :
   :-------------------------------:      :-----------------------------------:
4  :  Callee Save                   :   4  : Caller Save, general purpose      :
   :-------------------------------:      :-----------------------------------:
5  :  Callee Save                   :   5  : Task segment ( stack )            :
   :-------------------------------:      :-----------------------------------:
6  :  Callee Save                   :   6  : Job segment                       :
   :-------------------------------:      :-----------------------------------:
7  :  Callee Save                   :   7  : System segment                    :
   :-------------------------------:      :-----------------------------------:
8  :  Caller Save, ARG3, RET3       :
   :-------------------------------:
9  :  Caller Save, ARG2, RET2       :
   :-------------------------------:
10 :  Caller Save, ARG1, RET1       :
   :-------------------------------:
11 :  Caller Save, ARG0, RET0       :
   :-------------------------------:
12 :  Caller save                   :
   :-------------------------------:
13 :  Caller save, ( DP )           :
   :-------------------------------:
14 :  Return link ( RL )            :
   :-------------------------------:
15 :  Stack pointer ( SP )          :
   :-------------------------------:
```

Some VCPU-32 instructions have registers as an implicit target. For example, the ADDIL instruction uses GR1 as implicit target. The BE instruction saves the return link segment implicitly in SR0.

The caller can rely on that certain register are preserved across the call. In addition to the callee save registers Gr2 to GR7, the SP value, the RL and the DP value as well as SR4 to SR 7 are expected to be preserved across a call. In addition to the caller save segment registers, the processor state register as well as any registers set by privileged code are NOT preserved across a procedure call.

Parameter passing

The parameter area contains the formal arguments of the caller when making the call and potential return values when the caller returns. The first four arguments are passed in dedicated registers, ARG0 to ARG3. Although these arguments are always passed in registers, the stack frame reserves a memory location for them. A parameter list larger than four words will use the next locations for ARG5, ARG6 and so on. These arguments are always passed in the corresponding frame memory locations. Bytes, Half-Words and Words are passed by value in the argument register or corresponding memory location right-justified, zero-extended. A reference parameter is passed as a logical address which represents the address of the argument. In case of a full virtual address passed, both segment Id and offset occupy an argument slot each.

// ??? open item: function labels...

Upon return, the formal argument locations are used as the potential result return location. For example, a function to return an integer value would return it in the register RET0.

Local Calls

A local calls is defined as a call from a function to another function in the same module. The following figure shows a general flow of control for a local procedure call. In general the sequence consists of loading the arguments into registers or parameter area locations, saving any registers that are in the callers responsibility and branching to the target procedure. The called procedure will save the return link, allocate its stack frame and save any registers to be preserved across the call. Upon return, these registers are restored, the stack frame is deallocated and control transfers back via the return link to the caller procedure.

```
Caller:  ...
         load arguments
         save caller save registers
         branch and link                    -------->  Callee:   save return link
                                                                  allocate stack frame
                                                                  save callee save registers

                                                                  ...

                                                                  restore callee save registers
                                                                  deallocate stack frame
         restore caller save registers      <--------             branch to return link
         ...
```

Depending on the kind of function, not all of these tasks need to be performed. For example, a leaf routine need not save the return link register. A routine with no local data, no usage of callee save registers and being a leaf routine would not need a stack frame at all. There is great flexibility to omit steps not needed in the particular situation.

The following assembly code snippets shows examples of the local calling sequences. The first example will show a function that just adds two incoming arguments and returns the result. The example uses the register alias names. ARG0 and ARG1 are loaded with their values and a local call is made to the procedure. The target address of the called function is computed by the assembler and omitted in the examples to follow.

```
Source code:

void func1 {                                        int func2( int a, b ) {
   func( 500, 300 );                                      return( a + b );
}                                                   }

Assembly:

func1:   LDI ARG0, 500
         LDI ARG1, 300
         B   RL,   func2        ------------->      func2:   ADD RET0, ACRG0, ARG1
         ...                     <-------------              BV (RL)
```

The called function will just use the two incoming arguments, adds them and stores the result to RET0. The BV instruction will use the return address stored by the BL instruction to return. This example is very simple but also the most fundamental calling sequence. The called procedure does not make any further calls, hence there is no need to save a return link. In fact, the called procedure does not even need an own stack frame. Next is a small example where the called procedure will itself also make a call. The called procedure also has two local variables.

```
Source code:

void func1 {                           void func2( int a, b ) {                int func3( int a, b ) {
   func2( 500, 300 );                     int local1, local2;                    return( a + b );
}                                      }                                       }

Assembly:

func1:   LDI ARG0, 500
         LDI ARG1, 300
         B   RL,   func1   -------->   func2:   ST -8(SP), RL
                                                LDO SP, 56(SP)

                                                ...

                                                B RL,  func3     ---------->    func3: ADD RET0, ACRG0, ARG1
                                                ...               <--------             BV (RL)

                                                LDO -56(SP), SP
                                                LD RL, -8(SP)
...                        <--------            BV (RL)
```

The caller "func1" will produce its arguments and stores them in ARG0 and ARG1, as in the example before. If there were more than four arguments, they would have been stored with a ST instruction in their stack frame location. The callee "func2" is this time a procedure that calls another procedure and this needs a stack frame. First, the return link is saved to the frame marker of "Func1". RL would be overwritten by the call to "func3" and hence needs to be

saved. Next the stack frame for "Func2" is allocated. The size is 32bytes for the frame marker, 16 bytes for the 4 arguments ARG0 to ARG1. Note that even when there are fewer than 4 arguments, the space is allocated. Finally there are two local variables. In sum the frame is 56 bytes. The LDO instruction will move the stack accordingly.

The "func2" procedure will do its business and make a call to "func3". The parameters are prepared and a BL instruction branches to "func3". "func3" is a leaf procedure and there is no need for saving RL. If "func3" has local variables, then a frame will be allocated. Also, if "func3" would use a callee save register, there needs to be a spill area to save them before usage. The spill area is a part of the local data area too. "func3" returns with the BV instruction as before.

When "func2" returns, it will deallocate the stack frame by moving SP back to the previous frame, load the saved return link into RL and a BV instructions branches back to the aller "func1". Phew.

Long calls

The branch instruction has a reach of +/- 8Mbytes. There is the situation where in a very large module the branch instruction cannot reach the target function. In this case, the **B** instruction is replaced by two-instruction sequence to implement a "long call".

