VCPU-32 System Architecture and Instruction Set Reference

Helmut Fieres Version B.00.04 April, 2024

        - ITLB
        - PTLB
        - PCA
        - DIAG
        - RFI
        - BRK
    - Synthetic Instructions
    - This appendix lists all instructions by instruction group.
        -
        -
        -
        -
        -
        -
        -
        -
    - 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.
        - The processor unit and the L1 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**.
        - 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.
        - 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 ca be flushed and then purged or just purged. A code page can only be purged.
        - 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.
        - Another implementation could 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 and data access. Another approach could be to complement a single port joint 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.
        - TLBs are explicitly managed by software. The ITLB and PTLB instruction allow for insertion and deletion of TLB entries. The insert into TLB instructions perform the insertion in two parts. The first instruction puts the address related information into a TLB entry but marks the entry not valid yet. The second insert into TLB instruction will enter access right related information and marks the entry valid.
    - No CPU architecture with an idea of a runtime environment. Although the instruction set is generic enough 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 first present a high level system model, the register convention, execution model, calling convention and overall runtime model.
        - 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 the 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.
        - 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 index register and a segment register form a virtual address. To avoid juggling on every access both a segment and an index register, the segment register selection field in the respective instructions will either implicitly pick one of the upper four segment registers or specify on of the segment registers 1 to 3. This scheme allows to use in most cases just the offset portion of the virtual address when passing pointers to function and so on. The segment register is implicitly encoded in the upper two bits.
        - This section provides a high level overview of a running task execution. A running task will execute instruction pointed to by the IA-SEG.IA-OFS virtual address. 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 module.
        - The runtime description uses a basic model to describe the calling convention. The calling procedure is called the **caller**, the target procedure is called the **callee**.
        - During program execution procedures call other procedures. Upon entry, a procedure allocates a stack frame for 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 procedures. A leaf procedure is one that does not make any further calls and perhaps need not to allocate a stack frame. 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.T
    - 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 to store that kind of data during the calling sequence.
    - To the programmer general registers and segment registers are the primary registers to work with. Although the registers have 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.
    - 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. In case of a full virtual address both segment Id and offset occupy an argument slot each.
    - 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.
    - 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. Procedure calls are thus always local calls. Likewise a called function will return via a local branch. When a call is crossing a module boundary, small pieces of code need to be added which perform the additional work. These code sequences are called **stubs**. There are import stubs that are added for a procedure that makes an external call and export stubs for procedure that are a target for an external call.
    - // privilege changes are also considered as external calls, although perhaps local to a module
    -
    - // essentially a trap
    - // what needs to be in place before we can load an operating system, or alike ... IPL / ISL ?
- Part of the I/O memory address range is allocated to processor dependent code.
- // ??? note what part do we architect ?
    - VCPU-32 implements a memory mapped I/O architecture. 1/16 of physical memory address space is dedicated to the I/O system. The IO Space is further divided into a memory address range for the processor dependent code and the IO modules.
    - // Bus and IO Module
    -
    - The VCPU-32 instruction set and runtime architecture has been designed to take in consideration the effects of stalling and flushing a CPU pipeline. In general, such operations are in terms of performance costly and should be avoided. Also, access data memory twice or any indirection level of data access will also affect the pipeline performance i a negative way. This chapter presents a simple pipeline reference model for a VCPU-32 implementation.
- None so far.
- None so far.

Introduction

> "A vintage 32-bit register-memory model CPU ? You got to be kidding me. Shouldn't we design a modern superscalar, multi-core RISC based 32-bit machine? Or even better 64-bit?. After all, this is the 21st century and not the eighties. A lot happened since the nineties."
>
> "Well why not. Think vintage."
>
> "Seriously?"
>
> "OK, seriously. Designers of the eighties CPUs almost all used a micro-coded approach with hundreds of instructions. Also, 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 nineties shifted to RISC based designs with large sets of registers, fixed word instruction length, and instructions that are in general pipeline friendly. What if these principles had found their way earlier into these designs? What if a large virtual address space, a fixed instruction length and simple pipeline friendly instructions had found their way into these designs ?
>
> A 32-bit vintage CPU will give us a good set of design challenges to look into and opportunities to include modern RISC features as well as learning about instruction sets and pipeline design as any other CPU would do. Although not a modern design, it will still be a useful CPU. Let's see where this leads us. Most importantly it is an undertaking that one person can truly understand. In todays systems this became virtually impossible. And come on, it is simply fun to build something like a CPU on your own."
>
> "OK, so what do you have in mind ?"

Welcome to VCPU-32. VCPU-32 is a simple 32-bit CPU with a register-memory model and a 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. Many of the key architecture features come from there. However, other processors, the Data General MV8000, the DEC Alpha, and the IBM PowerPc, were also influential.

The original design goal that started this work was to truly understand the design process and implementation tradeoff 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 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 in the 80s were typically micro-coded complex instruction machine. In contrast, VCPU-32 instructions will be hard coded and should be in general "pipeline friendly" and avoid 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, 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 CPUs of the 80s and 90s, 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 increase the overall data path length slightly.

Modern RISC CPUs are 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 rather small number of general registers leading to a model where one operand is fetched from memory. 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 into segments. Segment and offset into the segment form a virtual address. In addition, a short form of a virtual address, called a logical address, will select a segment register from the upper logical address bits 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 architecture, instruction set and runtime for VCPU-32. It is organized into several chapters. The first chapter 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, 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. Finally, the remainder of the chapters summarize the instructions defined and also offer an instruction and runtime commentary to illustrate key points on the design choices taken.

