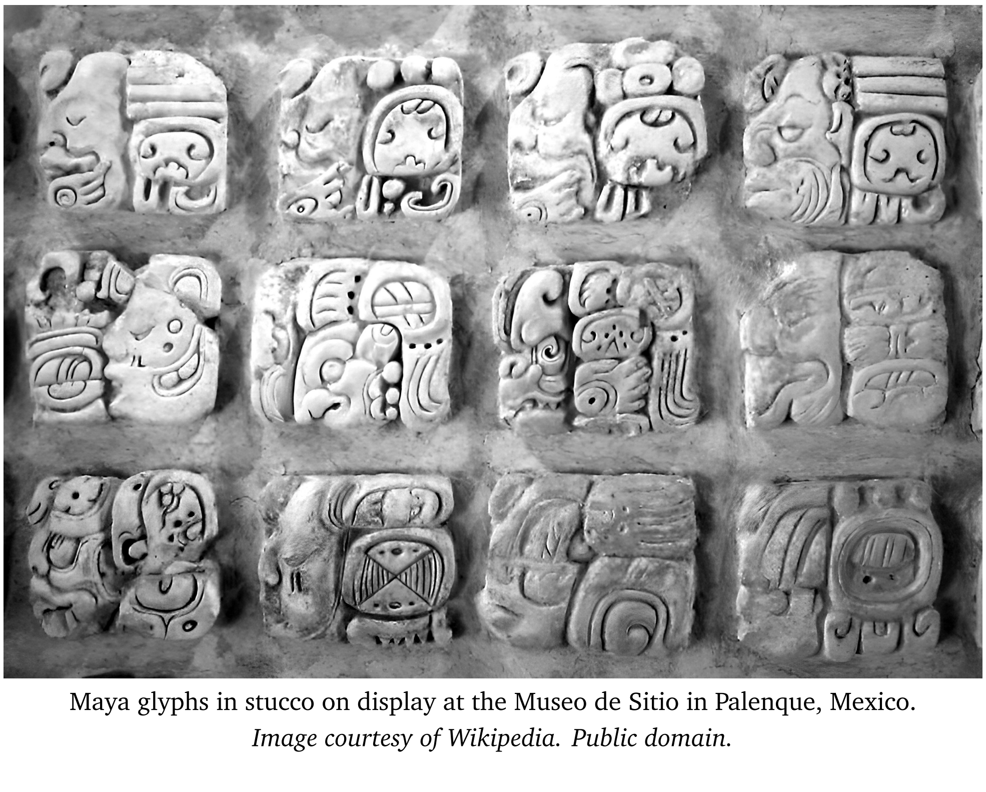
GLYPH-X

a super regular RISC architecture

January 22, 2026 – v0.6.0-current

Michael Clark
Independent Researcher



Maya glyphs in stucco on display at the Museo de Sitio in Palenque, Mexico.
Image courtesy of Wikipedia. Public domain.

to my mentors, whose guidance and wisdom made this work possible.

"*Perfection is achieved,*
not when there is nothing more to add,
but when there is nothing left to take away."

— Antoine de Saint-Exupéry

Contents

Preface

this document introduces a RISC architecture designed with simplicity as its dominating principle, along with several features not yet present in mainstream RISC architectures:

- a compression technique for split instruction and constant streams.
- a variable-length instruction format designed for vectorized decoding.
- an efficient frontier between hardware synthesis and software translation.
- a modern approach to virtualization, memory management, and capabilities.

the document begins with a general introduction to RISC architectures, including principal elements such as load-store operation, memory, registers, instructions, and constants. it then presents an overview of the proposed architecture, followed by sections on program streams, the variable-length instruction format, the instruction set, its assembly syntax, and the programming language calling convention.

this document does not simply present an architecture reimplementing existing ideas; rather, it introduces a novel architecture featuring an instruction coding scheme optimized for efficient parallel decoding in hardware or software, combined with contemporary practices in instruction semantics, including the use of single-bit predicates for compare, divide, branch, plus arithmetic instructions with carry, to achieve an optimal mapping to vectors.

in addition to the core instruction set, the architecture also defines extensions for capabilities, virtual domains, software-defined memory management and environment. the capability extension adds fine-grained page-based permissions and role based access control. the domain extension adds virtualization for trap routing, page translation, and context switching. the machine extension provides for custom address translation schemes. finally, the environment extension adds controls for power and reset.

this work is intended for computer architects, systems programmers, students of computing machinery, and anyone interested in the evolution of RISC design. by the end of the document, readers should be able to reason about computer programs at the instruction level and understand the logic behind the architecture's design choices—with enough detail to implement the architecture in hardware or software.

1. Architecture

1.1. Introduction

this section gives a brief introduction to RISC architectures.

A RISC[1] machine is a type of general-purpose computer with the characteristic that it has a reduced set of instructions in contrast to a CISC[2] machine. A RISC machine is Turing complete meaning it can perform any computation that a Turing machine can, given enough time and memory. a Turing machine [12] is a theoretical model of computation.

a RISC machine has a set of instructions which comprise basic operations such as: *load-from-memory*, *store-to-memory*, *add*, *subtract*, *compare*, plus *conditional branch* and *unconditional branch* instructions et cetera; which one can imagine as a list of instructions on a paper tape. each instruction has an *opcode*, which is a unique binary pattern that identifies the *operation*, plus several *operands*, which are arguments to the instruction.

```
    register-0 = load-from-memory at tape-address-0
    register-1 = load-from-memory at tape-address-1
    register-2 = add register-0 and register-1
                 store-to-memory register-2 at tape-address-2
```

some instructions have operands that point to values inside of *registers* in a *register-file* which is like a close filing cabinet containing cards with numbers on them, and some of these numbers are addresses that point to values in *main-memory* which is like a larger but slower filing cabinet. some of these values are *immediate* values which are small numbers listed inside of the instructions on the paper tape.

the paper tape is just a way conceptualize a list of instructions stored in *main-memory*. there is a special register called *PC* short for *program counter*, which points to the current position on the tape. after each instruction executes the tape is advanced to the next instruction and the *program counter* is incremented, until it encounters a *branch* instruction which causes it to move forwards or backwards to a different position on the tape. branch instructions can be *conditional* or *unconditional*. conditional branches are selectively executed based on the results of a comparison instruction.