```
Source code:

void func1 {                                          int func2( int a, b ) {
   func( 500, 300 );                                     return( a + b );
}                                                     }

Assembly:

func1:   LDI  ARG0, 500
         LDI  ARG1, 300
         LDIL R1,   L%func2
         BE   RL,   R%func2 ( SR4, R1 )   ---------->   func2:  ADD RET0, ACRG0, ARG1
         ...                              <----------           BV RL

         BV RL
```

The function address is loaded by loading the left-hand portion with LDIL instruction and performing a BE instruction, adding the right-hand portion of the function address to R1. The segment used is SR4 which by definition tracks the code segment. Note that the function address is a segment relative address. The **BE** instruction uses a segment relative offset stored in R1. This address is however only know when the module is placed in the code segment at its address. A linker is expected to "patch" the LDIL / BE instruction sequence with the final address of the function. Since the **BV** return instruction always works segment relative, no fix is required here.

Module data access

The DP register points to the current module global data area. Data items can be accessed with a LD and ST instruction. However, with indexed addressing mode a fairly small data area is directly reachable. A large global data area is typically addressed by forming the address with a two instruction sequence. The following example shows how to load a data word at DP relative offset "data_ofs" into R5, both short and long versions.

```
LDW   R5, data_ofs( DP )
```

Instructions that offer the operand mode encoding use the indexed mode in a similar way. The ADDIL instruction is used to add the left-hand portion of the offset to DP, storing it in R1. The ADD instruction adds the right-hand offset to R1, forming the address of the operand.

```
ADDIL L%data_ofs( DP )
ADD   R5, R%data_ofs( R1 )
```

In addition, there is there is the register indexed mode, which works with two registers to form the address. The base register could be referring to the global data area, DP and the offset register to the data item. This way, the entire address range is reachable too.

```
LDIL  R4, L%data_ofs
LDO   R4, R%data_ofs
ADD   R5, R4( DP )
```

Module Literal access

Literals are constant values stored in the code segment. They can be reached using SR4 logical addressing. A simple LD with segment SR4 and the offset is sufficient. If a larger offset is required, register indexed addressing can be used. In this mode, one register would be the base of the literal data area and the other would provide the offset.

```
; assume R3 is a pointer to the literal value area, which is a 0x00001000
; the literal item is at offset 100

LDH R4, 100( R3 )          ; load a halfword from literal area base + 100, i.e. 0x00001064
```

```
; assume R3 is a pointer to the literal value area, which is a 0x00001000
; the literal item is at offset 0x8000

LDIL R1, L%0x8000        ; form the contanst 0x8000
LDO  R1, R%0x8000
LDW  R4, R1(R3)          ; load a halfword from literal area base + 0x8000, i.e. 0x00009000
```

Intermodule calls

A compilation or assembly unit can contains more than one module. All code and data addressing is relative to the module itself. Local function calls are instruction counter relative, global data item addressing is DP address relative. and local variables are accessed SP relative.

```
  Code Space ( SR4 )                          Data Space ( SR5 )
:==========================:                :==========================:
:                          :                :                          :
:                          :                :                          :
                                            :                          :