Architecture Overview

This chapter presents an overview on the 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 where the data is obtained. 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
```

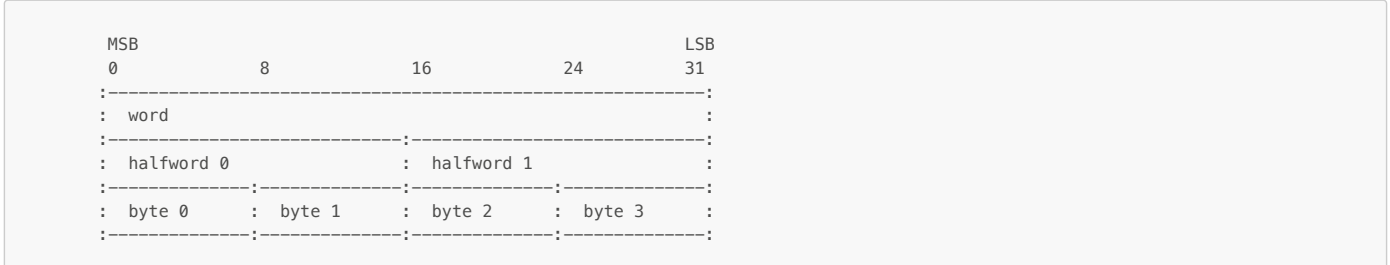
In contrast to similar historical register-memory designs, there is no operation which does a read and write operation to memory in the same instruction. For example, there is no "increment memory" instruction, since this would require two memory operations and is not pipeline friendly. Memory access 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. Computational operations and memory reference operations use a common addressing model, also called **operand mode**. In addition to the memory register mode, one operand mode supports also a 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.

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.

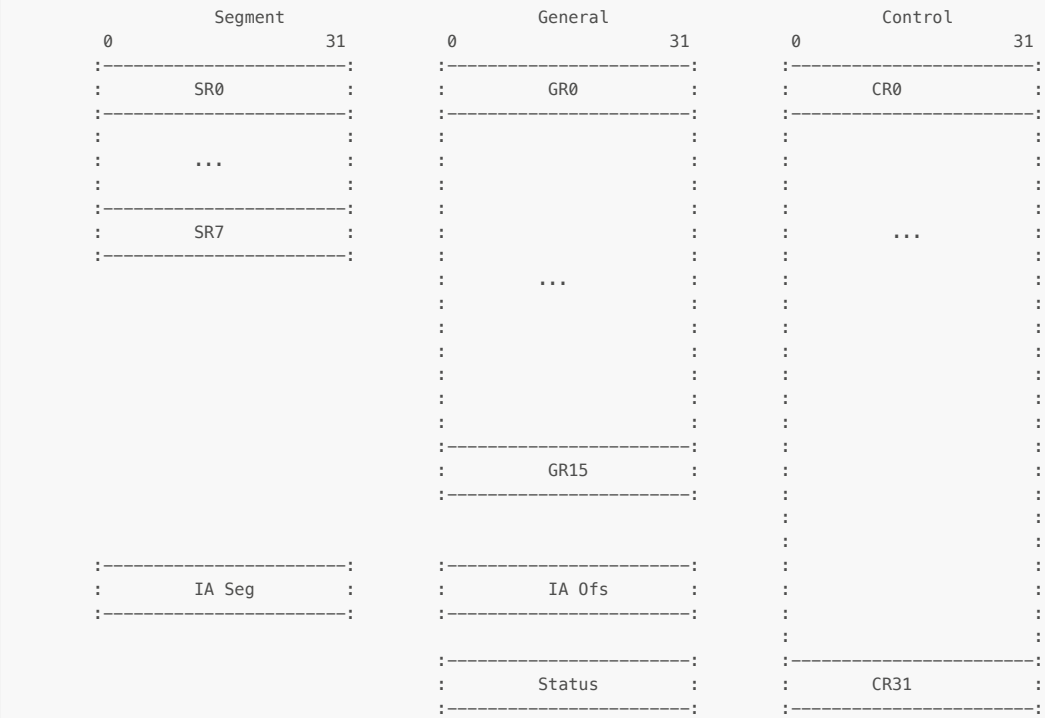
Data Types

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 address are however expressed as bytes addresses.



General Register Model

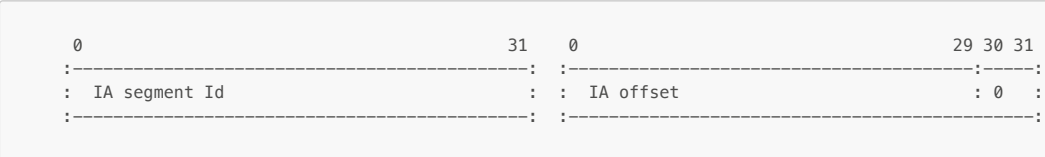
VCPU-32 features a set of registers. They are grouped in general registers, segment registers and control registers. There are sixteen general registers, labeled GR0 to GR15, and eight segment registers, labeled SR0 to SR7. All general registers can do arithmetic and logical operations. The eight segment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



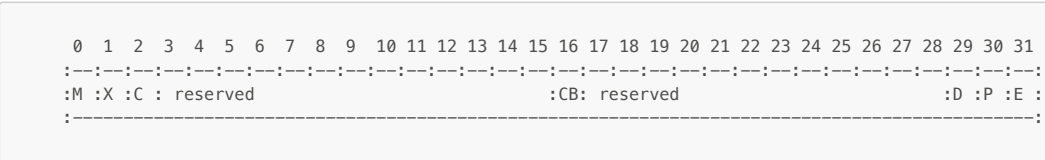
Some general registers have a dedicated use. Register zero will always return a zero value when read and a write operation will have no effect. Although there are only 16 general registers in the CPU, implementing such a register greatly simplifies the instruction design. General register one is a scratch register and also serves as an implicit target for some instruction. Segment register zero severs as n implicit target for some of the branching instructions. Other general register 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 three registers to hold the processor state. The **instruction address register** holds the address of the current instruction being executed. The instruction address is formed by the instruction address segment, IA-SEG, and the instruction address offset, IA-OFS. The instruction address is the virtual or absolute memory location of the current instruction. The lower two bits are zero, as instruction words are word aligned in memory.



The **status register** holds the processor state information, such as the carry bit or current execution privilege. The status register is labelled ST.



Bits 12 .. 17 are reserved for carry bits. A carry bit, bit 12, is generated for the ADD and SUB instruction. Bits 18 .. 23 represent the bit set that can be modified by the privileged MST instruction.

Flag	Name	Purpose
M	Machine Check	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 and SUB 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.

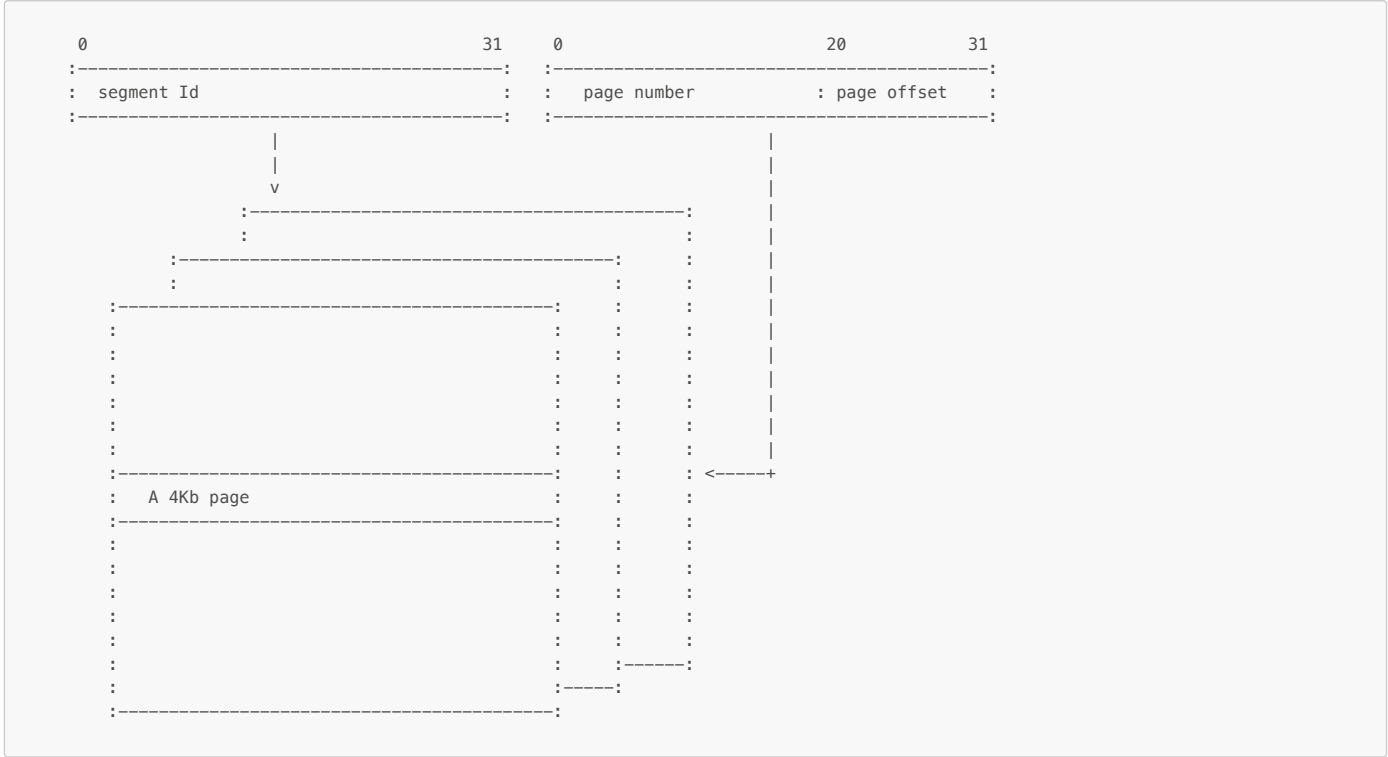
Control Registers

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 return link address, 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.
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.
8 - 11	**PID-n**	Protection ID register 0 to 3. The protection ID register hold the 15-bit protection ID and the write disable bit. When protection ID check is enabled, each virtual page accessed is checked for matching one of the protection ID. The write operation is additionally checked to match the write disable bit. There are four protection ID registers.
12 - 15	reserved	reserved for future use.
16	**I-BASE-ADR**	Interrupt and trap vector table base address. The absolute 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-STAT**	When an interrupt or trap is encountered, this control register holds the current status word.
18	**I-IA-SEG**	When an interrupt or trap is encountered, control register holds the current instruction address segment.
19	**I-IA-OFS**	When an interrupt or trap is encountered, control register holds the current instruction address offset.
20 - 22	**I-PARM-n**	Interrupts and pass along further information through these control registers.
23	**I-EIM**	External interrupt mask.
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 2^32 segments, each of which holds up to 2^32 words in size. Segments are further subdivided into pages with a page size of 4K Words. The concatenation of segment ID and offset form a **virtual address**.

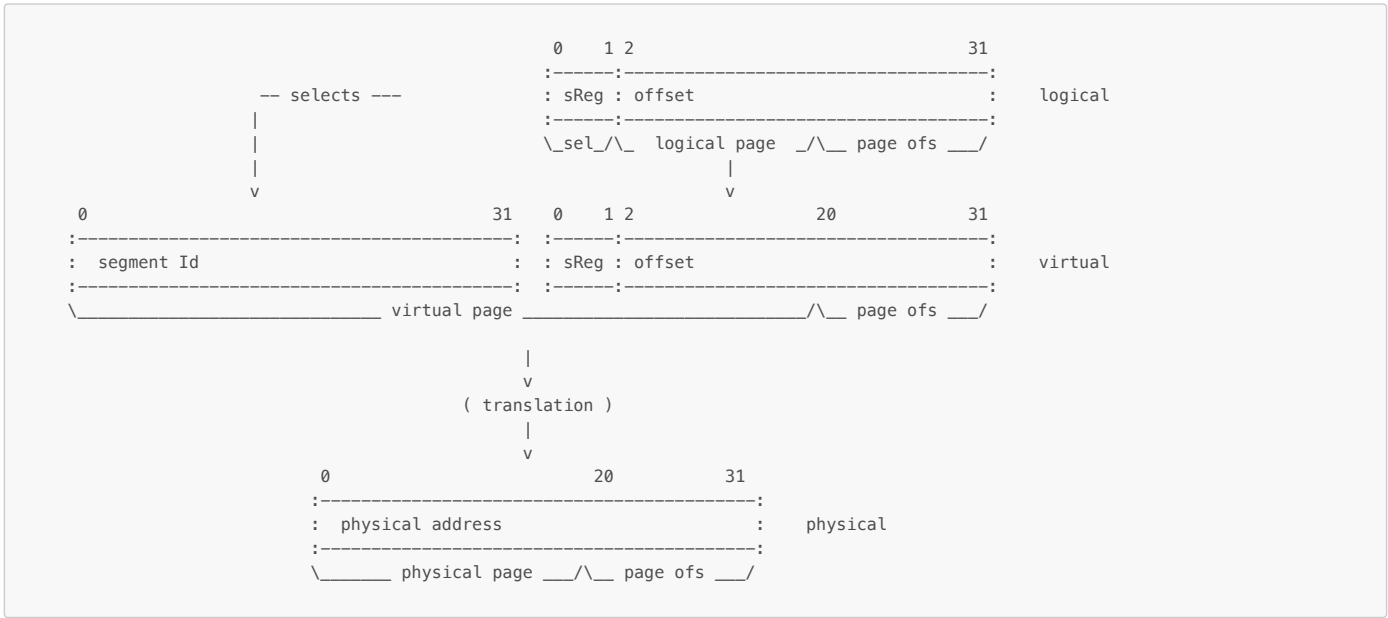


Implementations may not utilize the full 32-bit segment ID space. For example, a CPU could only implement a 16-bit segment Id, allowing for 65636 segments. Segments should be considered as address ranges from which an operating system allocates virtual memory space for code and data objects.

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 particular 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 a 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 of a segment is reachable via a logical address. It is however more likely that these registers point to different segments though.

The **virtual address** is the concatenation of a segment and an offset. Together they form a maximum address range of 2^32 segments with 2^32 bits 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 is the offset directly mapping to the physical address range with the segment part ignored. 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 not preclude supporting a larger physical address range though. A translation buffer hardware could allow for a larger physical address range beyond 4BBytes. This physical memory is however only reachable when translation is enabled.

Access Control

For all segments except the non-zero segment 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. On each instruction execution, the privilege bit in the instruction address is checked against the access type and privilege information field in the access control field, stored in the TLB. Access type and privilege level form the access control information field, which is checked for each 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. 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.

```
 0           2     3
:------------:-----:-----:
: type       : PL1 : PL2 :
:------------:-----:-----:

type: 0 - read-only,   read: PL <= PL1,    write : not allowed,   execute: not allowed
type: 1 - read-write,  read: PL <= PL1,    write : PL <= PL2,     execute: not allowed
type: 2 - execute,     read: PL <= PL1,    write : not allowed,   execute: PL2 <= PL <= PL1
type: 2 - gateway,     read: not allowed,  write : not allowed,   execute: PL2 <= PL <= PL1
```

The second dimension of protection is a **protection ID**. A protection ID is a value assigned to the segment address range at object creation time. The processor control register section contains four protection ID registers. For each access to a segment that has a non-zero protection ID associated, one of the protection ID registers must match the segment protection ID. If not, a protection violation trap is raised.

```
 0                               14  15
:-----------------------------------:--:
: Protection ID                     : W :
:-----------------------------------:--:
```

Protection IDs are typically used to form a grouping of access. A good example is a stack data segment, which is accessible at user privilege level in read and write access from every thread that has R/W access with user privilege. However, only the corresponding user thread should be allowed to access the stack data segment. If a protection ID is assigned to the user thread, this can easily be accomplished. In addition, the protection ID also features a write disable bit, for allowing a model where many threads can read the protected segment but only few or one can write to it. If the segment Id is zero, the execution level must be privileged and address translation and protection checking is disabled.

Adress translation and caching

The previous section depicted the virtual address translation. 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 TLBs for caching translation results and Caches for data caching. However, split TLBS and unified TLBs, L1 and L2 caches and other schemes are models to consider. This section just outline 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 (TLB). Depending on the hardware implementation, there can be a combined TLB or a separate instruction TLB and data TLB. The TLB is essentially a copy of the page table entry information that represents the virtual page currently loaded at the physical address.

```
TLB fields (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 :X :  : AR        :                 : Protect-ID                                      :
:-----------------------------------------------------------------------------------------------:
: VPN - high                                                                                    :
:-----------------------------------------------------------------------------------------------:
: VPN - low                                              : 0                                    :
:-----------------------------------------------------------------------------------------------:
: reserved                          : 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 to the TLB raises a trap.
B	data reference trap. If the bit is set, access to the data page raises a trap.
X	the code page can be modified at the highest privilege level.
Protection ID	The assigned bit protection ID for the page.
Access Rights	The access rights for the page.
VPN-H	the upper 32 bits of a virtual page address, which are also the segment ID.
VPN-L	the lower 20 bits of a virtual page address, which are the page in the segment.
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 U, D, T, B only apply to a data TLB. The X bit is only applies to the instruction 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. The implementation of the TLB is hardware dependent.

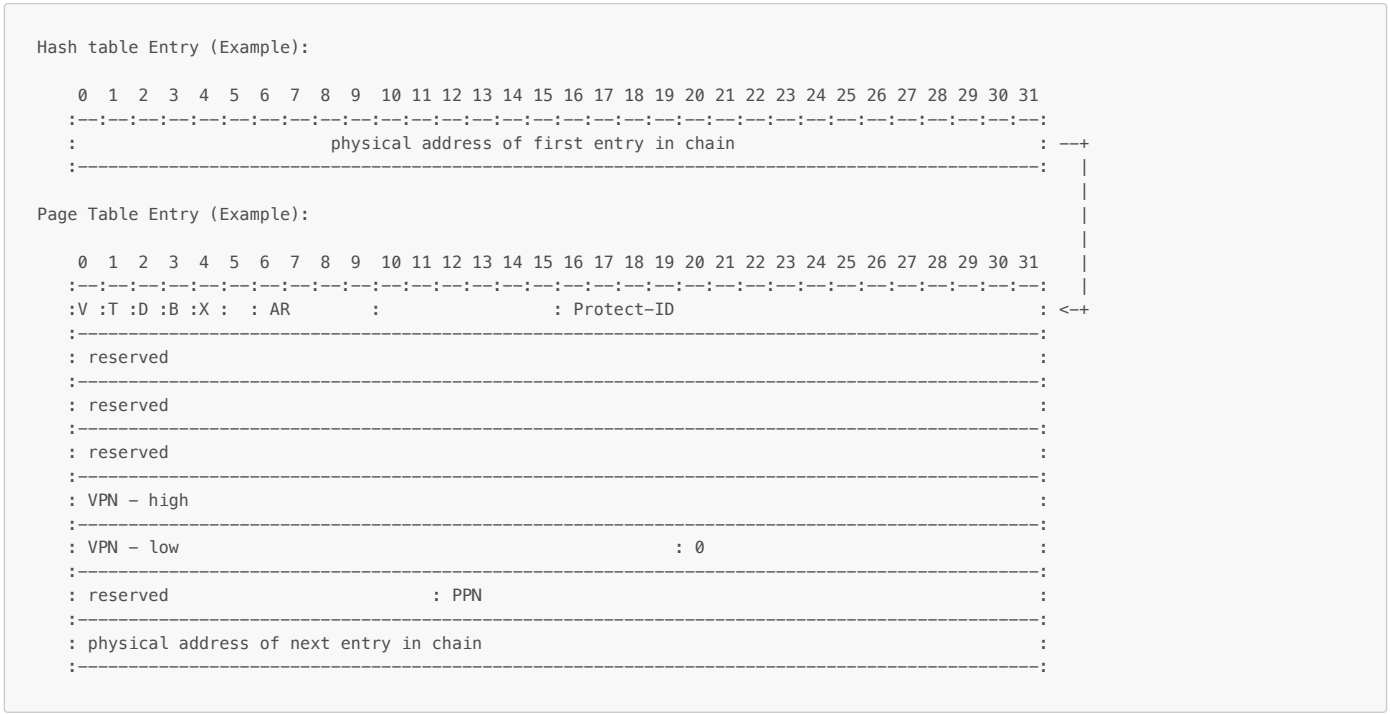
Caches

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.