[1]Reduced Instruction Set Computer
[2]Complex Instruction Set Computer

1.1.1. load-store

a *load-store* architecture [4] is a way to characterize RISC architectures where most instructions have simple operands that point to values held in registers, plus *load* and *store* instructions to retrieve and commit values to main memory. a *load-store* architecture alleviates the need to add complex addressing modes, plus intput-output to peripherals and secondary storage use *MMIO*[3] to avoid needing special *Input/Output* instructions.

1.1.2. registers

registers are temporary storage used to fill input and output operands for the *ALU*[4] before and after execution of instructions. registers are organized as a word-addressable store where each register number refers to $XLEN \in \{64, 128\}$ bits of data. *XLEN* is a parameter that specifiess the width of registers in bits.

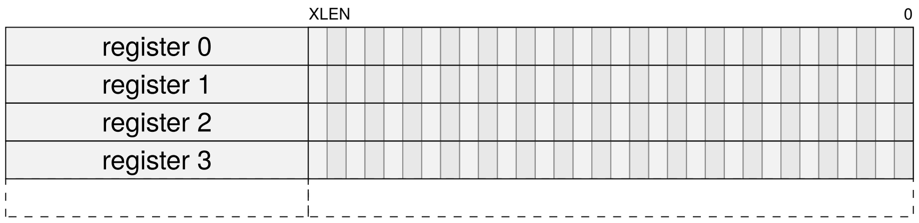


Figure 1.1.: organization of register storage.

1.1.3. memory

main memory is primary storage which in modern computers is most likely *DRAM*[5]. main memory is organized as a byte-addressable store where each address refers to a byte which is 8-bits of data. *ALEN* is a parameter that specifies the width of addresses in bits.

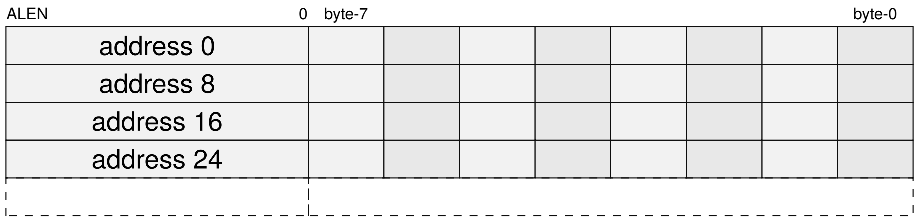


Figure 1.2.: organization of main-memory storage.

[3]*MMIO* - Memory-mapped I/O.
[4]*ALU* – Arithmetic logic unit.
[5]*DRAM* – Dynamic random-access memory.

1.1.4. instructions

instruction memory is a memory region dedicated to program instructions. it is organized as a word-addressable store, similar to main memory, but is read-only and reserved exclusively for instructions. this configuration is typically referred to as a Harvard architecture [1], in contrast to a Von Neumann architecture [9], which uses a shared memory region for both program instructions and data.

instruction words contain packets of 16-bits which are composed of size, opcode, and operand fields. instruction packets can be appended together to form larger instruction words with larger opcode and operand fields. section 1.5 provides details on the instruction template and instruction forms.

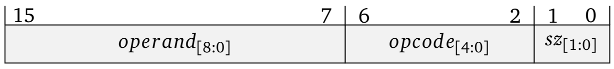


Figure 1.3.: 16-bit instruction containing one packet.

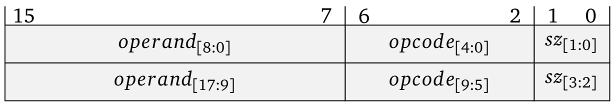


Figure 1.4.: 32-bit instruction containing two packets.

instruction words are packed together into instruction blocks containing instruction size information that is used to fuse them together to compose variable-length instructions [8]. instruction words can begin on any 16-bit aligned address. instruction memory is addressed with displacements from the *program counter* called *pc-relative* addresses.

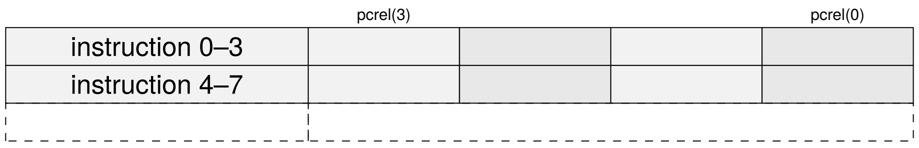


Figure 1.5.: organization of instruction-memory with 16-bit instructions.

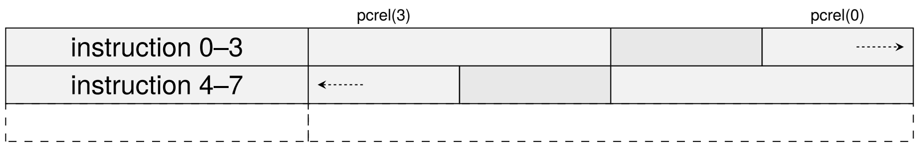


Figure 1.6.: organization of instruction-memory with 16-bit and 32-bit instructions.

1.1.5. constants

constant memory [11] is a memory region dedicated to constants. it is organized as a word-addressable store like main-memory, but is read-only and restricted to constants. instructions can refer to constants using *ib-relative* addresses inside instruction operands. section 1.2 provides more details on the split instruction and constant streams.

constant memory is addressed with displacements from the *immediate base* register called *ib-relative* addresses which are stored inside instruction operand slots, are sized, scaled and aligned addresses computed relative to the *immediate base* register, which is like a program counter for constants. these diagrams show $ib8(n)$ through $ib64(n)$ *ib-relative* addresses for access to 8-bit through to 64-bit constants respectively: *note: ib-relative addresses alias a single constant storage space containing all types.*

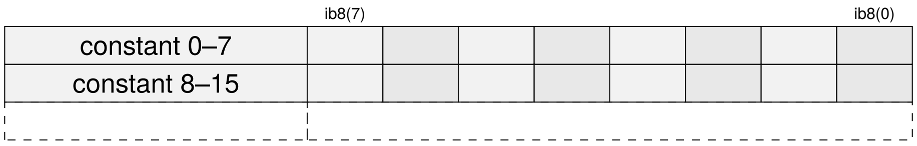


Figure 1.7.: organization of *ib8* constant-memory storage.