:                          :
:                          :                :                          :
:                          :                :                          :
:==========================:                :==========================: <-- DP
: Module A code            :                : Module A globals         :
:                          :                :                          :
:                          :<-- PSW         :                          :
:                          :                :==========================:
:                          :                : Module B globals         :
:==========================:                :                          :
: Module B code            :                :                          :
:                          :                :                          :
:                          :                :==========================:
:                          :                :                          :
:==========================:                :                          :
:                          :
:                          :
:                          :                :                          :
:                          :                :                          :
                                            :==========================:
                                            : Stack                    :
                                            :                          :
                                            :   Stack frame            :<-- SP
                                            :                          :
                                            :                          :
                                            :==========================:
                                            :                          :
                                            :                          :
```

Invoking a function in a another module requires to reach the target procedure. Furthermore, the DP value for the taget is that of the target module. Assemblers and compilers should not create different calling sequences depending on whether particular call is local to a module or referring to a procedure in another module. The approach is to replace the branch instruction with a branch to a stub, which contains the outgoing code to perform all the operations before calling the target function. Within a module, a calling sequence would be a branch from the calling function to the stub, from the stub to the target function, from the target function back to the stub and then back to the calling function. That is in total four branches. Therefore, a different approach is chosen for intermodule calls within the same compilation unit.

A call to a function in another module is considered a long call. The assembler/compiler creates a long call code sequence. At module compile time, the final target is however not know yet. The long call needs to be updated when the final position of the target module in the code space is known.

Since the target function relies on the DP value of its own module, the DP value may needs to be adjusted. Instead of saving and restoring the callers DP value and setting the new DP value on each call, the callers DP value is placed in a callee save register and thus preserved across a call. The target function needs to load its own DP value when needed. The compilation unit needs for the approach a module table, where for each module the offset of the DP area is kept.

The following code snippet shows an intermodule call within the same compilation unit.

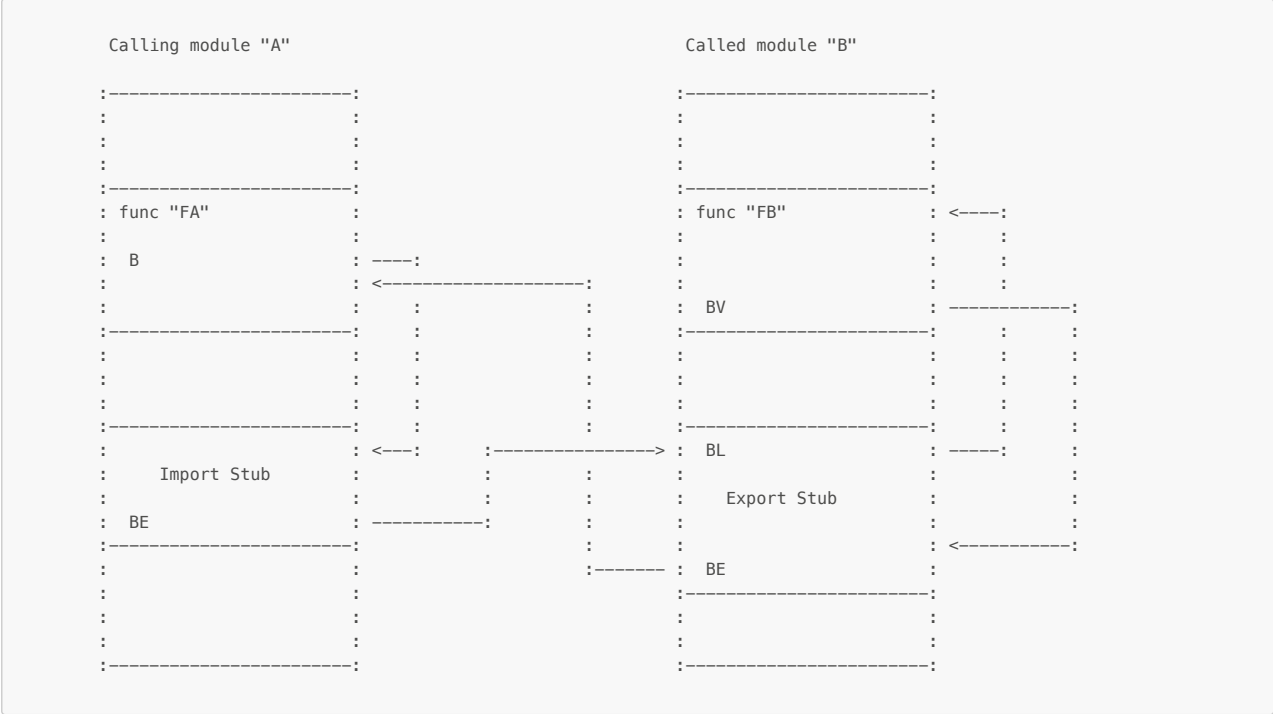
```
Source code Module A:                                   Source Code Module B:

                                                        int i;

void func1 {                                            int func2( int a, b ) {
   func( 500, 300 );                                       return( a + b + i );
}                                                       }

Assembly:

func1:   LDI  ARG0, 500
         LDI  ARG1, 300
         LDIL R1,   L%func2
         BE   RL,   R%func2 ( SR4, R1 )   ---------->   func2:    ...    ;load DP into R11 ... to do ...

                                                                   LDW R3, 0( R11 )
                                                                   ADD R2, ACRG0, ARG1
                                                                   ADD RET0, R2, R3
         ...                              <----------              BV RL