A system may choose direct map or set-associative caches. Furthermore, there are L1 and L2 cache models. For example, separate instruction and data L1 caches connecting to an external L2 cache with larger cache memories. The architecture just specifies instructions to handle cache operations.

Page Tables

Physical memory is organized in pages with a size of 4K words. The architecture does not specify the page table model. There are again several models of page tables possible. One model is to use an inverted page table model, in which each physical page has an entry in the page table. During a TLB miss, a hash function computes an index into the a hash table which potentially has a link 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 at least the virtual page address, the physical pages address, 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. The example shown above contains several reserved fields. 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.
X	the code page can be modified at the highest privilege level. (**note** better name ?)
Protection ID	The assigned bit protection ID for the page.
Access Rights	The access rights for the page.
VPN-H	the upper 32 bits of a virtual address, which are also the segment ID.
VPN-L	the lower 20 bits of a virtual address, which are the page in the segment.
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 hash table and then follow the chain of page table entires. The following code fragment is 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   = 12;     // number of bits for page offset, page size of 4K
const uint hashTableMask    = 0x3FF;  // a hash table of 1024 entries, memory size dependent, must be a power of two.

uint hash_index ( uint32_t segment, uint32_t offset ) {

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

The function builds the hash index, which may also be used for TLB entries, virtually tagged caches and walking the page table itself. In case of a TLB miss, the hash value is directly available to the miss handler via a control register. In the example above, the segment portion is left shifted by a value of 4. The hash function will map 16 consecutive virtual pages to consecutive indices.

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 relativ 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 the segment is stored in SR0 and the offset in GR1.

Conditional Branches. VCPU-32 features two conditional branch instructions. These instructions compares two register values or test a register 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 gateways. 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. Return from a higher privilege level to a code page with lower privilege level, will automatically set the lower privilege level.

In contrast to PA-RISC, VCPU-32 implement only two privilege levels, **user** and **priv** mode. The privilege status is not encoded in the instruction address, but rather in the process status word. A transition from user to privilege mode is accomplished by setting the "X" bit in the status register when the GATE instruction is executed on a gateway page with appropriate privilege mode settings. 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. If the architecture chooses one day to implement a four level privilege level protection architecture, the access rights fields and where to store the current execution privilege level needs to be revisited. Perhaps it should then just follow the PA-RISC architecture in this matter.

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 ST, IA_SEG, IA-OFS to the I-STAT, I-IA-SEG and I-IA-OFS control reg.
- save interrupted instruction to I-TEMP0.
- copy few GRs to shadow GRs( CRs ) to have room to work ( also SRs ? Tbd. )
- set ST  to zero, this will turn address translation and protection checking off, putting the CPU in privileged mode.
- set IA-SEG to zero.
- set IA-OFS to 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	-	-	-
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 adr segId	data adr ofs
18	**Break instruction trap**	IA of the current instruction	instr	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.

Since a code page is non-writeable, there needs to be a mechanism to allow the temporary modification of the instruction. One approach is to allocate a debug bit for the page table. The continuous instruction privilege check would also check the debug bit and in this case allow to write to a code page. All these modifications only take place at the highest privilege level. When resuming the debugger, the original instruction executes and after execution, the break point instruction needs to be set again, if desired. This would be the second trap, but this time only the break point is set again.

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.

Instruction Set Overview

This chapter gives a brief overview on the instruction set. The instruction set is divided into five general groups of instructions. There are memory reference, immediate, branch, computational and system control type instructions. This chapter will present an overview on the general layout, instruction encoding and the high level way of describing the instruction operation in the chapters to follow.

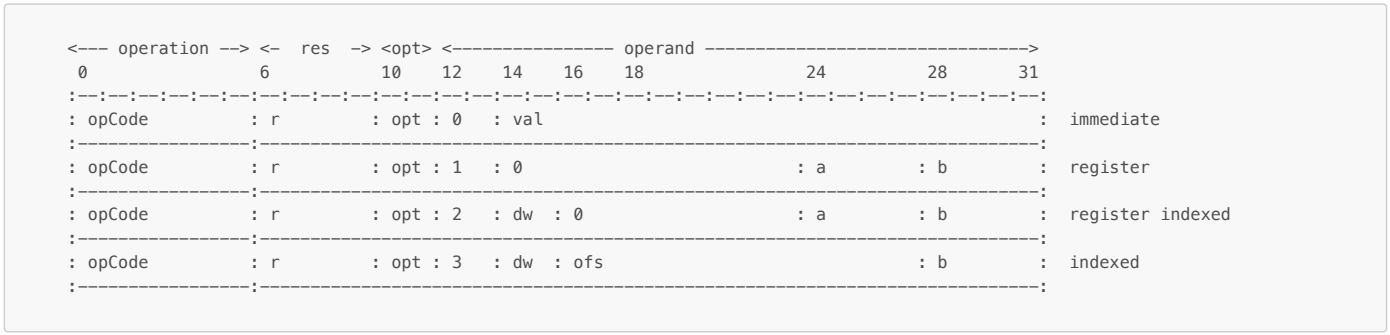
General Instruction Encoding

The instruction uses fixed word length instruction format. Instructions are always one machine word. In general there is a 6 bit field to encode the instruction opCode. The instruction opCode field is encoded as follows:

In the interest of a simplified hardware design, the opCode and opCode specific instruction fields are regular and on fixed positions whenever possible. As shown above, the opCode is always at bits 0 .. 5. But also registers, modes and offsets are at the same locations. The benefit is that the decoding logic can just extract the fields without completely decoding the instruction upfront. For some instructions not all bits instruction word are used. They are reserved fields and should be filled with a zero for now. Throughout the remainder of this document numbers are shown in three numeric formats. A decimal number is a number starting with the digits 1 .. 9. An octal number starts with the number 0, and a hex number starts with the "0x" prefix.

Operand Encoding

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 modes, register modes, and addressing modes. The following figure gives an overview of the instruction layout for instructions with operand encoding.



There is one **immediate operand mode**, which 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. An example would be an 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 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 four segment register 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. The last option 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 operand address is built by adding an signed byte offset to the base register "b". 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.

Instruction Operand Notation

Each instruction is also presented in an assembler style format. The field names found in the instruction format map to the names used in the assembler notation. The names used in the format, such as "r", "a" or "val" directly map to the fields in the instruction layout. An exception are the opMode format instructions where the names "opMode" and "opArg" are used. The following table shows these formats.

Operand Format	opMode Group	Assembler Notation Example	Comment
Immediate	0	ADD r1, 500	add 500 to the general register r1
One register	1	---	SubMode 0. A zero value and one register as operand operation. This opMode combination is typically used for implementing pseudo instructions
Two register	1	OR r2, r6, r7	SubMode 1. A two register operation. "r2" in "r", "r6" in "a" and "r7" in "b"
Register indexed	2	LD r6, r4(r10)	Add r4 to r10 and use the upper two bits of the result to select among segment registers 4 to 7. "seg" field is 0, "dw" field is 0, r4 in "a", r10 in "b" and r6 in "r"
Register indexed	2	LDH r6, r4(sr1, r10)	Add r4 to r10 and use segment s1 as the segment register. "seg" field is 1, "dw" field is 1, r4 in "a", r10 in "b" and r6 in "r"
Indexed	3	ST 100(r12), r4	Store r4 to the virtual memory address formed by adding 100 to r12 and selecting based on the upper two bits the segment register. "seg" field is 0, "dw" field is 0, "ofs" field is 100, r12 in "b"
Indexed	3	ORH r5, 100(s1, r12)	OR r5 to the halfword content found at virtual memory address formed by adding 100 to r12 and segment register s1. "seg" field is one, "dw" field is 1, "ofs" field is 100, r12 in "b"

The instruction described in the following chapter that use an operand encoding will refer to it with the name "operand". As shown above, the operand syntax controls how the parameters "opMode", "seg", "dw", "val", "a" and "b" are set in an instruction. For opMode zero, the subModes are encoded in the "seg" field.

Instruction Operation Notation

The instructions described in the following chapters contain the instruction word layout, a short description of the instruction and also a high level pseudo code of what the instruction does. 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 control register also have dedicated names, such as for example the shift amount control register, labelled "shamt". Instruction operation 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.
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, bitLen)	loads data from virtual or physical memory. The "seg" parameter is the segment and "ofs" the offset into the segment. The bitLen is the number of bits to load. If the bitLen 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, bitLen)	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.
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.
searchInstructionTlbEntry(seg, ofs, entry)	lookup the TLB entry and if found return a pointer to the entry.
allocateInstructionTlbEntry(seg, ofs, entry)	allocate an entry in the TLB and return a pointer to the entry.
purgeInstructionTlbEntry(entry)	purge the TLB entry by simply invalidating the entry.
searchDataTlbEntry(seg, ofs, entry)	lookup the TLB entry and if found return a pointer to the entry.
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.
allocateDataTlbEntry(seg, ofs, entry)	allocate an entry in the TLB and return a pointer to the entry.
purgeDataTlbEntry(entry)	purge the TLB entry by simply invalidating it.
flushDataCache(seg, ofs)	write the cache line that contains the virtual address back to memory and purge the entry.
purgeDataCache(seg, ofs)	remove the cache line that contains the virtual address by simply invalidating it.
purgeInstructionCache(seg, ofs)	remove the cache line that contains the virtual address by simply invalidating it.

Instruction groups

The next chapters present the instruction set. The set itself is divided into several groups. The **memory reference** group contains the instructions to load ans 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 of 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, will 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 **LDx** and **STx** instruction are load and store using the operand encoding format to specify the actual address. The final virtual address is built using the upper two bits of the computed address offset to select among the segment registers SR4..SR7. The **LDEx** and **STEx** implement access to virtual memory using segment register SR1..SR3 and and offset. These instruction can form a virtual address to access the entire address range. The instructions use a **W** for word, a **H** for half-word and a **B** for byte operand size.

The **LDAW**, **LDWAX** and **STAW** instruction implement word access to the physical memory computing a physical address to access.

The **LDWR** and **STWC** instructions support atomic operations. The **LDWR** instruction loads a value form memory and remember this access. The **STWC** instruction will store a value to the same location that the LR instructions used and return a failure if the value was modified since the last LR 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 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. 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. The extended address mode instructions will raise a trap.

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 offset computation. A negative offset will be added before the address computation, a positive offset after the address computation.