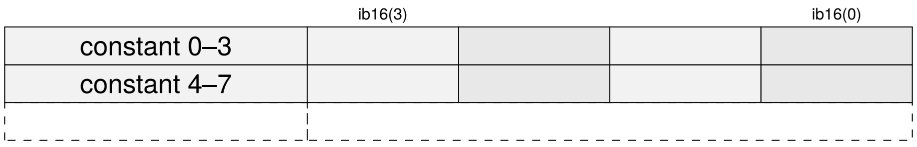


Figure 1.8.: organization of *ib16* constant-memory storage.

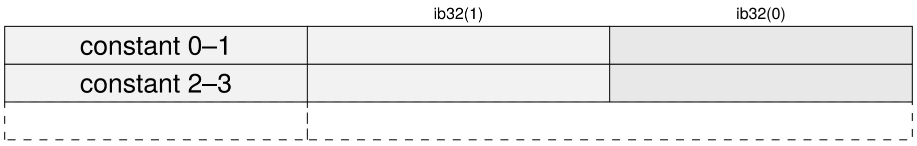


Figure 1.9.: organization of *ib32* constant-memory storage.

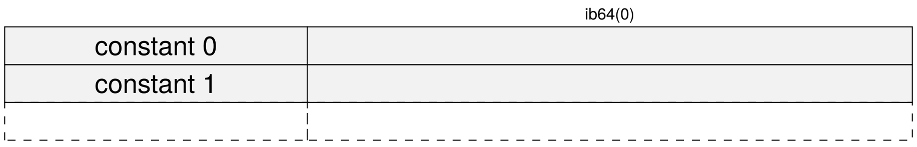


Figure 1.10.: organization of *ib64* constant-memory storage.

1.2. Program streams

glyph seperates the stored programs into two streams, one with instructions and one with constants [11]. the instruction stream is addressed with the *program counter (pc)* and the constant stream is addressed with the *immediate base (ib)* register. this could be referred to as split program stream Harvard architecture.

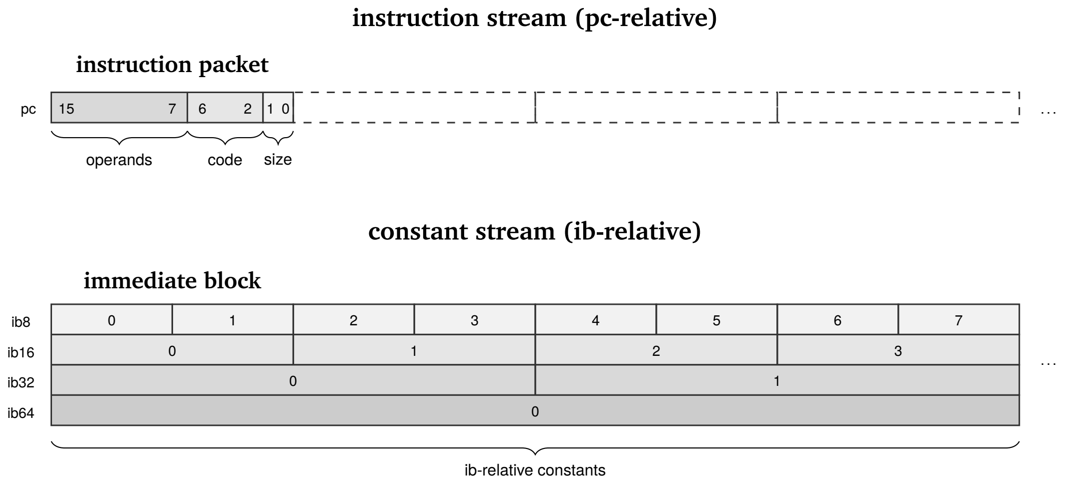


Figure 1.11.: program counter and immediate base register.

instruction blocks are aligned memory blocks addressed by the *program counter*. instruction memory is addressed with *pc-relative* addresses in instruction immediate values or indirectly with constants accessed using *ib-relative* addresses stored inside instruction operand slots.

immediate blocks are aligned memory blocks addressed by the *immediate base* register. constant memory is addressed with *ib-relative* addresses stored inside instruction operand slots, which are sized, scaled, and aligned relative to the *immediate base* register.

the constant stream branches independently using the *immediate-block-jump* instruction to update the *immediate base* register, or together with the *program counter* in procedure call and return instructions that branch instruction and constants at the same time; *jump-and-link-immediate-block* and *jump-to-link-immediate-block* add and subtract address vectors to *(pc,ib)*. the *pack-indirect* instruction allows absolute *(pc,ib)* addresses to be packed into an address vector for indirect calls.

the instruction forms use bonded operand slots for immediate operands and operand bits do not overlap opcode bits. the use of immediate blocks means large immediate constants can all be accessed using short references encoded inside of operand slots for instructions that use an immediate block relative addressing mode.

1.3. Core concepts

glyph is a super regular RISC architecture that encodes constants in a secondary stream accessed via an *immediate base* register that points at immediate blocks containing constants accessed via a constant address mode. the *immediate base* register branches like the *program counter*, and procedure calls and returns set and restore *(pc,ib)* together.

glyph uses relative address vectors in its link register which is different to typical RISC architectures. glyph does this so that the branch instructions can fit *(pc,ib)* into the a single link register for compatibility with traditional RISC architectures. glyph achieves this by packing two relative *(pc,ib)* displacements into a relative address vector[6].

immediate blocks can be switched using the immediate block branch instruction. immediate blocks, unlike typical RISC architectures, mean that most relocations are word sized like CISC architectures, and can use C-style structure packing and alignment rules.

this list outlines some differentiating elements of the super regular RISC architecture:

- variable length instruction format supporting 16, 32, and 64-bit instructions.
- 1-bit predicate for compare, branch, add with carry, and subtract with borrow.
- 16-bit compressed instruction packets that can access 8 registers.
- *(pc,ib)* is a *program counter* and *immediate base* register address vector.
- link register contains a packed relative *(pc,ib)* address vector to function entry.
- `ibj` — *immediate-block-jump* adds a relative address to the *immediate base* register.
- `movw` — *move-word-immediate-block* uses a displacement to access a constant.
- `jalib` — *jump-and-link-immediate-block* or *link-function-call* copies the address vector into the link register and adds constants to *(pc,ib)*. it is used to branch the *program counter* and *immediate base* register at the same time for calling procedures.
- `jtlib` — *jump-to-link-immediate-block* or *link-function-return* subtracts link vector from and adds constants to *(pc,ib)*. it is used to branch the *program counter* and the *immediate base* register at the same time for returning from called procedures.
- `pin` — *pack-indirect* packs two absolute addresses as relative address vector from *(pc,ib)* and is used for calling absolute addresses such as virtual functions.

the `pin` and `link` instructions form a modular arithmetic ring of relative address vectors, due to their use of relative addresses. relative address vectors can be encrypted, decrypted, and authenticated to form a verifiable chain of addresses for control flow integrity.

[6] the architecture defines two parameters: *ALEN* and *XLEN*, which respectively to refer to width of addresses and width of general purpose registers in bits. when $XLEN \geq ALEN \times 2$ it is possible to pack absolute addresses instead of relative addresses, as would be the case where *ALEN=64* and *XLEN=128*.

1.4. Core parameters

glyph has serveral architectural parameters that define the dimensions of an instance of the architecture to allow the archiecture to be instantiated in several different configurations. these configurations are grouped into architectural profiles.

1.4.1. architectural parameters

glyph defines a unified floating-point and integer scalar register file with 8 or 64 registers and an optional unified floating-point and integer vector register file with 64 registers.

- 16-bit instruction packet can access 8 registers with up to 3 operands.
- 32-bit instruction packet can access 64 registers with up to 3 operands.
- 64-bit instruction packet can access 64 registers with up to 6 operands.

glyph architectural parameters are as follows:

Parameter	Value	Description
SREG	8, 64	number of scalar registers
VREG	–, 64	number of vector registers
ILEN	16, 32, 64	width of largest instruction in bits
ALEN	64	width of arithmetic on scalars in bits
XLEN	64, 128	width of scalar registers in bits
VLEN	512, 4096	width of vector registers in bits
GLEN	128	width of vector lane group in bits

Table 1.1.: architectural parameters

1.4.2. architectural profiles

glyph architectural parameters are linked to the dimensions of the register slots in the variable length instruction packets. the 16-bit packet can be used in a freestanding microcontroller profile. the 32-bit packet contains scalar and packed-SIMD[7] instructions on scalar registers. the 64-bit packet contains packed-SIMD instruction on vector registers.

profile	SREG	VREG	ILEN	ALEN	XLEN	VLEN	GLEN
scalar-min	8	-	16	64	64	-	-
scalar-max	64	-	32	64	128	-	-
vector-min	64	64	64	64	128	512	128
vector-max	64	64	64	64	128	4096	128

Table 1.2.: architectural profiles

[7] SIMD - Single Instruction Multiple Data

1.5. Instruction format

instructions use a variable length format [8] with a 16-bit instruction packet that supports 16, 32, 64, and 128-bit instruction words. the instruction packet uses a *super regular* scheme, whereby successive instruction packets extend the fields in the previous instruction packet [7]. instruction words can begin on any 16-bit aligned address.

1.5.1. instruction sizes

the variable length instruction format has a *2-bit* size field inspired by LEB128 [5] in a fixed position in every instruction packet, to reduce the complexity of size-decoding for variable length instructions[8]. this table shows the size fields for the different instruction sizes:

Instruction size	Size vector
16-bit	`{00}`
32-bit	`{01,11}`
64-bit	`{10,11,11,11}`

Table 1.3.: variable-length instruction size fields

1.5.2. instruction decoding

the instruction size encoding and field extension scheme has been designed to minimize complexity for parallel decoding in hardware and software. the following table shows the instruction width combinations for various width parallel instruction decoders.

Packets	Bits	Bytes	n_{combos}	$\log_2(n)$
1-wide	16-bit	2	1	0
2-wide	32-bit	4	16	4
4-wide	64-bit	8	58	5.86
8-wide	128-bit	16	574	9.16
16-wide	256-bit	32	51904	15.66

Table 1.4.: instruction decode combinations

[8]instructions are considered *well-formed* if the size field of subsequent packets has the value `11`.

1.5.3. instruction forms

the variable length instruction format has a single base format where fields in the template instruction form are extended by successive instruction packets.

the instructions forms are super regular in that operand and opcode bits do not overlap and the number and complexity of the formats is reduced so that vectorized instruction decoding is simple in both hardware and software. the scheme is designed so that *1-bit* of coding space in the larger packet can be used to extend register sizes.

instruction templates

this section details the layout of the variable sized instructions:

15 7	6 2	1 0
$operand_{[8:0]}$	$opcode_{[4:0]}$	$sz_{[1:0]}$

op16
16-bit instruction packet

15 7	6 2	1 0
$operand_{[8:0]}$	$opcode_{[4:0]}$	$sz_{[1:0]}$
$operand_{[17:9]}$	$opcode_{[9:5]}$	$sz_{[3:2]}$

op32
32-bit instruction packet

15 7	6 2	1 0
$operand_{[8:0]}$	$opcode_{[4:0]}$	$sz_{[1:0]}$
$operand_{[17:9]}$	$opcode_{[9:5]}$	$sz_{[3:2]}$
$operand_{[26:18]}$	$opcode_{[14:10]}$	$sz_{[5:4]}$
$operand_{[35:27]}$	$opcode_{[19:15]}$	$sz_{[7:6]}$

op64
64-bit instruction packet

16-bit instruction forms

this section details the layout of the 16-bit instruction forms:

15 7	6 2	1 0
$imm_{[5:0]}$	$opcode_{[4:0]}$	**00**

op16i
16-bit large immediate

15 13	12 7	6 2	1 0
$rc_{[2:0]}$	$imm_{[5:0]}$	$opcode_{[4:0]}$	**00**

op16ri
16-bit one operand with immediate

15 13	12 10	9 7	6 2	1 0
$rc_{[2:0]}$	$rb_{[2:0]}$	$imm_{[2:0]}$	$opcode_{[4:0]}$	**00**

op16rri
16-bit two operand with immediate

15 13	12 10	9 7	6 2	1 0
$rc_{[2:0]}$	$rb_{[2:0]}$	$ra_{[2:0]}$	$opcode_{[4:0]}$	**00**

op16r3
16-bit three operand

32-bit instruction forms

this section details the layout of the 32-bit instruction forms:

15 7	6 2	1 0
$imm_{[8:0]}$	$opcode_{[4:0]}$	**01**
$imm_{[17:9]}$	$opcode_{[9:5]}$	**11**

op32i
32-bit large immediate

15 13	12 7	6 2	1 0
$rc_{[2:0]}$	$imm_{[5:0]}$	$opcode_{[4:0]}$	**01**
$rc_{[5:3]}$	$imm_{[11:6]}$	$opcode_{[9:5]}$	**11**

op32ri
32-bit one operand with immediate

15 13	12 10	9 7	6 2	1 0
$rc_{[2:0]}$	$rb_{[2:0]}$	$imm_{[2:0]}$	$opcode_{[4:0]}$	**01**
$rc_{[5:3]}$	$rb_{[5:3]}$	$imm_{[5:3]}$	$opcode_{[9:5]}$	**11**

op32rri
32-bit two operand with immediate

15 13	12 10	9 7	6 2	1 0
$rc_{[2:0]}$	$rb_{[2:0]}$	$ra_{[2:0]}$	$opcode_{[4:0]}$	**01**
$rc_{[5:3]}$	$rb_{[5:3]}$	$ra_{[5:3]}$	$opcode_{[9:5]}$	**11**

op32r3
32-bit three operand

64-bit instruction forms

this section details the layout of the 64-bit instruction forms:

15 7	6 2	1 0
$imm_{[5:0]}$	$opcode_{[4:0]}$	**10**
$imm_{[11:6]}$	$opcode_{[9:5]}$	**11**
$imm_{[17:12]}$	$opcode_{[14:10]}$	**11**
$imm_{[23:18]}$	$opcode_{[19:15]}$	**11**

op64i
64-bit large immediate

15 13	12 7	6 2	1 0
$rc_{[2:0]}$	$imm_{[5:0]}$	$opcode_{[4:0]}$	**10**
$rc_{[5:3]}$	$imm_{[11:6]}$	$opcode_{[9:5]}$	**11**
$rf_{[8:6]}$	$imm_{[17:12]}$	$opcode_{[14:10]}$	**11**
$rf_{[11:9]}$	$imm_{[23:18]}$	$opcode_{[19:15]}$	**11**

op64rri
64-bit two operand with immediate

15 13	12 10	9 7	6 2	1 0
$rc_{[2:0]}$	$rb_{[2:0]}$	$imm_{[2:0]}$	$opcode_{[4:0]}$	**10**
$rc_{[5:3]}$	$rb_{[5:3]}$	$imm_{[5:3]}$	$opcode_{[9:5]}$	**11**
$rf_{[8:6]}$	$re_{[8:6]}$	$imm_{[8:6]}$	$opcode_{[14:10]}$	**11**
$rf_{[11:9]}$	$re_{[11:9]}$	$imm_{[11:9]}$	$opcode_{[19:15]}$	**11**

op64r4i
64-bit four operand with immediate

15 13	12 10	9 7	6 2	1 0
$rc_{[2:0]}$	$rb_{[2:0]}$	$ra_{[2:0]}$	$opcode_{[4:0]}$	**10**
$rc_{[5:3]}$	$rb_{[5:3]}$	$ra_{[5:3]}$	$opcode_{[9:5]}$	**11**
$rf_{[8:6]}$	$re_{[8:6]}$	$rd_{[8:6]}$	$opcode_{[14:10]}$	**11**
$rf_{[11:9]}$	$re_{[11:9]}$	$rd_{[11:9]}$	$opcode_{[19:15]}$	**11**

op64r6
64-bit six operand

1.6. Register file

glyph has register state comprised of a predicate register, program counter, and immediate base registers, a unified floating-point and integer scalar register file, and an optional unified floating-point and integer vector register file, each with an extensible number of registers.

1.6.1. template architectural profile

the register state accessible by the template architectural profile is comprised of:

- 1-bit predicate register (*flag*).
- ALEN-bit *program counter* register (aligned to 2 bytes).
- ALEN-bit *immediate base* register (aligned to 64 bytes).
- SREG × XLEN-bit scalar registers (r0 — r*N*).
- VREG × VLEN-bit vector registers (v0 — v*N*).

the following diagram shows the template architectural profile register state:

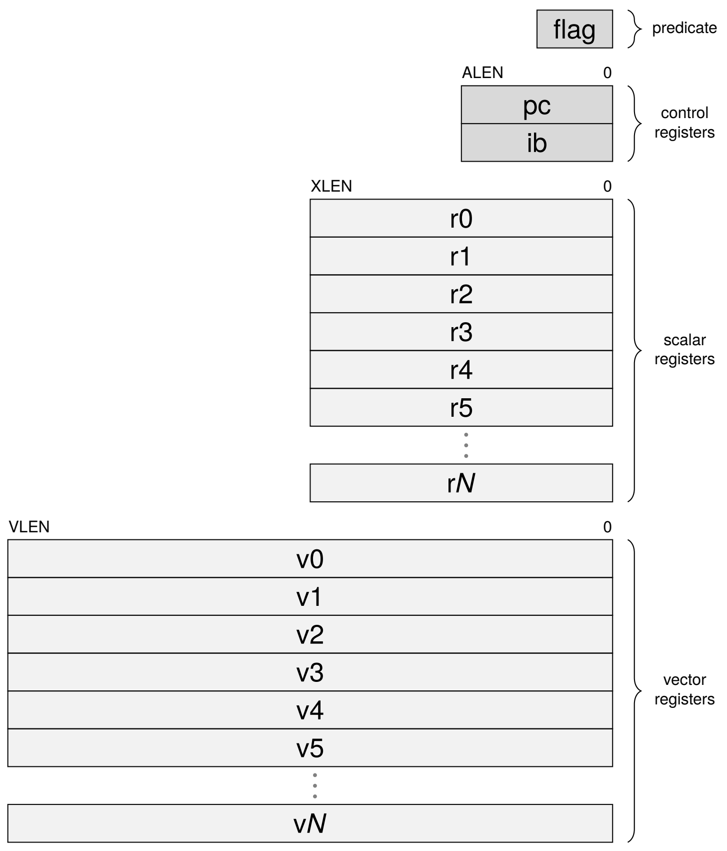


Figure 1.12.: template architectural profile register state.