         BV RL
```

A final topic is inter-module calls within the compilation unit when the target has a different privilege level. The architecture implements a privilege level change via the GATE instruction, which resides on a gateway type page.

// ??? **note** what exactly is all needed ? Is it only code or also global data at a new priv level ? Or should we just state that a compilation unit has always the same priv level. If not, it is considered an external call ?

External Calls

An external call is defined as a call that between compilation units that reside in a separate file. For example, the system library called from a program is considered a compilation unit in a separate file. Since functions are in general unaware whether they are called from inside the compilation unit or from another compilation unit, code sequences are necessary to manage the external call in both units. They are called **import stubs** and **export stubs**. The following figure shows the general flow of control for an external call.



The diagram shows function "FA" in module "A" calling function "FB" in module "B". To call "FB", an import stub is added to module "A". All routines in "A" will use this stub when calling "FB" in module "B". After some housekeeping, the import stub makes an external call using the BLE instruction. The target is another stub, the export stub for "B". Every procedure that is export from a module needs to have such an export stub code sequence. The export stub will perform a local branch and link to the target procedure "FB". "FB" will upon return just branch back to the export stub. As a last step, the export stub will perform an external branch to the location after the call instruction in "FA".

There are two data structures used for an external call. The first is the stack frame marker, which contains locations for storing the calling function return link and code segment Id. The other data structure is the linkage table, which contains an entry for each external function.

// ??? what about global data of the target compilation unit ?

...

Import Stub

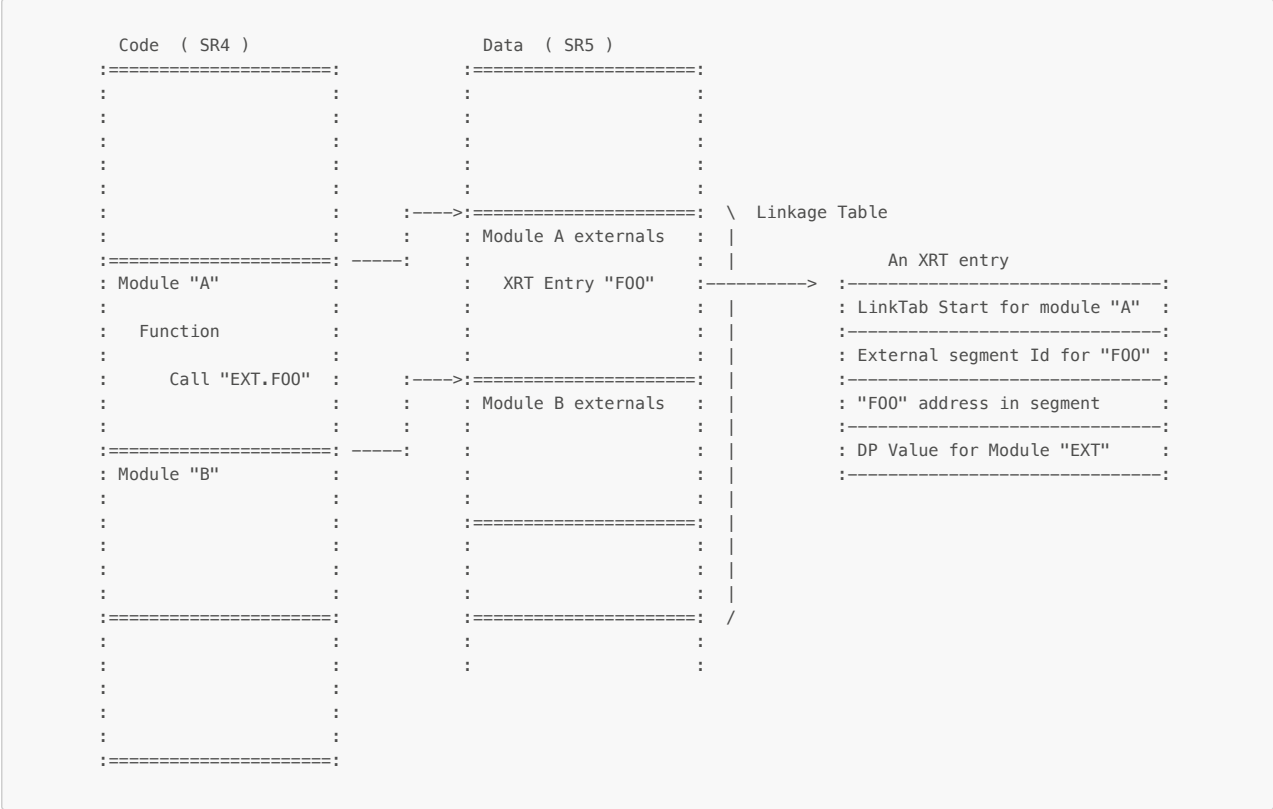
// import stub - uses stack frame locations for housekeeping

Export Stub

// export stub - uses stack frame locations for housekeeping

Linkage Table

The linkage table is a memory structure built by the program load operation. It contains the information necessary to locate the external routine to call. Each loaded module has a set of entries in this table. An entry describes where to find the external routine. There is one table for a running task.



- how to get to the linkage table subsection ? -> DP - 4 location could hold the pointer to the XRT subtable for the module.

Privilege level changes

// privilege changes are also considered as external calls, although perhaps local to a module

// GATE instruction and external calls

Trap handling

External Interrupt handling

Debug Subsystem

// essentially a trap

// support for sigle stepping ?

System Startup sequence

// what needs to be in place before we can load an operating system, or alike ... IPL / ISL ?

// role of PDC

Processor Dependent Code

Part of the I/O memory address range is allocated to processor dependent code.

// entry and exit conditions

// invocation mechanism

// runs privileged always

// groups of PDC services

VCPU-32 Input/Output system

The physical address space

VCPU-32 implements a memory mapped I/O architecture. 1/16 of physical memory address space is dedicated to the I/O system. The I/O Space is further divided into a memory address range for the processor dependent code and the IO modules.

Adress range start	**end**	**Usage**
0x00000000	0xEFFFFFFF	Physical memory
0xF0000000	0xF0FFFFFF	Processor dependent code
0xFF000000	0xFFFFFFFF	IO memory

Concept of an I/O Module

// Bus and IO Module

// hard physical pages

External Interrupts

// to do ...

Instruction Set Summary

This appendix lists all instructions by instruction group.

Computational Instructions Operand Encoding

```
<--- operation --> <-  res  -> <opt> <---------------- operand ---------------------------------->
 0                 6           10    12    14    16    18                24          28       31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: opCode          : r         : opt : 0   : val                                                 :  immediate
:-----------------:-----------------------------------------------------------------------------:
: opCode          : r         : opt : 1   : 0                           : a         : b         :  register
:-----------------:-----------------------------------------------------------------------------:
: opCode          : r         : opt : 2   : dw  : 0                     : a         : b         :  register indexed
:-----------------:-----------------------------------------------------------------------------:
: opCode          : r         : opt : 3   : dw  : ofs                               : b         :  indexed
:-----------------:-----------------------------------------------------------------------------:
```

Computational Instructions using Operand Mode Format

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: ADD     ( 0x10 ): r         :L :O : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: ADC     ( 0x11 ): r         :L :O : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: SUB     ( 0x12 ): r         :L :O : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: SBC     ( 0x13 ): r         :L :O : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: AND     ( 0x14 ): r         :N :C : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: OR      ( 0x15 ): r         :N :C : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: XOR     ( 0x16 ): r         :N :0 : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: CMP     ( 0x17 ): r         :cond : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: CMPU    ( 0x18 ): r         :cond : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
```