LDW, LDH, LDB

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

Format

```
LDx [.M] r, operand
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDx    ( 0x1A ) : r         :0 :M : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The load instruction will load the operand into the general register "r". See the section on operand encoding for the defined operand modes. OpModes 0 and 1 are undefined for this instruction and result in an illegal instruction trap. 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 offset must match the alignment size of the data to fetch.

Operation

```
if ( opMode < 2 ) illegalInstructionTrap( );

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

GR[r] <- zeroExtend( memLoad( seg, ofs, len ), len );
```

Exceptions

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

Notes

The LDx and STx instructions uses a logical address and can access a quadrant defined by the upper two bits of the computed address offset to select among SR4..SR7. This is the most common usage of the load/store instruction and the implicit segment register selection allows for larger offset in the instruction. For accessing a data item anywhere in the virtual address range, the extended load and store instructions are available.

STW, STH, STB

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

Format

```
STx [.M] operand, 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: STx    ( 0x1B ) : r         :0 :M : operand                                                    :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The load instruction will store the data in general register "r" to memory. See the section on operand encoding for the defined operand modes. Operand modes 0 and 1 are undefined for this instruction and result in illegal instruction trap. 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. Note that there is no register indexed mode for the STx instruction.

Operation

```
if ( opMode < 2 ) illegalInstructionTrap( );

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

memStore( seg, ofs, GR[r], len );
```

Exceptions

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

Notes

The LDx and STx instructions uses a logical address and can access a quadrant defined by the upper two bits of the computed address offset to select among SR4..SR7. This is the most common usage of the load/store instruction and the implicit segment register selection allows for larger offset in the instruction. For accessing a data item anywhere in the virtual address range, the extended load and store instructions are available. In contrast to the LDx instruction, the STx instruction would need to access three registers (the value to store, the base and index register), which would require a greater hardware effort.

LDEW, LDEH, LDEB

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

Format

```
LDEx [.M] r, ofs(b)
LDEXx [.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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDEx   ( 0x30 ) : r         :0 :M : seg : dw  : ofs                              : b         :
:-----------------:-------------------------------------------------------------------------------:
: LDEXx  ( 0x30 ) : r         :1 :M : seg : dw  : 0                     : a        : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The load extended instruction will load the operand into the general register "r". The "seg" field selects among the segment registers. A value of zero will use the upper two bits of the computed offset to select among SR4..SR7. All other values select among SR1..SR3. The "dw" field defines the data width to load. The "dw" field specifies the data width. The offset is computed by adding the offset or the content of "a" to "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 offset must match the alignment size of the data to fetch.

Operation

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

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

len <- instr.[14..15];
seg   <- segSelect( instr[12..13] );
GR[b] <- GR[b] + lowSignExtend( ofs, 12 );

GR[r] <- zeroExtend( memLoad( seg, offset, len ), len );
```

Exceptions

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

Notes

The load and store extended instruction family allow data access across the entire virtual address range using segment registers SR1..SR3. The address offset can reach any data byte in the segment address range.

STEW, STEH, STEB

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

Format

```
STEx [.M] ofs(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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: STEx   ( 0x32 ) : r         :0 :M : seg : dw  : ofs                             : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The store extended instruction will store general register "r" to the virtual address. The "seg" field selects among the segment registers. A value of zero will use the upper two bits of the computed offset to select among SR4..SR7. All other values select among SR1..SR3. The "dw" field defines the data width to load. The "dw" field specifies the data width. The offset is computed by adding the offset to "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 offset must match the alignment size of the data to fetch.

Operation

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

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

len <- instr.[14..15];
seg   <- segSelect( instr[12..13] );
GR[b] <- GR[b] + lowSignExtend( ofs, 12 );

GR[r] <- memStore( seg, ofs, GR[r], len );
```

Exceptions

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

Notes

The load and stored extended instruction family allow data access across the entire virtual address range using segment registers SR1..SR3. The address offset can reach any data byte in the segment address range. The store instruction does not support an indexed mode. In contrast to the LDEXx instruction, the STEx instruction would need to access three registers (the value to store, the base and index register), which would require a greater hardware effort.

LDWA, LDWAX

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

Format

```
LDWA [.M] r, ofs (b)
LDWAX [.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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDWA   ( 0x32 ) : r         :0 :M : ofs                                          : b         :
:-----------------:-------------------------------------------------------------------------:
: LDWAX  ( 0x32 ) : r         :1 :M : 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 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.[M] ) {

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

GR[b] <- GR[b] + lowSignExtend( ofs, 16 );

GR[r] <- memLoad( 0, GR[b] + lowSignExtend( offset, 18 ), 32 );
```

Exceptions

- Privileged operation trap
- Alignment trap

Notes

None.

STWA

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

Format

```
STWA [.M] ofs (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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: STWA   ( 0x33 ) : r         :0 :M : ofs                                       : 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 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.[M] ) {

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

GR[b] <- GR[b] + lowSignExtend( ofs, 16 );

memStore( 0, GR[b] + lowSignExtend( ofs, 18 ), GR[r], 32 );
```

Exceptions

- Privileged operation trap
- Alignment trap

Notes

None.

LDWR

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

Format

```
LDWR r, operand
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDWR   ( 0x1C ) : r         : 0   : operand                                                     :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The LDWR instruction is used for implementing semaphore type operations. See the section on operand encoding for the defined operand mode encoding. The first part of the instruction behaves exactly like the LDW instruction. A logical memory 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 LDWR instruction supports only word addressing.

Operation

```
if (( opMode < 2 ) || ( dataWidth != 0 )) illegalInstructionTrap( );

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

GR[r] <- memLoad( seg, ofs, 32 );

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

Exceptions

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

STWC

Conditionally store a value to memory.

Format

```
STWC operand 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: STWC   ( 0x1D ) : r         : 0   : operand                                                     :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The STWC conditional instruction will store a value in "r" to the memory location specified by the operand address. 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 (( opMode < 2 ) || ( dataWidth != 0 )) illegalInstructionTrap( );

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

   memStore( operandAdrSeg( instr ), operandAdrOfs( instr ), GR[r], 32 );
   GR[r] <- 0;

} else GR[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
- 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...

Immediate Instructions

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 fields is used. 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.

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[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.

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 zero. Any potential overflows are ignored.

Operation

```
GR[0] = GR[r] + ( 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 R0. The instruction sequence

```
ADDIL  R11, L%ofs
LDW    R3, R%ofs(R0)
```

LDO

Loads the effective address offset of the operand.

Format

```
LDO r, operand
```

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

Exceptions

- Illegal Instruction Trap

Notes

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

Branch Instructions

The control flow instructions are divided into unconditional and conditional branch type instructions. The unconditional branch type instructions allow in addition to altering the control flow, the save of 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 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**, except that a general register contains the instruction address relative offset.

The **BV** is an instruction segment absolute instruction. The branch address for the BV instruction is the segment relative offset computed using a segment relative base, adding a signed offset from another register.

The **BE**, **BLE** and **BVE** instruction are the inter-segment branches. The **BE** instruction branches to a segment absolute address encoded in the segment and general offset register. In addition, a signed offset encoded in the instruction can is added to form the target segment offset. The **BLE** instruction will in addition return the return address in the segment register SR0 and the offset on the general register GR0. The **BVE** will use a general register containing a logical address. The segment is selected from the upper two bits.

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 where the GATE instruction resides.

The conditional branch instructions combine an operation such as comparison or test with a local instruction address relative branch if the comparison or test condition is met. The **CBR** instruction 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.

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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BL    ( 0x20 )  : r         : ofs                                                               :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The branch instruction performs a branch to an instruction address relative location. The target address is shifted by 2 to the left and added 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[r]  <- IA-OFS + 4;
IA-OFS <- IA-OFS + ( lowSignExt( ofs << 2 ), 22 );
```

Exceptions

- Taken branch trap.

Notes

Using general register zero as the return link register will essentially be just a branch instruction without saving the return link.

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 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[r]  <- IA-OFS + 4 ;
IA-OFS <- IA-OFS + ( GR[b] << 2 );
```

Exceptions

- Taken branch trap

Notes

Using general register zero as the return link register will essentially be just a branch instruction without saving the return link.

BV

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

Format

```
BV 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BV     ( 0x23 ) : 0                                                  : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The branch vectored instruction performs an unconditional branch by adding the offset register in "a" shifted by two bits 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.

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 result in an instruction protection trap. A branch from a higher privilege to a lower privilege level result in the privilege level adjusted in the status register. Otherwise, the privilege level remains unchained.

Operation

```
IA-OFS <- GR[b] + ( GR[a] << 2 );
```

Exceptions

- Taken branch trap
- Instruction memory protection trap

Notes

The **BV** instruction with general register "a" being register zero is typically used in a procedure return. The **B** instruction with a return link left in a general register, which can directly be used by this instruction to return to the location after the **B** instruction.

BE

Perform an unconditional external branch.

Format

```
BE 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 ) : 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 sign extended offset is added. If code translation is disabled, the offset is the absolute physical address and the "a" field ignored.

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 result in the privilege level adjusted in the status register. Otherwise, the privilege level remains unchained.

Operation

```
IA-SEG <- GR[a];
IA-OFS <- GR[b] + lowSignExt(( ofs << 2 ), 20 );
```

Exceptions

- Taken branch trap
- Instruction memory protection trap

Notes

None.

BLE

Perform an unconditional external branch and save the return address.

Format

```
BLE 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BLE    ( 0x25 ) : ofs                                                 : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The branch and link external instruction branches to an absolute 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 sign extended offset "ofs" is added. The return address is saved in SR0 and GR0. If code translation is disabled, the offset is the absolute physical address and the "a" field being ignored.

Since the BLE 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 result in the privilege level adjusted in the status register. Otherwise, the privilege level remains unchained.

Operation

```
SR[0] <- IA-SEG;
GR[1] <- IA-OFS + 4;

IA-SEG <- GR[a];
IA-OFS <- GR[b] + lowSignExt(( ofs << 2 ), 20 );
```

Exceptions

- Taken branch trap
- Instruction memory protection trap

Notes

None.

BVE

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

Format

```
BVE 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: BVE    ( 0x26 ) : 0                                                  : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The branch and link external instruction branches to an absolute location in another code segment. The target address is built from adding general register "a" shifted by 2 to the base register "b". The segment register is selected based on the upper two bits of general register "b".

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 result in the privilege level adjusted in the status register. Otherwise, the privilege level remains unchained.

Operation

```
IA-SEG <- segSelect( GR[b] );
IA-OFS <- GR[b] + ( GR[a] << 2 );
```

Exceptions

- Taken branch trap
- Instruction memory protection trap

Notes

None.

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 adding "ofs" shifted by 2 bits to the left 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 resulting privilege level is deposited in the P bit of the GR "r". Execution continues at the target address with the new privilege level.

Operation

```
GR[r]  <- ST.[P];
IA-OFS <- IA-OFS + lowSignExt(( ofs << 2 ), 24 );

if ( ST.[C] ) {

   tmp = IA-OFS + lowSignExt(( ofs << 2 ), 24 );

   searchInstructionTlbEntry( IA-SEG, tmp, &entry );

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

} else ST.[P] <- 0;
```

Exceptions

None.

Notes

None.

CBR

Compare two registers and branch on condition.

Format

```
CBR [ .<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    ( 0x27 ) : cond   : ofs                                    : a         : b         :
:-----------------:-------------------------------------------------------------------------:
```

Description

The CBR instruction compares general registers "a" and "b" for the condition specified in the "cond" field. If the condition is met, the target branch address is the 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 )                          4 : NE  ( a != b )
1 : LT ( a < b, Signed )                    5 : LE ( a <= b, Signed )
2 : GT ( a > b, Signed )                    6 : GE ( a >= b, Signed )
3 : LS ( a <= b, Unsigned )                 7 : HI ( a > b, Unsigned )
```

Operation

```
switch( cond ) {

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

if ( res ) IA-OFS <- IA-OFS + lowSignExt(( ofs << 2 ), 17 );
else       IA-OFS <- IA-OFS + 4;
```

Exceptions

None.

Notes

Often a comparison is followed by a branch in an instruction stream. VCPU-32 therefore features conditional branch instructions that both offer the combination of a register evaluation and branch depending on the condition specified.

Computational Instructions

The arithmetic, logical and bit field operations instruction represent the computation type instructions. Most of the computational instructions use the operand instruction format, where the operand fields, opMode and opArg, define one of the instruction operands. The computation instructions are divided into the numeric instructions **ADD**, **ADDC**, **SUB** and **SUBC**, 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. The LDI instruction allows to load a 16-bit value into the right or left half of a register. There are options to clear the register upfront, and to deposit the respective half without modifying the other half of the register. This way immediate values with a full 32-bit range can be constructed.

The **LDSID** instruction will return the segment id resulting from the operand mode and argument.

ADD, ADDC

Adds the operand to the target register.

Format

```
ADD[.<opt>] r, operand
ADDC[.<opt>] r, operand
```

```
 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                                                    :
:----------------:-------------------------------------------------------------------------------:
: ADDC   ( 0x11 ) : 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. The **ADDC** instruction behaves identical the **ADD** instruction, except it also adds the carry bit from the status register.