1.6.2. *scalar-min* architectural profile

the register state accessible by the *scalar-min* architectural profile is comprised of:

- 1-bit predicate register (*flag*).
- 64-bit *program counter* register (aligned to 2 bytes).
- 64-bit *immediate base* register (aligned to 64 bytes).
- 8 $\times$ 64-bit scalar registers (r0 — r7).

the following diagram shows the *scalar-min* architectural profile register state:

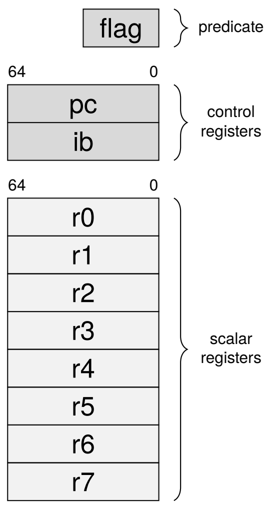


Figure 1.13.: *scalar-min* architectural profile register state.

when $XLEN < ALEN \times 2$ it is not possible to pack absolute addresses in the link register, so the `PIN` and `LINK` instructions save relative addresses in the link register. this increases latency in branch address calculation compared to absolute addresses.

1.6.3. *scalar-max* architectural profile

the register state accessible by the *scalar-max* architectural profile is comprised of:

- 1-bit predicate register (*flag*).
- 64-bit *program counter* register (aligned to 2 bytes).
- 64-bit *immediate base* register (aligned to 64 bytes).
- 64 × 128-bit scalar registers (r0 — r63).

the following diagram shows the *scalar-max* architectural profile register state:

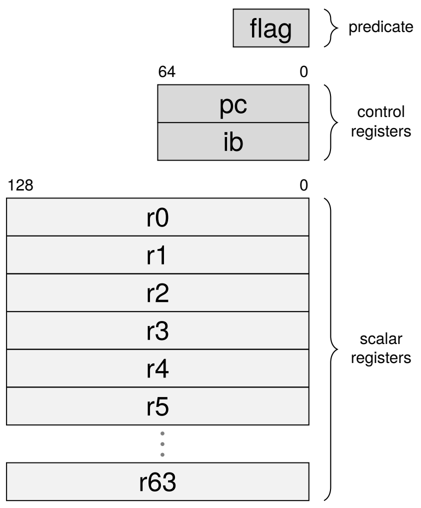


Figure 1.14.: *scalar-max* architectural profile register state.

when $XLEN \geq ALEN \times 2$ it is possible to pack absolute addresses in the link register, so the `PIN` and `LINK` instructions save abosulte addresses in the link register instead. this reduces latency in branch address calculation compared to relative addresses.

1.6.4. *vector-min* architectural profile

the register state accessible by the *vector-min* architectural profile is comprised of:

- 1-bit predicate register (*flag*).
- 64-bit *program counter* register (aligned to 2 bytes).
- 64-bit *immediate base* register (aligned to 64 bytes).
- 64 × 128-bit scalar registers (r0 — r63).
- 64 × 512-bit vector registers (v0 — v63).

the following diagram shows the *vector-min* architectural profile register state:

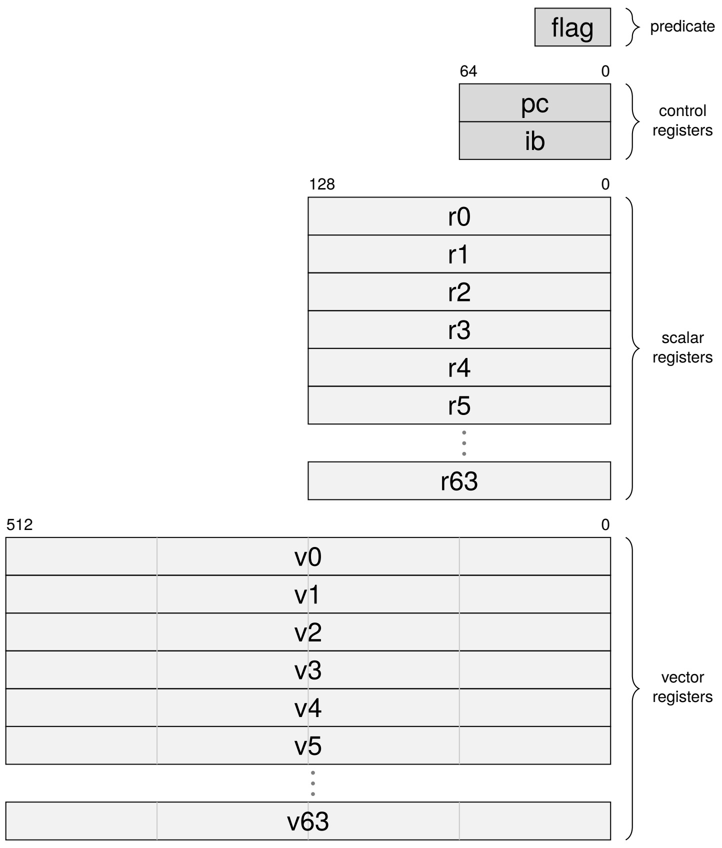


Figure 1.15.: *vector-min* architectural profile register state.