Computational Instructions

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LSID    ( 0x04 ): r         : 0                                                   : b         :
:-----------------:-----------------------------------------------------------------------------:
: EXTR    ( 0x05 ): r         :S :A :0            : len          :0 : pos          : b         :
:-----------------:-----------------------------------------------------------------------------:
: DEP     ( 0x06 ): r         :Z :A :I : 0        : len          :0 : pos          : b         :
:-----------------:-----------------------------------------------------------------------------:
: DSR     ( 0x07 ): r         :0 :A : 0           : shamt        : 0   : a         : b         :
:-----------------:-----------------------------------------------------------------------------:
: SHLA    ( 0x08 ): r         :I :L :O : 0                 : sa  : 0   : a         : b         :
:-----------------:-----------------------------------------------------------------------------:
: CMR     ( 0x09 ): r         :cond       : 0                           : a         : b         :
:-----------------:-----------------------------------------------------------------------------:
```

Immediate Instructions

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDIL    ( 0x01 ): r         : val                                                             :
:-----------------:-------------------------------------------------------------------------------:
: ADDIL   ( 0x02 ): r         : val                                                             :
:-----------------:-------------------------------------------------------------------------------:
: LDO     ( 0x03 ): r         : ofs                                                 : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Memory Reference Instruction

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDw     ( 0x30 ): r         :0 :M : seg : dw  : ofs                                 : b         :
:-----------------:-------------------------------------------------------------------------------:
: LDw     ( 0x30 ): r         :1 :M : seg : dw  : 0                       : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: STw     ( 0x31 ): r         :0 :M : seg : dw  : ofs                                 : b         :
:-----------------:-------------------------------------------------------------------------------:
: STw     ( 0x31 ): r         :1 :M : seg : dw  : 0                       : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: LDR     ( 0x34 ): r         : 0   : seg : 2   : ofs                                 : b         :
:-----------------:-------------------------------------------------------------------------------:
: STC     ( 0x35 ): r         : 0   : seg : 2   : ofs                                 : b         :
:-----------------:-------------------------------------------------------------------------------:
: LDA     ( 0x32 ): r         :0 :M : 0   : 2   : ofs                                 : b         :
:-----------------:-------------------------------------------------------------------------------:
: LDA     ( 0x32 ): r         :1 :M : 0   : 2   : 0                       : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: STA     ( 0x33 ): r         :0 :M : 0   : 2   : ofs                                 : b         :
:-----------------:-------------------------------------------------------------------------------:
: STA     ( 0x33 ): r         :1 :M : 0   : 2   : 0                       : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Control Flow Instructions

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: B       ( 0x20 ): r         : ofs                                                             :
:-----------------:-------------------------------------------------------------------------------:
: GATE    ( 0x21 ): r         : ofs                                                             :
:-----------------:-------------------------------------------------------------------------------:
: BR      ( 0x22 ): r         : 0                                                   : b         :
:-----------------:-------------------------------------------------------------------------------:
: BV      ( 0x23 ): r         : 0                                                   : b         :
:-----------------:-------------------------------------------------------------------------------:
: BE      ( 0x24 ): r         : 0                                         : a       : b         :
:-----------------:-------------------------------------------------------------------------------:
: BVE     ( 0x25 ): r         : 0                                         : a       : b         :
:-----------------:-------------------------------------------------------------------------------:
: CBR     ( 0x26 ):cond : ofs                                           : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: CBRU    ( 0x27 ):cond : ofs                                           : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

System Control Instructions

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: MR      ( 0x0A ): r         :D :M : 0                                       : s              :
:-----------------:-------------------------------------------------------------------------:
: MST     ( 0x0B ): r         :mode : 0                                             : b        :
:-----------------:-------------------------------------------------------------------------:
: LDPA    ( 0x39 ): r         : 0   : seg : 0                           : a         : b        :
:-----------------:-------------------------------------------------------------------------:
: PRB     ( 0x3A ): r         :W :I : seg : 0                           : a         : b        :
:-----------------:-------------------------------------------------------------------------:
: ITLB    ( 0x3B ): r         :T : 0                                    : a         : b        :
:-----------------:-------------------------------------------------------------------------:
: PTLB    ( 0x3C ): 0         :T :M : seg : 0                           : a         : b        :
:-----------------:-------------------------------------------------------------------------:
: PCA     ( 0x3D ): 0         :T :M : seg :F : 0                        : a         : b        :
:-----------------:-------------------------------------------------------------------------:
: DIAG    ( 0x3E ): r         : info      : 0                           : a         : b        :
:-----------------:-------------------------------------------------------------------------:
: RFI     ( 0x3F ): 0                                                                         :
:-----------------:-------------------------------------------------------------------------:
: BRK     ( 0x00 ): info1     : 0                   : info2                                   :
:-----------------:-------------------------------------------------------------------------:
```

Instruction Operation Description Functions

Instruction operations are described in a pseudo C style language using assignment and simple control structures to describe the instruction operation. In addition there are functions that perform operations and compute expressions. The following table lists these functions.