Operation

```
switch( opMode ) {

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

   case 1: tmpA + tmpB + instr.[C]; break;

   case 2:
   case 3: {

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

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

   } break;
}

for ADD:  res <- GR[a]; tmpB <- GR[b];

for ADDC: res <- tmpA + tmpB + instr.[C];

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

   GR[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
- Alignment trap

Notes

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

SUB, SUBC

Subtracts the operand from the target register.

Format

```
SUB[.<opt>] r, operand
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: SUB    ( 0x12 ) : r         :C :L : operand                                                    :
:----------------:-------------------------------------------------------------------------------:
: SUBC   ( 0x13 ) : r         :C :L : 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. The **SUBC** instruction behaves identical the **SUB** instruction, except it also adds the carry bit from the status register.

Operation

```
  switch( opMode ) {

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

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

   case 2:
   case 3: {

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

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

   } break;
}

for SUB: res <- tmpA - tmpB;

for SUBC: res <- tmpA - tmpB + instr.[C];

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

   GR[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
- Alignment trap

Notes

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

AND

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

Format

```
AND[.<opt>] r, operand
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: AND    ( 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 allows to negate the result making the AND a NAND operation. The "C" bit allows to complement (1's complement) the operand input, which is the "b" register. See the section on operand encoding for the defined operand modes.

Operation

```
switch( opMode ) {

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

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

   case 2:
   case 3: {

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

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

   } break;
}

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

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

Exceptions

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

OR

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

Format

```
OR[.<opt>] r, operand
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: OR     ( 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 allows to negate the result making the OR a NOR operation. The C bit allows to complement (1's complement) the operand input, which is the regB register. See the section on operand encoding for the defined operand modes.

Operation

```
switch( opMode ) {

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

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

   case 2:
   case 3: {

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

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

   } break;
}

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

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

Exceptions

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

XOR

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

Format

```
XOR[.<opt>] r, operand
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: XOR    ( 0x16 ) : r         :N :C : 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 OR a XNOR operation. The "C" bit allows to complement (1's complement) the operand input, which is the "b"" register. See the section on operand encoding for the defined operand modes.

Operation

```
switch( opMode ) {

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

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

   case 2:
   case 3: {

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

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

   } break;
}

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

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

Exceptions

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

Notes

None.

CMP, CMPU

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

Format

```
CMP[.cond>] r, operand
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: CMP    ( 0x17 ) : r         :cond : operand                                                    :
:----------------:-------------------------------------------------------------------------------:
: CMPU   ( 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 )                          1 : NE ( r != operand )
2 : LT ( r <  operand, Signed )                  3 : LE ( r <= operand, Signed )
```

Operation

```
switch( opMode ) {

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

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

   case 2:
   case 3: {

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

      tmpA <- GR[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[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 instructions can also be used to implement the greater than (GT) and greater or equal (GE) condition.

CMP.GT r1, r2 -> CMP.LE r2, r1 CMP.GE r1, r2 -> CMP.LT r2, r1

CMR

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

Format

```
CMR [.<cond>] r, 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: 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 )                           4 : NE  ( b != 0 )
1 : LT ( b < 0, Signed )                    5 : LE ( b <= 0, Signed )
2 : GT ( b > 0, Signed )                    6 : GE ( b >= 0, Signed )
3 : EV ( even( b ))                         7 : OD ( odd( b ))
```

Operation

```
switch( cond ) {

   case 0: res <- ( GR[b] == 0 );   break;
   case 1: res <- ( GR[b] > 0 );    break;
   case 2: res <- ( GR[b] < 0 );    break;
   case 3: res <- ( ~ GR[b].[31] ); break;
   case 4: res <- ( GR[b] != 0 );   break;
   case 5: res <- ( GR[b] <= 0 );   break;
   case 6: res <- ( GR[b] >= 0 );   break;
   case 7: res <- ( GR[b].[31] );   break;
}

if ( res ) GR[r] <- GR[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: ( R2 -> a, R6 -> b, R1 -> c, R4 -> d )

   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.

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          : 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. It is encoded as 31- pos. 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.

Operation

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

GR[r] <= extract( GR[b], pos, len );

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

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
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: DEP    ( 0x06 ) : r         :Z :A :I : 0            : len          : pos          : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The instruction extracts the right justified bit field of length "len" in general register "b" and deposits this field in the general register "r" at the specified position. The "pos" field specifies the rightmost bit for the bit field to deposit. It is encoded as 31- pos. 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". 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.[A] ) tmpPos <- SHAMT.[27..31];
else tmpPos <- pos;

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

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

deposit( GR[r], tmpB, pos, 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 synthesized instructions.

DSR

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

Format

```
DSR [.<opt>] r, b, a, shAmt
```

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: DSR    ( 0x07 ) : r         :A : 0                 : shamt        :0 : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

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 <- shamt;

if ( instr.[Z] )  tmp = 0;
else              tmp = GR[a];

GR[r] <- rshift( cat( GR[b], tmp ), shamt );
```

Exceptions

- illegal instruction trap.

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 pseudo instructions.

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

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

Exceptions

- Overflow trap

Notes

None.

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

Exceptions

None.

Notes

None.

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 a segment or control register. 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 word. 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.

MR

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

Format

```
MR [.<opt>] r, s
```

```
 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 instruction MR copies data from a segment or control register "s" to a general register "r" and vice versa. If the "D" bit is set, the copy direction is from "r" to a segment or control register in "s", otherwise from "s" to "r". The "M" bit indicates whether a segment register or a control register is moved. Setting a value for a privileged segment or control register is a privileged operation.

Operation

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

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

} else {

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

Exceptions

- privileged operation trap.

Notes

Although there are option bits for direction and register type, the assembler will simplify the instruction syntax. By deducing the the type of register, the "D" and "M" bits are set by the assembler.

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                                          : val / b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The MST instruction sets the status bits 26.. 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 26..31 with the bits in the general register "b". Mode 1 and 2 will interpret the 6-bit field bits 26..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 26..31 )
1 - set status bits using the bits in "val" ( bits 26..31 )
2 - clears status bits using the bits in "val" ( bits 26..31 )
3 - undefined.
```

Operation

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

GR[r] <- cat( 0.[0..25], ST.[26..31] );

switch( mode ) {

   case 0: {

       ST.[26..31] <- GR[b].[26..31];

   } break;

   case 1: {

      if ( instr.[26] ) ST.[26] <- 1;
      if ( instr.[27] ) ST.[27] <- 1;
      if ( instr.[28] ) ST.[28] <- 1;
      if ( instr.[29] ) ST.[29] <- 1;
      if ( instr.[30] ) ST.[30] <- 1;
      if ( instr.[31] ) ST.[31] <- 1;

   } break;

   case 2: {

      if ( instr.[26] ) ST.[26] <- 0;
      if ( instr.[27] ) ST.[27] <- 0;
      if ( instr.[28] ) ST.[28] <- 0;
      if ( instr.[29] ) ST.[29] <- 0;
      if ( instr.[30] ) ST.[30] <- 0;
      if ( instr.[31] ) ST.[31] <- 0;

    } break;

   default: illegalInstructionTrap( );
}
```

Exceptions

- privileged operation trap

Notes

None.

LDPA

Load the physical address for a virtual address.

Format

```
LDPA r, operand
```

```
 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    ( 0x37 ): r         : 0         : seg :                       : 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 form by adding "a" to "b" and using the segment selector in "seg". The result is ambiguous for a virtual address that translates to a zero address. LDPA is a privileged operation.

Operation

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

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

GR[r] <- loadPhysAdr( seg, ofs );
```

Exceptions

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

Notes

None.

PRB

Probe data access to a virtual address.

Format

```
PRB r, operand
```

```
 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         :M :P :I :0 :seg  : 0                    : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The PRB instruction determines whether data access at the requested privilege level stored in the general register "r" is allowed. If the probe operation succeeds, a value of one is returned in "r", otherwise a zero is returned. The "M" 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 "P" bit designates the privilege level to use for the probe operation. The "I" bit, if set, uses the value in register "a" as the privilege level to test for instead of the "P" bit. If protection ID 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 ( opMode < 2 ) illegalInstructionTrap( );

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

if ( ! searchDataTlbEntry( seg, ofs, &entry )) {

   if ( instr.[I] ) {

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

   } else {

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

} else nonAccessDataTlbMiss( );
```

Exceptions

- non-access data TLB miss trap

Notes

None.

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 :M : 0                              : 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 data to be inserted is grouped into two steps. The first step will enter the virtual address and physical address is associated with. The second step will enter the access rights and protection information and marks the entry valid. The TLB entry is set to invalid and the address fields are filled. 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 either the physical address or the access rights and protection information. The "M" bit indicates which part of the TLB insert operation is being requested. A value of zero refers to part one, which will enter the address information, a value of one will enter the access rights and protection information. The sequence should be to first issue the ITLB.A instruction for the address part, which still leaves the TLB entry marked invalid. Issuing the second part, the ITLB.P instruction, which sets the access rights and protection information, will mark the entry valid after the operation.

Argument Word Layout

Depending on the "T" bit, the protection information part or the address part is passed to the TLB subsystem. The individual bits are described in the architecture chapter TLB section.

```
 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: 0                               : PPN                                                 :      T = 0
:-------------------------------------------------------------------------------------------:

 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 :X :  : AR        :                 : Protect-ID                                :    T = 1
:-------------------------------------------------------------------------------------------:
```

Operation

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

if ( instr.[T] )
   if ( ! searchDataTlbEntry( SR[a], GR[b], &entry )) allocateDataTlbEntry( SR[a], GR[b], &entry );
else
   if ( ! searchInstructionTlbEntry( SR[a], GR[b],, &entry )) allocateInstructionTlbEntry( SR[a], GR[b], &entry );

if ( instr.[M] ) {

   entry.[addressPart] <- GR[r];
   entry.[V] <- false;

} else {

   entry.[protectPart] <- GR[r];
   entry.[V] <- true;
}
```

Exceptions

- Privileged operation trap

Notes

None.

PTLB

Removes a translation entry from the TLB.

Format

```
PTLB <opt> (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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: PTLB   ( 0x3C ) : 0         :T : 0                                  : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The DTLB instruction removes a translation from the instruction or data TLB by marking the entry invalid. The virtual address is encoded in "a" for the segment register and "b" for the offset. 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.

Operation

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

if ( instr.[T] )
   if ( searchDataTlbEntry( SR[a], GR[b], &entry )) purgeDataTlbEntry( SR[a], GR[b], &entry );
else
   if ( searchInstructionTlbEntry( SR[a], GR[b], &entry )) purgeInstructionTlbEntry( SR[a], GR[b], &entry );
```

Exceptions

- privileged operation trap

Notes

None.

PCA

Flush and / or remove cache lines from the cache.

Format

```
PCA <opt> (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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: PCA    ( 0x3D ) : 0         :T :F : 0                                : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
```

Description

The PCA instruction flushes or purges a cache line from the instruction or data cache. The virtual address is encoded in "a" for the segment register and "b" for the offset. An instruction cache line can only be purged, a data cache line can also be written back to memory and then optionally purged. A flush operation is only performed when the cache line is dirty. The "T" bit indicates whether the instruction or the data cache is addressed. A value of zero references the instruction cache. The "F" bit will indicate whether the data cache is to be purged without flushing it first to memory. If "F" is zero, the entry is first flushed and then purged, else just purged. The "F" bit has no meaning for an instruction cache.

Operation

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

   if ( ! instr.[F] ) flushDataCache( SR[a], GR[b] );
   purgeDataCache( SR[a], GR[b] );

} else purgeInstructionCache( SR[a], GR[b] );
```

Exceptions

- privileged operation trap
- Non-access ITLB trap
- Non-access DTLB trap

Notes

None.

DIAG

Issues commands to hardware specific components and implementation features.

Format

```
DIAG 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: DIAG   ( 0x3E ) : r         :  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 if the input and output arguments are processor implementation dependent and described in the respective processor documentation.