1.6.5. *vector-max* architectural profile

the register state accessible by the *vector-max* architectural profile is comprised of:

- 1-bit predicate register (*flag*).
- 64-bit *program counter* register (aligned to 2 bytes).
- 64-bit *immediate base* register (aligned to 64 bytes).
- 64 × 128-bit scalar registers (r0 — r63).
- 64 × 4096-bit vector registers (v0 — v63).

the following diagram shows the *vector-max* architectural profile register state:

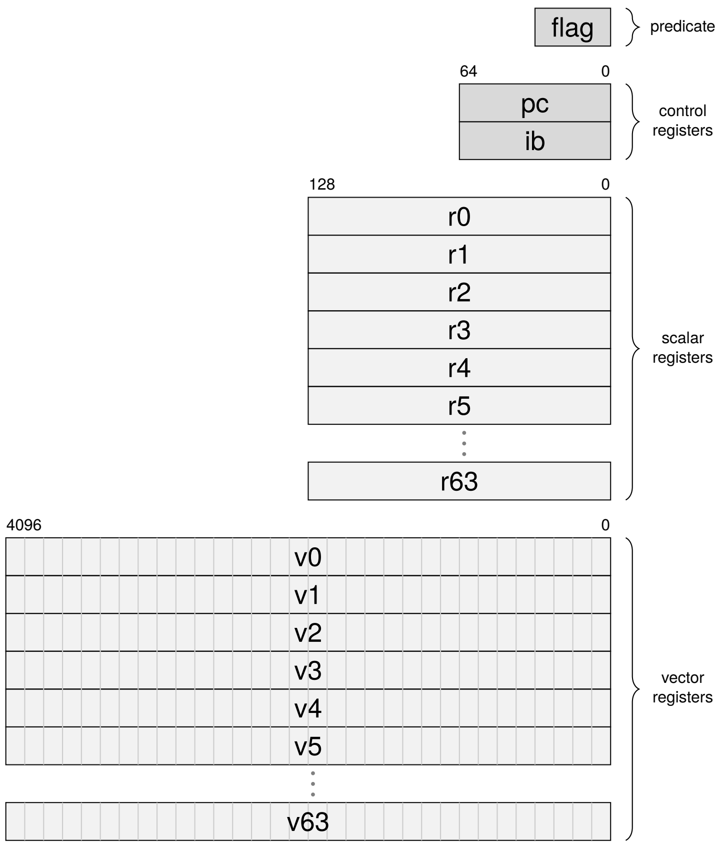


Figure 1.16.: *vector-max* architectural profile register state.

1.7. Example pipeline

an illustrative micro-architecture is proposed based on the classic 5-stage RISC micro-architecture [10] with the addition of an *operand fetch* stage and a *constant memory* port. this revised 6-stage micro-architecture is composed of the following pipeline stages:

- IF — *instruction fetch*: reads instructions from memory into a fetch buffer.
- ID — *instruction decode*: decodes instruction length, opcode, and operands.
- OF — *operand fetch*: reads operands from register file and constant memory.
- EX — *execute*: performs logical operations or arithmetic on the operands.
- MA — *memory access*: loads data from or stores data to memory.
- WB — *writeback*: writes results back to the register file.

a simplified micro-architecture using those pipeline stages might look like this: this example omits hazard detection and forwarding logic for the sake of simplicity.

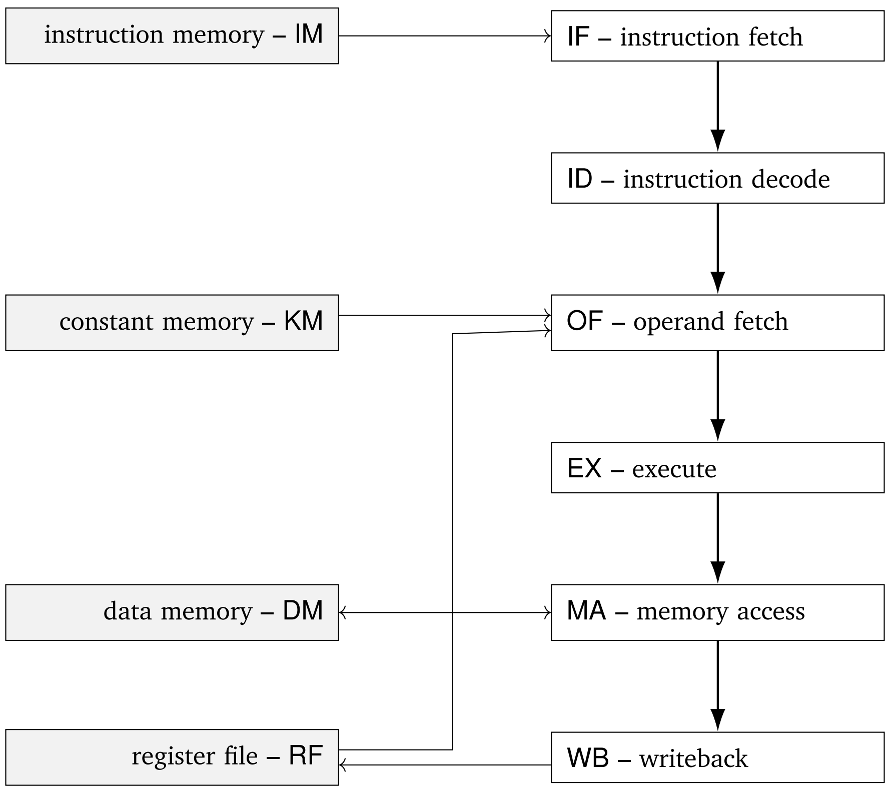


Figure 1.17.: sample 6-stage micro-architecture with support for constant memory.