Function	Description
cat(x, y)	concatenates the value of x and y.
lowSignExtend(x, len)	performs a sign extension of the value "x". The sign bit is stored in the rightmost position and applied to the extracted field of the remaining left side bits.
signExtend(x, len)	performs a sign extension of the value "x". The "len" parameter specifies the number of bits in the immediate. The sign bit is the leftmost bit.
zeroExtend(x, len)	performs a zero bit extension of the value "x". The "len" parameter specifies the number of bits in the immediate.
segSelect(x)	returns the segment register number based on the leftmost two bits of the argument "x".
ofsSelect(x)	returns the 30-bit offset portion of the argument "x".
operandAdrSeg(instr)	computes the segment Id from the instruction offset computed. (See the operand encoding diagram for modes 2 and 3)
operandAdrOfs(instr)	computes the offset portion from the instruction and mode information. (See the operand encoding diagram for modes 2 and 3). For load and store instructions, the base register is optionally adjusted with the offset value.
operandBitLen(instr)	computes the operand bit length from the instruction and mode information. (See the operand encoding diagram for modes)
rshift(x, amount)	logical right shift of the bits in x by "amount" bits.
memLoad(seg, ofs, len)	loads data from virtual or physical memory. The "seg" parameter is the segment and "ofs" is the offset into the segment. The bitLen is the number of bits to load. If "len" is less than 32, the data is zero sign extended. If virtual to physical translation is disabled, the "seg" is zero and "ofs" is the offset from where to load a word from physical memory.
memStore(seg, ofs, val, len)	store data to virtual or physical memory. The "seg" parameter is the segment and "ofs" the offset into the segment. If virtual to physical translation is disabled, the "seg" is zero and "ofs" is the offset from where to store the word "val" from physical memory. "len" specifies the number of bits to store.
loadPhysAdr(seg, ofs)	returns the physical address for a virtual address. The "seg" parameter is the segment and "ofs" the offset into the segment. If virtual to physical translation is disabled, the "ofs" is the physical address in memory.
allocateInstructionTlbEntry(seg, ofs, entry)	allocate an entry in the instruction TLB and return a pointer to the entry.
purgeInstructionTlbEntry(entry)	purge the instruction TLB entry by simply invalidating the entry.
searchInstructionTlbEntry(seg, ofs, entry)	lookup the instruction TLB entry and if found return a pointer to the entry.
allocateDataTlbEntry(seg, ofs, entry)	allocate an entry in the data TLB and return a pointer to the entry.
purgeDataTlbEntry(entry)	purge the dataTLB entry by simply invalidating it.
searchDataTlbEntry(seg, ofs, entry)	lookup the data TLB entry and if found return a pointer to the entry.

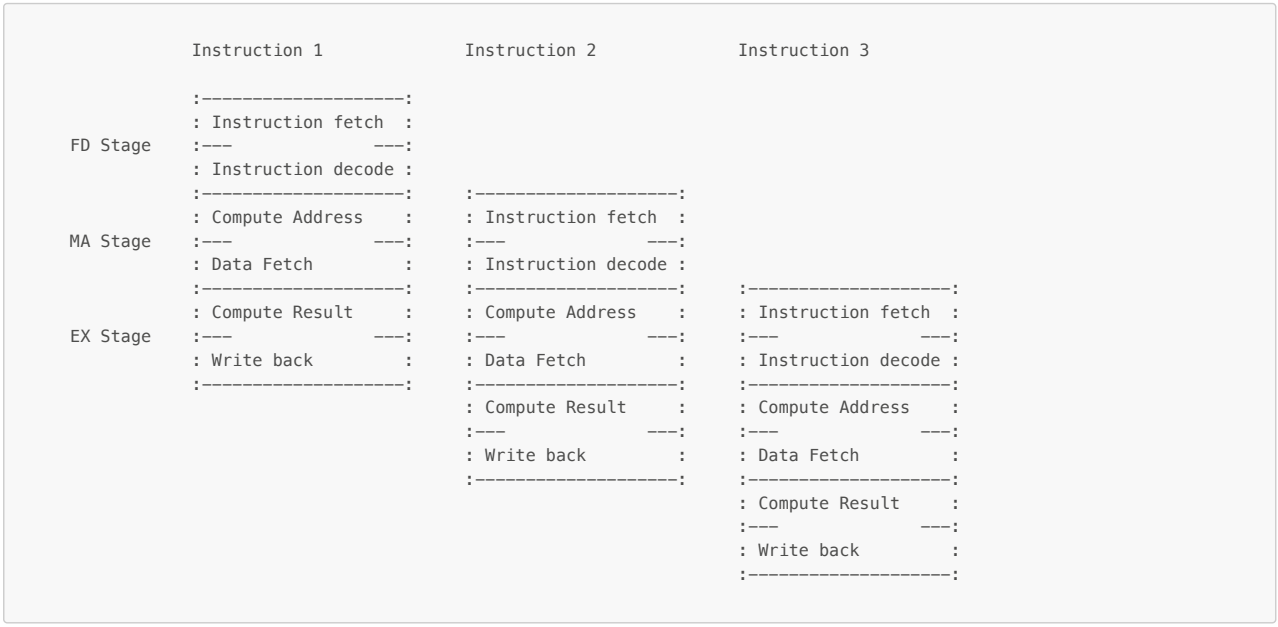
Function	Description
readAccessAllowed(tlbEntry, privLevel)	checks the TLB entry for the requested access mode and privilege level.
writeAccessAllowed(tlbEntry, privLevel)	checks the TLB entry for the requested access mode and privilege level.
purgeInstructionCache(seg, ofs)	remove the instruction cache line that contains the virtual address by simply invalidating it.
flushDataCache(seg, ofs)	write the data cache line that contains the virtual address back to memory and purge the entry.
purgeDataCache(seg, ofs)	remove the data cache line that contains the virtual address by simply invalidating it.

A three-stage pipelined CPU model

The VCPU-32 instruction set and runtime architecture has been designed with a pipeline CPU in mind. In general, pipeline operations such as stalling or flushing a CPU pipeline are in terms of performance costly and should be avoided where possible. Also, accessing data memory twice or any indirection level of data access would also affect the pipeline performance in a negative way. This chapter presents a simple pipeline reference model of a VCPU32 implementation for discussing architecture and instruction design rationales.

The VCPU-32 pipelined reference implementation also can be found in the simulator, which uses a three stage pipeline model. The stages are the **instruction fetch and decode stage**, the **memory access** stage and the **execute** stage. This section gives a brief overview on the pipelining considerations using the three-stage model. The architecture does not demand that particular model. It is just the first implementation of VCPU-32. The typical challenges such as structural hazards and data hazards will be identified and discussed.