Operation

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

... perform the requested operation
```

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-IA-SEG, I-IA-OFS and I-STAT.

Operation

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

IA_SEG <- I-IA-SEG;
IA-OFS <- I-IA-OFS;
ST     <- I-STAT;
```

Exceptions

- privileged operation trap

Notes

The RFI instruction is also used to perform a context switch. Changing from one thread to another is accomplished by loading the control registers **I-IA-SEG**, **I-IA-OFS** and **I-STAT** 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 and 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.

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( info1, 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.

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.

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. An assembler could offer a generic "load immediate" synthetic instruction and either use a one instruction or two instruction sequence. 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.

// ??? **note** need a better to present them ... there are many...

The following table shows potential synthetic instructions.

Pseudo 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.
LDI	LDI GRx, val	LDO GRx, val	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.
CLR	CLR GRn	OR GRn, GR0	Clears a general register. There are many instructions that can be used for a CLR.
MR	MR GRx, GRy	OR GRx, GRy, GR0 ; This overlaps the MR instruction for moving two general registers.	
ASR	ASR GRx, shamt	EXTR.S 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. 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	
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	ADD [.< opt >] GR x, val	
DEC	DEC [.< opt >] GRx, val	SUB [.< opt >] GR x, val	
NEG	NEG GRx	SUB [.] GRx, GRx, opMode 1	
COM	COM GRx	OR.N GRx, GRx, opMode 1	
LDSR	LDSR SRx, GRy	MR SRx, GRy	Assembler deduces form S and G register position the D and M flags.
LDCR	LDCR CRx, GRy	MR SRx, GRy	Assembler deduces form S and G register position the D and M flags.
STSR	STSR GRx, SRy	MR SRx, GRy	Assembler deduces form S and G register position the D and M flags.
STCR	STCR GRx, CRy	MR SRx, GRy	Assembler deduces form S and G register position the D and M flags.

| ... | ... | ... | more to come ... | |||||

Instruction Set Summary

This appendix lists all instructions by instruction group.

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

```
 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         :C :L : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: ADDC    ( 0x11 ): r         :C :L : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: SUB     ( 0x12 ): r         :C :L : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: SUB     ( 0x13 ): r         :C :L : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: AND     ( 0x14 ): r         :N :C : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: OR      ( 0x15 ): r         :N :C : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: XOR     ( 0x16 ): r         :N :C : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: CMP     ( 0x17 ): r         :cond : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: CMPU    ( 0x18 ): r         :cond : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
```

```
 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         : pos         : b         :
:-----------------:-----------------------------------------------------------------------------:
: DEP     ( 0x06 ): r         :Z :A :I :            : len         : 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         :
:-----------------:-----------------------------------------------------------------------------:
```

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/H/B ( 0x1A ): r         : 0   : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: STW/H/B ( 0x1B ): r         : 0   : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: LDWR    ( 0x1C ): r         : 0   : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
: STWC    ( 0x1D ): r         : 0   : operand                                                   :
:-----------------:-----------------------------------------------------------------------------:
```

Immediate Instructions

```
:-----------------:-------------------------------------------------------------------------------:
: LDIL    ( 0x01 ): r         : val                                                               :
:-----------------:-------------------------------------------------------------------------------:
: ADDIL   ( 0x02 ): r         : val                                                               :
:-----------------:-------------------------------------------------------------------------------:
: LDO     ( 0x03 ): r         : ofs                                                 : b           :
:-----------------:-------------------------------------------------------------------------------:
```

Extended Memory Reference 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDWE    ( 0x30 ): r         :0 :M : seg : dw  : ofs                                 : b           :
:-----------------:-------------------------------------------------------------------------------:
: LDWEX   ( 0x30 ): r         :1 :M : seg : dw  : 0                       : a         : b           :
:-----------------:-------------------------------------------------------------------------------:
: STWE    ( 0x31 ): r         :0 :M : seg : dw  : ofs                                 : b           :
:-----------------:-------------------------------------------------------------------------------:
: STWEX   ( 0x31 ): r         :0 :M : seg : dw  : 0                       : a         : b           :
:-----------------:-------------------------------------------------------------------------------:
```