- **Instruction fetch and decode**. The first stage will fetch the instruction word from memory and decode it. There are two parts. The first part of the stage will use the instruction address and attempt to fetch the instruction word from the instruction cache. At the same time the translation look-aside buffer will be checked whether the virtual to physical translation is available and if so whether the access rights match. The second part of the stage will decode the instruction and also read the general registers from the register file.
- **Memory access**. The memory access stage will take the instruction decoded in the previous stage and compute the address for the memory data access. This also the stage where a segment or control register are accessed. In addition, unconditional branches are handled at this stage. Memory data item are read or stored depending on the instruction. Due to the nature of a register/memory architecture, the memory access has to be performed before the execute stage. This also implies that there needs to be an address arithmetic unit at this state. The classical 5-stage RISC pipeline with memory access past the execute stage uses the ALU for this purpose.
- **Execute**. The Execute Stage will primarily do the computational work using the values passed from the MA stage. The computational result will be written back to the registers on the next clock cycle. Within the execute stage, there is the computational half followed by the result committing half. In addition, instructions that need to commit two results push one result to the first half of the instruction fetch and decode stage. Currently, the base register modification option of the load instructions will commit the loaded value in the EXECUTE stage but push the base register update to the instruction fetch half of the next instruction.



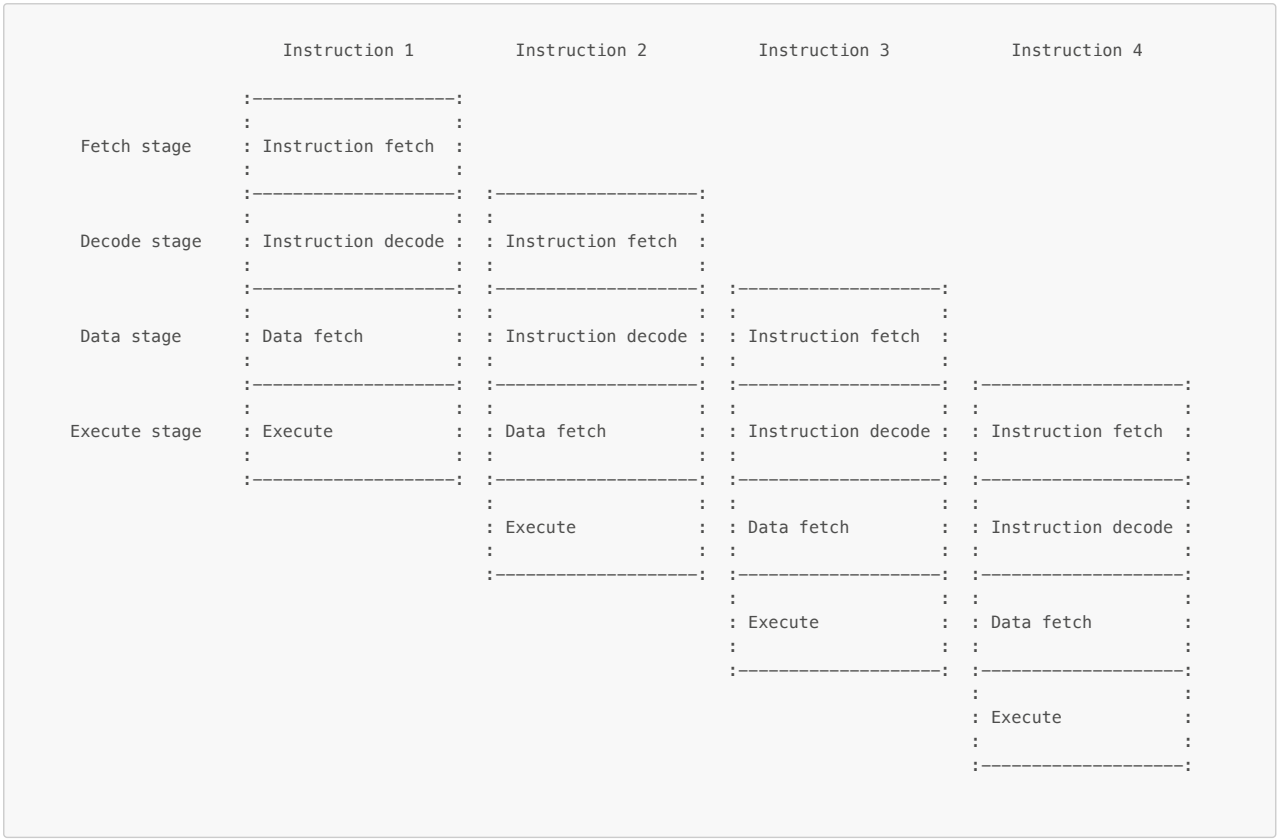
Note that the figure above is certainly one of many ways to implement a pipeline. The three major stages are also further divided internally. For example, the fetch and decode stage could consist of two sub stages. Likewise, the memory access stages is divided into an address calculation sub-stage and the actual data access. Dividing into three stages simplifies the bypass logic as there are only two spots to insert any register overriding. This is especially important for the memory access stage, which uses the register content to build addresses. Two separate stages, i.e. address computation and memory access, would require options to redo the address arithmetic when detecting a register interlock at the memory access stage.

Some instructions, such as store instruction, require to read three data items from the register file and optional also write back two items. Instead of designing a CPU with a three port read and two port write register file, the pipeline shown above is structured ion a way that the access to the register file is done on the first half of the fetch/decode stage and the second half of the memory access stage. The read could thus be split into fetching two registers in the decode substage and when needed a third register during the compute access substage. Likewise, the write back of the execute stage will write back the computation result on the second half of the execute stage, and the address increment write back on the first half of the instruction fetch stage of the next instruction.

In a similar way, the TLB and Caches are spread over the pipeline. The instruction TLB and Cache access takes place on the first half of the fetch/decode stage, the data access on the second half of the memory access stage. In other words the access for instructions and data do not overlap and could be served by a unified TLB and cache.