Absolute Memory Reference 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
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: LDWA    ( 0x32 ): r         :0 :M : ofs                                             : b           :
:-----------------:-------------------------------------------------------------------------------:
: LDWAX   ( 0x32 ): r         :1 :M : 0                                   : a         : b           :
:-----------------:-------------------------------------------------------------------------------:
: STWA    ( 0x33 ): r         :0 :M : ofs                                             : 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 ): 0                                             : a         : b           :
:-----------------:-------------------------------------------------------------------------------:
: BE      ( 0x24 ): ofs                                           : a         : b           :
:-----------------:-------------------------------------------------------------------------------:
: BLE     ( 0x25 ): ofs                                           : a         : b           :
:-----------------:-------------------------------------------------------------------------------:
: BVE     ( 0x26 ): 0                                                         : b           :
:-----------------:-------------------------------------------------------------------------------:
: CBR     ( 0x27 ): cond   : ofs                                  : a         : b           :
:-----------------:-------------------------------------------------------------------------------:
```

System Control Instructions

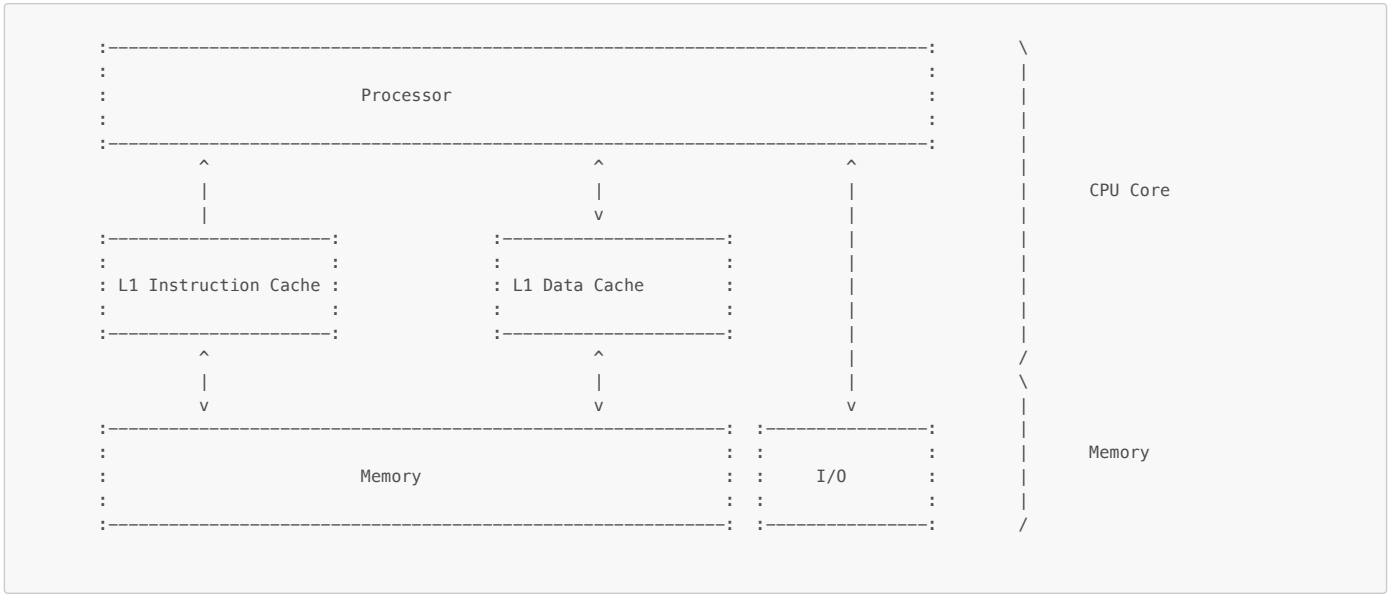
```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--:
: MR      ( 0x0A ): r         :D :M : 0                                       : s               :
:-----------------:-------------------------------------------------------------------------------:
: MST     ( 0x0B ): r         :mode : 0                                       : s               :
:-----------------:-------------------------------------------------------------------------------:
: LDPA    ( 0x37 ): r         : 0         : seg :                       : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: PRB     ( 0x3A ): r         :M :P :I :0 : seg : 0                     : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: ITLB    ( 0x3B ): r         :T :M : 0                                 : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: PTLB    ( 0x3C ): 0         :T : 0                                    : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: PCA     ( 0x3D ): 0         :T :F : 0                                 : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: DIAG    ( 0x3E ): r         : 0                                       : a         : b         :
:-----------------:-------------------------------------------------------------------------------:
: RFI     ( 0x3F ): 0                                                                             :
:-----------------:-------------------------------------------------------------------------------:
: BRK     ( 0x00 ): info1     : 0                   : info2                                       :
:-----------------:-------------------------------------------------------------------------------:
```

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 L1 Cache

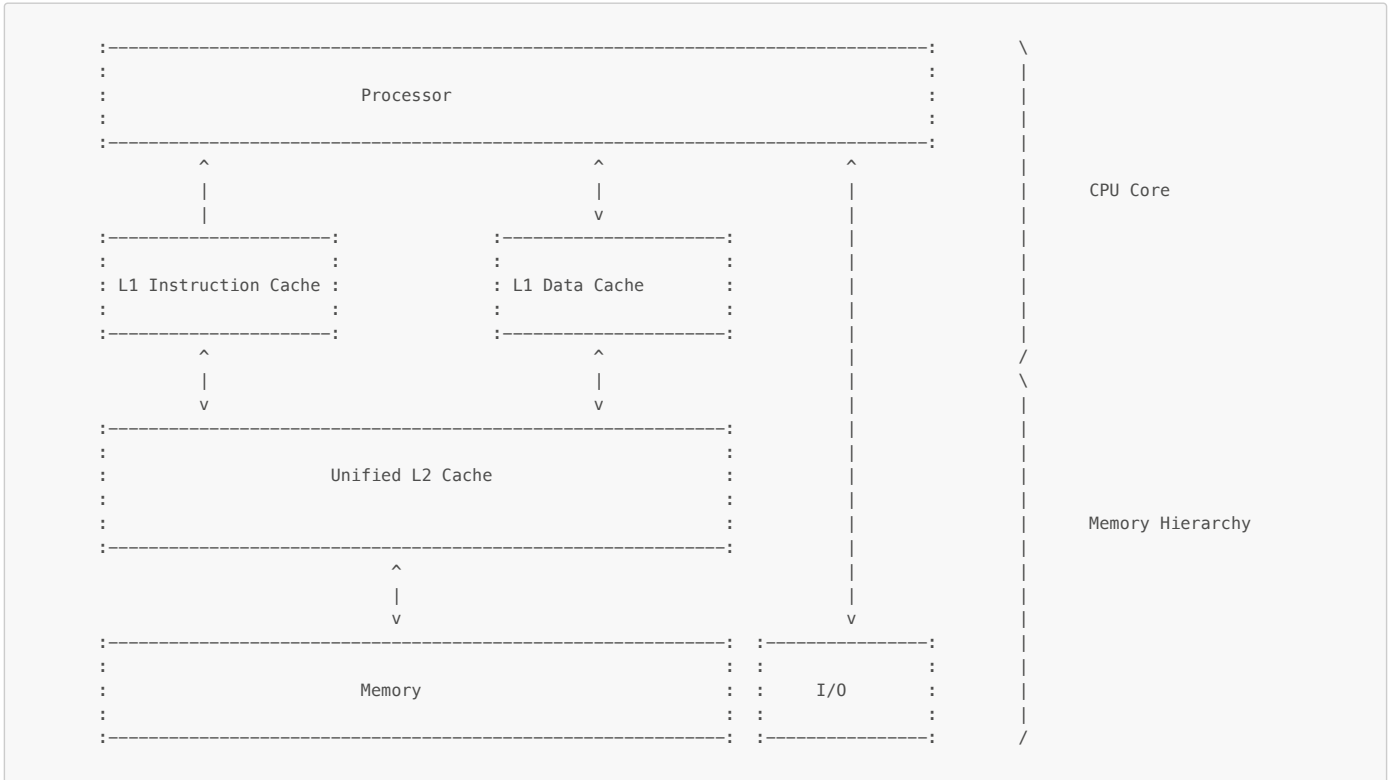
The processor unit and the L1 caches form the "CPU core". The pipeline design makes a reference to memory during instruction fetch and then optional data access. Since both operations potentially take place for different instructions but in the same cycle, a separate **instruction cache** and **data cache** is a key part of the overall architecture. These two caches are called **L1 caches**.



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.

Unified L2 Cache

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.

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 ca 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.

Separate Instruction and Data TLB

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.

Joint TLBs

Another implementation could 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 and data access. Another approach could be to complement a single port joint 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

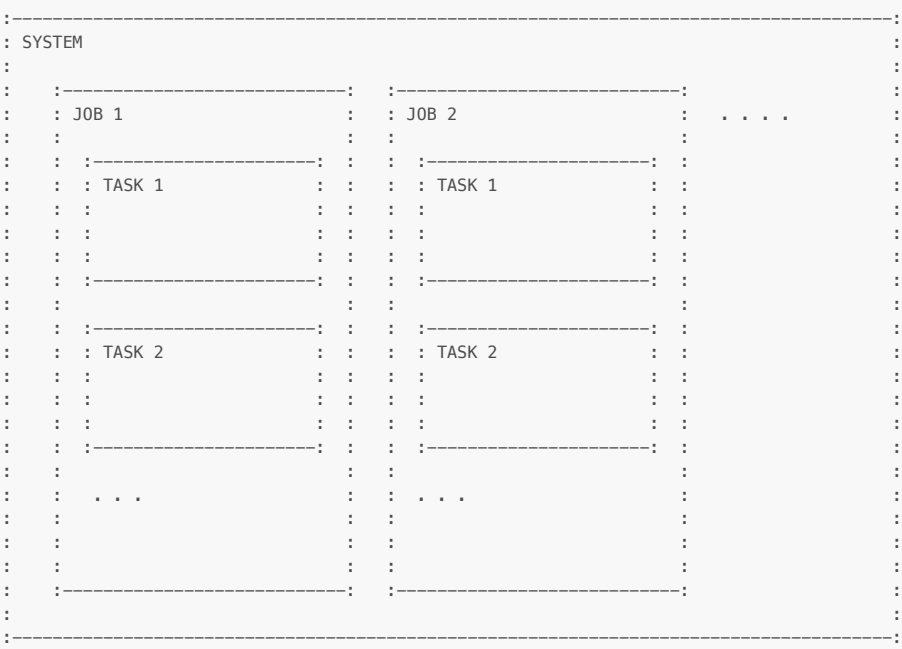
TLBs are explicitly managed by software. The ITLB and PTLB instruction allow for insertion and deletion of TLB entries. The insert into TLB instructions perform the insertion in two parts. The first instruction puts the address related information into a TLB entry but marks the entry not valid yet. The second insert into TLB instruction will enter access right related information and marks the entry valid.

VCPU-32 Runtime Environment

No CPU architecture with an idea of a runtime environment. Although the instruction set is generic enough 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 first present a high level system model, the register convention, execution model, calling convention and overall runtime model.

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 the 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.

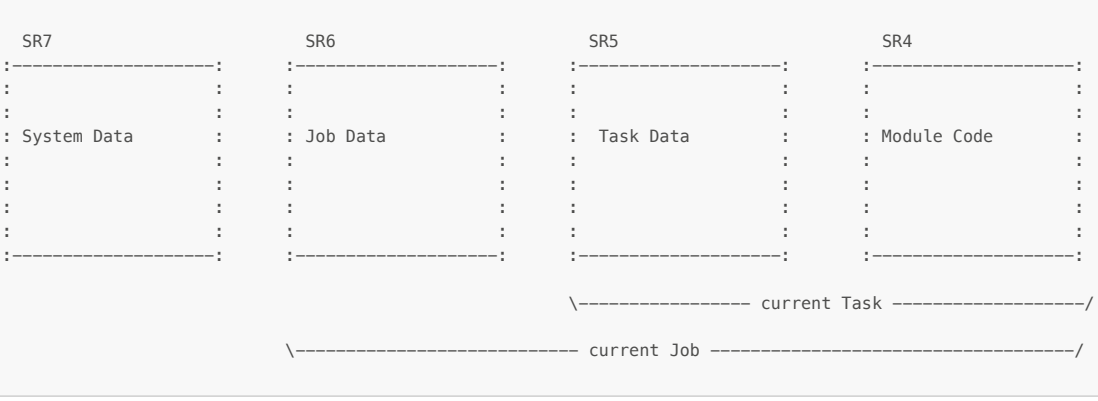


A running task accesses the data areas through an address formed by the dedicated segment registers and the offset into this segment. The next section will describe how this high level system model will be mapped to the registers of the CPU.

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 index register and a segment register form a virtual address. To avoid juggling on every access both a segment and an index register, the segment register selection field in the respective instructions will either implicitly pick one of the upper four segment registers or specify on of the segment registers 1 to 3. This scheme allows to use in most cases just the offset portion of the virtual address when passing pointers to function and so on. The segment register is implicitly encoded in the upper two bits.

Segment 4 to 7 thus 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 for there on. Segment register 7 is assigned to contain the segment ID of this space. Once set at system startup, its content will never change. 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. 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 the SR6 segment register. This register is set every time the execution changes to a thread of another job. This area can be up to one gByte in size. Finally, in a similar manner, the task local data is pointed to by the segment register 5. This data area contains through local data and the stack.

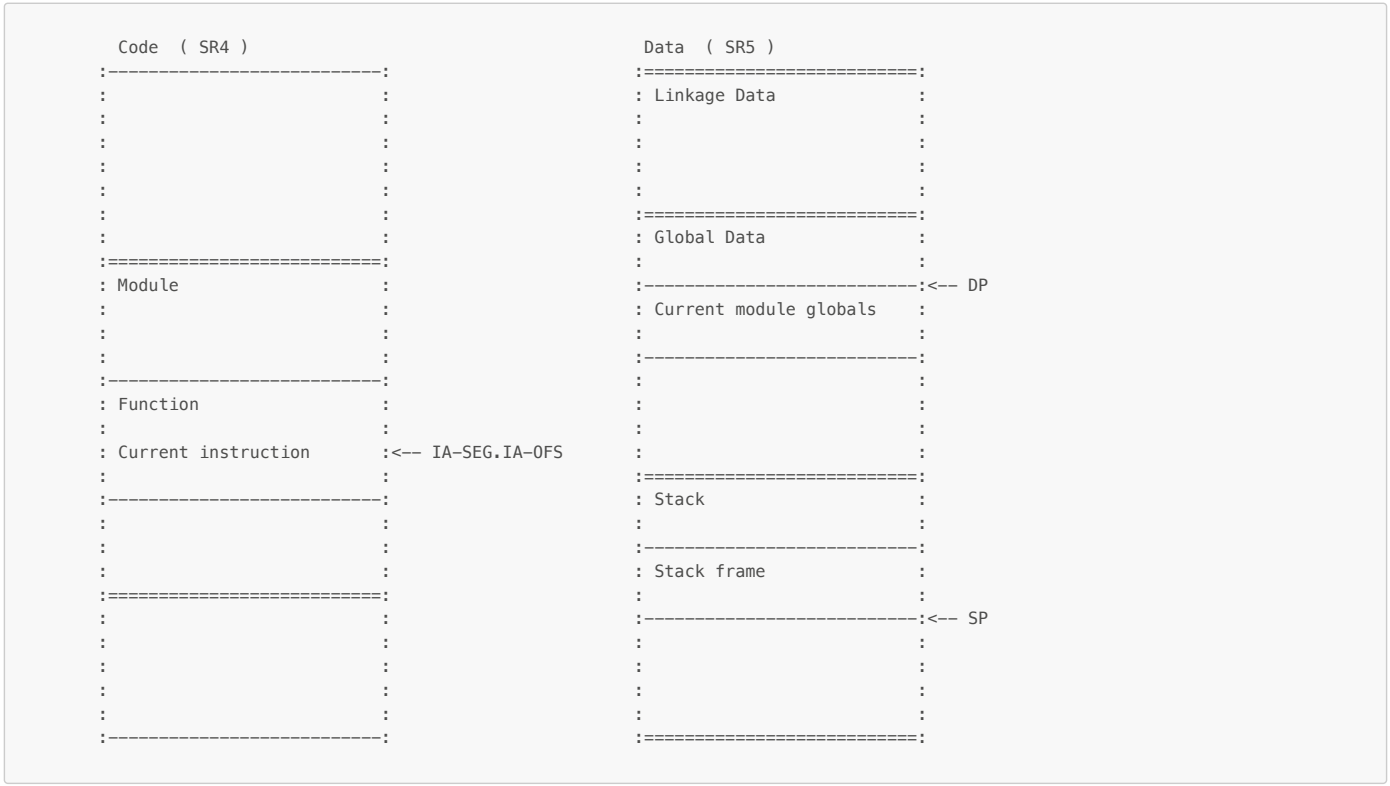


Switching between task will set SR5 to the segment containing the task data and stack. SR6 is set to the job data segment to which the task belongs. Naturally, setting any of these registers is a privileged operation. SR4 will track the current code module so that program literals can be referenced.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

Task Execution

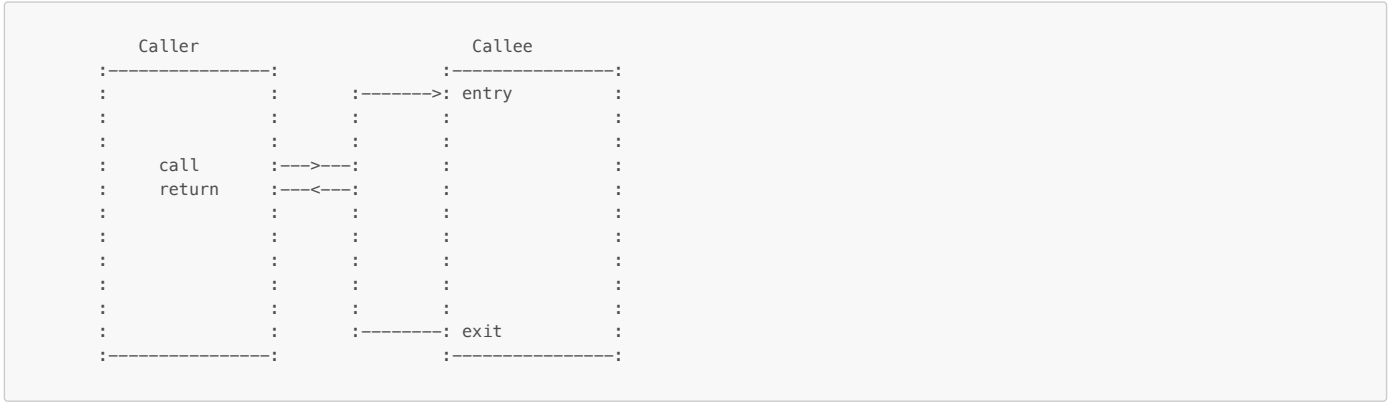
This section provides a high level overview of a running task execution. A running task will execute instruction pointed to by the IA-SEG.IA-OFS virtual address. Code is organized as a collection of modules, which themselves are a collection of functions. By runtime convention, the code is in the SR4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

The SR5 quadrant contains the global data of the current module, 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.



Calling Conventions and Naming

The runtime description uses a basic model to describe the calling convention. The calling procedure is called the **caller**, the target procedure is called the **callee**.

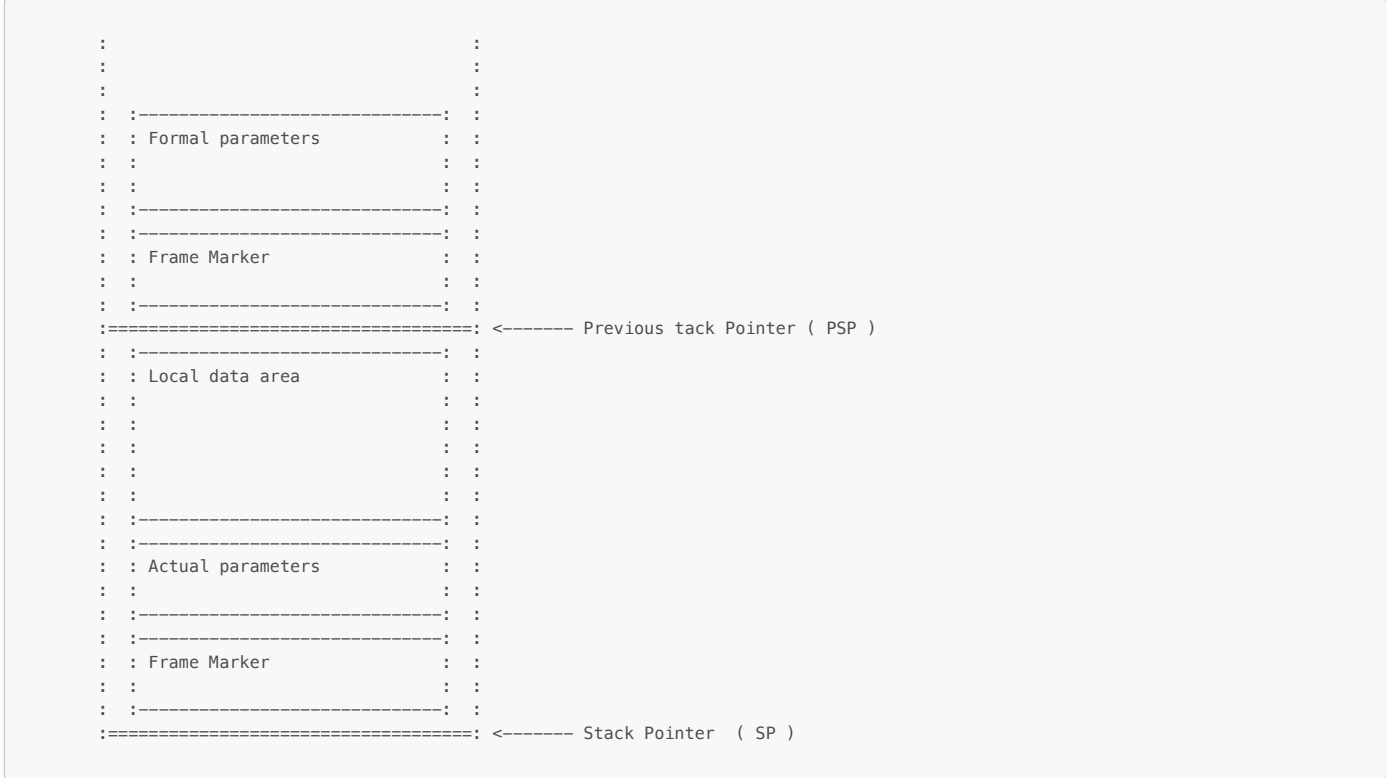


"Call" transfers control to the callee. **"Entry"** is the point of entry to the callee. **"Exit"** is the point of exit from the caller. **"Return"** is the return point of program control to the caller. In addition, there could be pieces of code that enable more complex calling mechanisms such as inter module calls. They 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 procedures call other procedures. Upon entry, a procedure allocates a stack frame for 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 procedures. A leaf procedure is one that does not make any further calls and perhaps need not to allocate a stack frame. The stack are will grow toward higher memory addresses. The

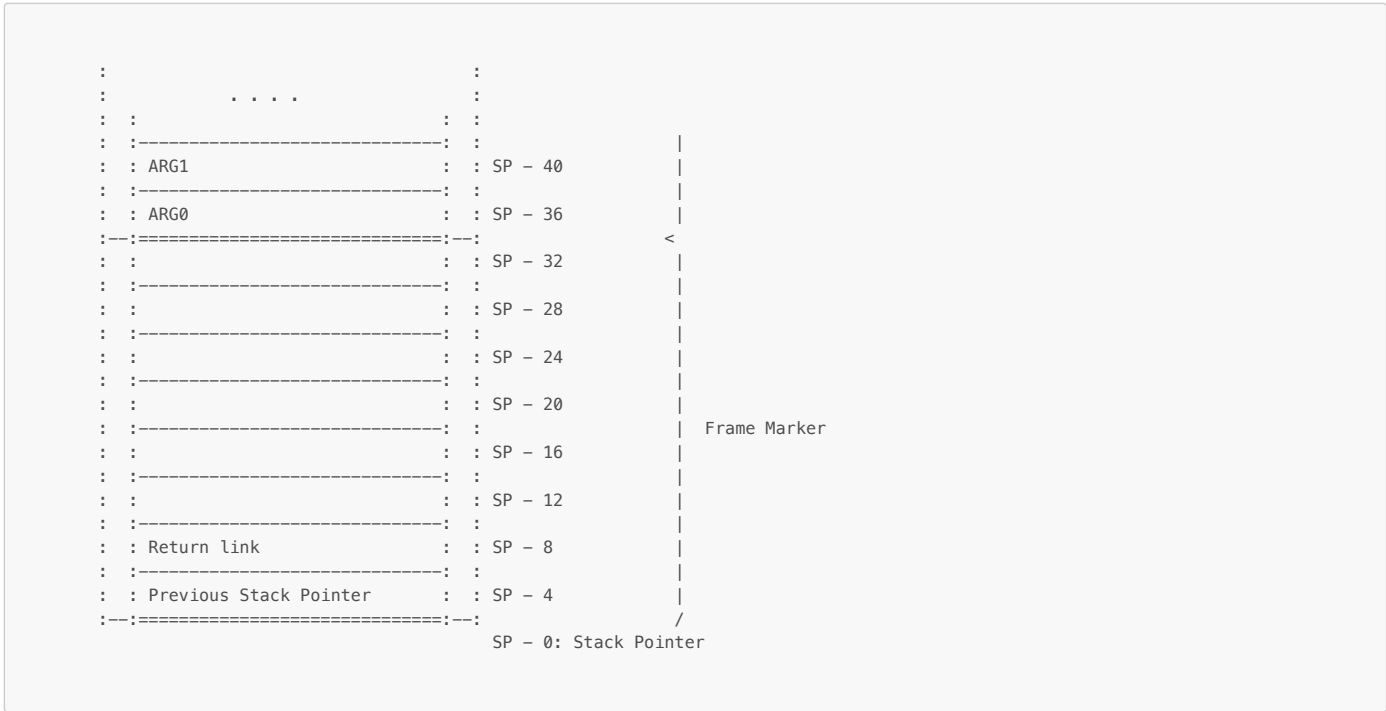
data stack pointer will always points to the location where the next frame will be allocated. Addressing the current frame and the frame marker and parameter area of the previous is SP minus an offset.T



A stack frame is the associated data area for a procedure. 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 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. The parameter area contains the formal arguments of the caller when making the call and potential return values when the caller returns.

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 to store that kind of data during the calling sequence.



Register Partitioning

To the programmer general registers and segment registers are the primary registers to work with. Although the registers have 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
0	Zero
1	Scratch
2	Callee Save
3	Callee Save
4	Callee Save
5	Callee Save
6	Callee Save
7	Callee Save
8	Caller Save, ARG3, RET3
9	Caller Save, ARG3, RET2
10	Caller Save, ARG2, RET1
11	Caller Save, ARG1, RET0
12	Caller Save
13	Global Data (DP)
14	Return link (RL)
15	Stack pointer (SP)

	Segment registers
0	Return Segment Link
1	Caller save, general purpose
2	Caller save, general purpose
3	Callee Save, general purpose
4	Caller Save, general purpose
5	Task Segment (stack)
6	Job Segment
7	System Segment

Some VCPU-32 instructions have a registers as an implicit target. For example, the ADDIL instruction uses general register one as implicit target. The BLE instructions saves the return link implicitly in GR1 and SR0. The caller can rely on that certain register are preserved across the calls. In addition to the callee save registers Gr2..GR7, the SP value, the RL and the DP value as well as SR4 to SR 7 are preserved across a call. In addition to the caller save segment registers, the processor status registers as well as any registers set by privileged code is 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. In case of a full virtual address 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 return an integer value would return it in the register RET0.

Local Calls

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 procedure, 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 pf callee save registers and being a leaf routine would not need a stack frame at all. there is great flexibility go omit steps not needed in the particular situation.

The following assembly code snippets shows examples of the 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:

Caller "func1"                              Callee "func2"

LDI ARG0, 500
LDI ARG1, 300
BL  RL, "func2"         ---------------->   ADD RET0, ARG0, 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" {                          int func2( int a, b ) {                int func3( ) { }
   func2( 500, 300 );                      int local1, local2;
}                                       }                                          func3( a, b );

Assembly:

"func1"                                 "func2"                                "func3"

LDI ARG0, 500
LDI ARG1, 300
BL  RL, "func1"      --------------->   ST -8(SP), RL
                                        LDO SP, 56(SP)

                                        ...

                                        BL RL, "func2"      --------------->   ...

                                        ...                 <---------------   BV RL

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

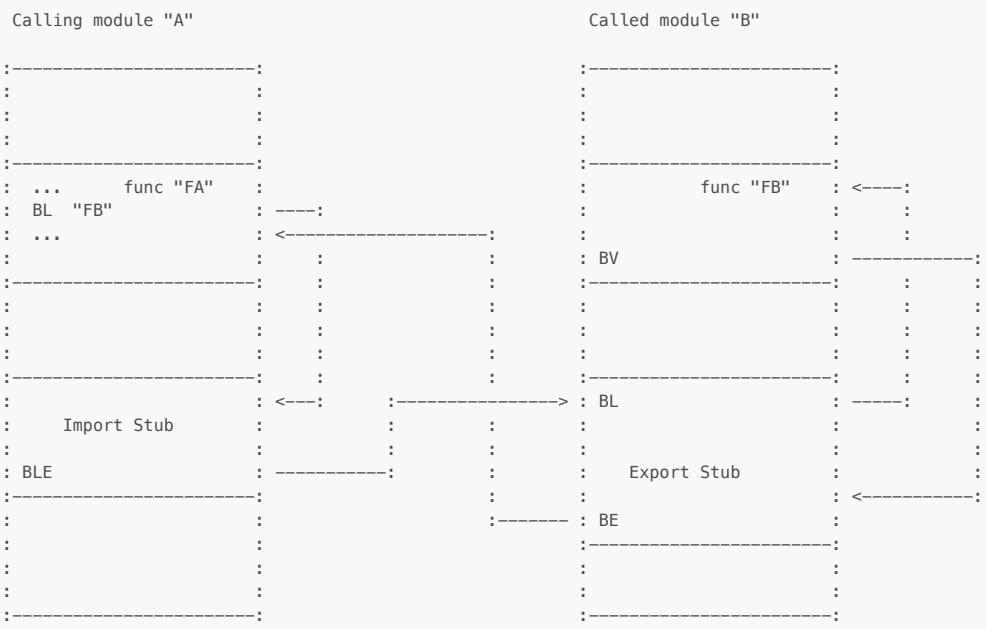
The caller "func1" will produce its arguments and stores them in ARG0 and ARG1, as in the example before. If there were more than four arguments, they would have been stored with a ST instruction in their stack frame location. The callee "func2" is this time a procedure that calls another procedure and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

External Calls

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. Procedure calls are thus always local calls. Likewise a called function will return via a local branch. When a call is crossing a module boundary, small pieces of code need to be added which perform the additional work. These code sequences are called **stubs**. There are import stubs that are added for a procedure that makes an external call and export stubs for procedure that are a target for an external call.



The diagram shows "FA" in module "A" calling "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".

// import stub - uses stack frame locations for housekeeping

// export stub - uses stack frame locations for housekeeping

// linkage table - info how to get to the external module and procedure

Privilege level changes

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

// GATE instruction and external calls

Traps and Interrupt handling

Debug

// essentially a trap

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

// ??? note what part do we architect ?

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 IO 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

A pipelined CPU

The VCPU-32 instruction set and runtime architecture has been designed to take in consideration the effects of stalling and flushing a CPU pipeline. In general, such operations are in terms of performance costly and should be avoided. Also, access data memory twice or any indirection level of data access will also affect the pipeline performance i a negative way. This chapter presents a simple pipeline reference model for a VCPU-32 implementation.

The VCPU-32 reference implementation uses a three stage pipeline model. There 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 any 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.

Note that this is perhaps one of many ways to implement a pipeline. The three major stages could also be further divided internally. For example, the fetch and decode stage could consist of two sub stages. Likewise, the memory access stages could be divided into an address calculation sub-stage and the actual data access. Dividing into three stages however 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.

Notes

None so far.

References

None so far.