A four stage pipeline CPU model

The three stage pipeline model, while simple and somewhat elegant, will have its limits in clock cycle time. It is also a scalar pipeline model. Nevertheless, it can be implemented and for the simulator valuable insights can be obtained. Any higher performancde implementation would however lean towards more stages and superscalar techniques. The following sketch is a four stage pipeline.



The **fetch** stage will fetch the next instruction from memory. In contrast to the 3-stage model, there is no way to determine the next instruction address for simple branches already in the fetch stage. In order to improve the branch address prediction, techniques such as a **branch target buffer** and **branch prediction hardware** become essential. If they are not present, any unconditional branch would in one cycle penalty, since the next address is only known in the decode stage at the earliest. Yet the fetch stage needs to provide the next address for the next instruction fetch. A branch target buffer will store instruction address and branch target address for branch type instructions. If the entry matches, all is fine. Otherwise the address can only be compzted in the decode stage and there will be a one cycle penalty.

As pipelines grow larger and superscalar, branch prediction becomes imminent. Any misprediction will cost several cycles and the flushing of several instructions with superscalar designs. The 3-stage pipeline model just used a static prediction scheme and due to the design of fetch and decode in one stage, there was no real need for target address buffering and predictions. The 4-stage pipeline model will introduce these enhancements.

The **decode** stage will decode the instruction and fetch the required data from the register file. Due to the nature of address translation in VCPU-32 the general register file as well as the segment register file need to be accessed in serial order. First, the general register will provide the address offsets and accessing the segment register file using the upper two bits of the offset will prvide the segment part of the virtual address.

The **data fetch** stage will access memory for reading or storing data. Like the fetch stage, it will primarily just access memory. Since both instruction fetch and data fetch or write will access memory overlapping for instructions in flight, seprate L1 caches are required. It is envisoned that both L1 caches connect to a unified L2 cache.

The **execute** stage will perform the computation tasks as before. It also manages execption handing. VCPU-32 provides a precise exception model. A misprediction of a condiztional branch is also resoilved at the execute stage.

In summary, a 4-stage pipeline model will result in a shorter cycle time at the expense of more hardware. Especially the branch target buffer and branch prediction are a significant addition.

Notes

Nullification

A concept found in RISC architectures for reducing pipeline unfrieNdly branches is **nullification**. The idea is to "nullify" an instruction on the result of the previous instruction. For example, a test instruction sets the nullify bit in the status register to indicate that the follow-on instruction will have no effect on the proecessor state but rather treated as a NOP. For experimenting with the concept, the architecture needs a few changes:

1.) the operand modes will feature for mode 1 a 4-bit field to store the nullifictaion options. There are at least two options: "never" and "always". A value of zero in the field is interpreted as "never".

```
        <-- operation --> <-  res  -> <opt> <----------------- operand ---------------------------------->
        0                 6           10    12    14    16    18                24          28       31
       :--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
       : opCode          : r         : opt : 0   : val                                                 :  immediate
       :-----------------:-------------------------------------------------------------------------------:
       : opCode          : r         : opt : 1   : 0   : n-cond    : 0         : a         : b         :  register
       :-----------------:-------------------------------------------------------------------------------:
        : opCode          : r          : opt : 2   : dw  : n-cond    : 0          : a          : b          :  register
indexed
       :-----------------:-------------------------------------------------------------------------------:
       : opCode          : r         : opt : 3   : dw  : ofs                               : b         :  indexed
       :-----------------:-------------------------------------------------------------------------------:
```

Mode 0 and 3 do not have room for the nullify condition field.

2.) The procerrs state needs to have a bit "N".

```
        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
       :--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
       :M :X :C :0 :CB:N : 0               :0 :D :P :E : PSW.[seg]                                     :  PSW-0
       :-------------------------------------------------------------------------------------------------:
       : PSW.[ofs]                                                                          : 0   :  PSW-1
       :-------------------------------------------------------------------------------------------------:
```

The N bit is set when the nullification is evaluated to true. An instruction checks the N bit before committing its results.

3.) Candidates for a nullification condition field.

All computational instructions with the aforementioned operand field are candidates for the nullification option. In addition, the EXTR, DEP, DSR, SHLA and CMP have room for the nullification field.

4.) Removal of the CMR instrcution.

The CMR instruction could easily be replaced by a CMP instruction that conditionally nullifies the follon on instruction.

```
    Example:

    C-code:     if ( a < b ) c = d

    Assumption: ( a -> R2, b -> R6,c -> R1, d -> R4 )

     CMP.LT R1, R2, R6   ; compare R2 < R6 and store a zero or one in R1
     ADD R1, R4, R0      ; move R4 to R1
```

5.) The assembler needs to come up with a way to specifiy nullification in the instruction syntax.

The case for register indexed mode

The current implementation provides a base "register plus offset" and a "base register plus register" addressing mode. Modern CPUs such as the RISC-V family instruction set do not offer a register plus register mode. The key argument is hardware complexity and the fact that the register indexed mode can easily be implemented with a two

instruction sequence, where the first instructons builds the address offset and the next instrcution just uses the result as base register.

To be investigated. Address adjustments as part of an register indexed access is a powerful feature when looping through arrays and well worth the additional hardware and complexity.

Instruction bundling

Supercalar prcessors attempt to execute more than one instruction in one cycle. In a superscalar design the hardware detects teh potential hazards of the instrcutions in flight. When the instructions can furthermore execute in an out of order model, management of the dependencies becomes even more complex and require a lot of hardware estate.

VLIW architectures group instructions in a bundle fetched by the instruction fetch stage, complemented with a template field that will tell the hardware about the type of instructions in the bundle. It is the responsibilty of the compiler to ensure that the instructions in a bundle will not conflict. Although an assembler could also work with instruction bundles, considerting all deoendencies and potential conflicts are a hard an cumbersome task for a human assembler programmer.

References

None so far.