2. System

2.1. User-level registers

the architecture provides several user-level status and control registers for floating-point status and control, clock time, clock frequency, thread address and domain.

user-level status and control registers can be read and written using the `sysread`, `syswrite`, `sysset`, and `sysclear` instructions. user-level status and control registers are available in contexts where their associated bit is enabled in `scontrol`.

user-level registers

the following table lists the user-level registers:

no.	**name**	**description**
floating-point unit registers		
`0x00`	fpstatus	floating-point status
`0x01`	fpcontrol	floating-point control
clock and frequency registers		
`0x02`	ctime	clock time
`0x03`	cfreq	clock frequency
thread address registers		
`0x04`	addr	user thread address
`0x05`	domain	user thread domain

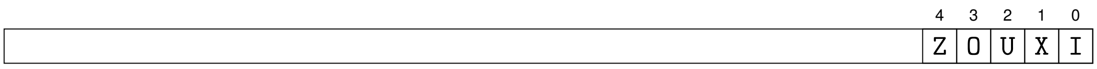
Table 2.1.: user-level registers

2.1.1. floating-point unit registers

the section describes the user-level floating-point unit registers.

floating-point status (`fpstatus`)

`fpstatus` is a read-only register containing status information for floating-point operations. the `Z`, `O`, `U`, `X`, and `I` fields contain accrued floating-point exceptions. this register is visible if `scontrol.F` is enabled.

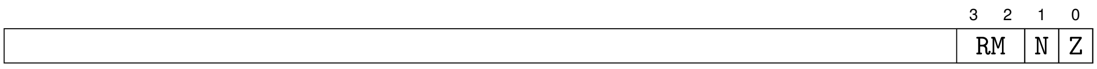


no.	name	description
`1 << 0`	I	invalid operation
`1 << 1`	X	inexact result
`1 << 2`	U	numeric underflow
`1 << 3`	O	numeric overflow
`1 << 4`	Z	divide by zero

Table 2.2.: floating-point status flags

floating-point control (`fpcontrol`)

`fpcontrol` is a read-write register containing control information for floating-point operations. the `RM` field contains the current floating-point rounding mode. the `Z` field enables *flush-to-zero* and *denormals-are-zero*. the `N` field enables *not-strict-ieee754*, allowing *fused-multiply-add* and other optimizations that may affect precision. this register is visible if `scontrol.F` is enabled.



no.	name	description
`1 << 0`	Z	*flush-to-zero* and *denormals-are-zero*
`1 << 1`	N	*not-strict-ieee754*

Table 2.3.: floating-point control flags

no.	name	description
0	RN	round to nearest (even)
1	RD	round down (towards $-\infty$)
2	RU	round up (towards $+\infty$)
3	RZ	round towards zero (truncate)

Table 2.4.: floating-point control round mode field

2.1.2. clock and frequency registers

the section describes the user-level clock and frequency registers.

clock time (`ctime`)

`ctime` is a read-only register containing the wall-clock tick counter since power on in clock tick units denoted by `cfreq`. this register is visible if `scontrol.T` is enabled.

63 … 0
clock time

clock frequency (`cfreq`)

`cfreq` is a read-only register containing the wall-clock tick interval in picoseconds; $\frac{10^{12}}{f}$ where f is the frequency in Hertz (Hz). this register is visible if `scontrol.T` is enabled.

63 … 0
clock frequency

2.1.3. thread address registers

the section describes the user-level thread address registers.

thread address (`addr`)

`addr` is a read-only register containing an alias of the unique system-wide topological address for this thread from `saddr`. this register is visible if `scontrol.D` is enabled.

63	62 … 0
0	*thread address*

thread domain (`domain`)

`domain` is a read-only register containing an alias of the system-wide domain address for this thread from `sdomain`. this register is visible if `scontrol.D` is enabled.

63	62 … 0
1	*thread domain*

2.2. System registers

the architecture provides several privileged status and control registers for trap handling, system configuration, address translation, timers, interrupts, and debugging.

privileged system-level status and control registers can be read and written using the `sysread`, `syswrite`, `sysset`, and `sysclear` instructions. privileged system registers are accessible from executable pages with system or physical address spaces.

privileged registers

the following table lists the privileged trap, system, debug and machine registers:

no.	name	description
privileged trap registers		
0x10	tstatus	trap status
0x11	tcontrol	trap control
0x12	tscratch	trap scratch
0x13	tvalue	trap value
0x14	tcause	trap cause
0x15	tack	trap acknowledge
0x16	tepc	trap exception program counter
0x17	teib	trap exception immediate base
0x18	thpc	trap handler program counter
0x19	thib	trap handler immediate base
privileged system registers		
0x21	scontrol	system control
0x22	sfeature	system feature
0x23	sversion	system version
0x24	sptr	system page table root
0x25	saddr	system thread address
0x26	sdomain	system thread domain
0x27	starget	system target address
0x28	smessage	system target message
0x29	stimer	system deadline timer
0x2a	sie	system interrupt enable
0x2b	sip	system interrupt pending
privileged debug registers		
0x30	dstatus	debug status
0x32	dcycle	debug cycle counter
0x33	dinst	debug instruction counter
0x34	dstop	debug stop instruction
0x35	dfetch	debug monitor fetch address
0x36	dread	debug monitor read address
0x37	dwrite	debug monitor write address

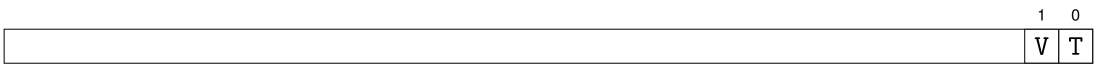
Table 2.5.: privileged registers

2.2.1. privileged trap registers

the section describes the privileged trap registers.

trap status (`tstatus`)

`tstatus` is a read-only register containing trap status. the `T` field indicates whether a timer interrupt is pending. the `V` field indicates whether a virtual interrupt is pending.

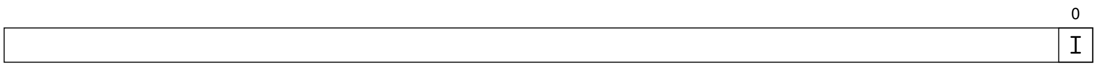


no.	name	description
`1 << 0`	T	timer interrupt pending
`1 << 1`	V	virtual interrupt pending

Table 2.6.: trap status fields

trap control (`tcontrol`)

`tcontrol` is a read-write register containing control bits. the `I` field is read-write and controls whether system interrupts are enabled.

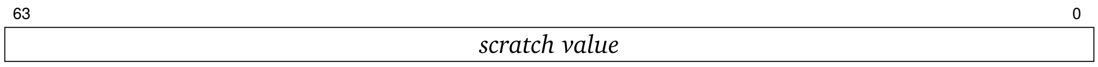


no.	name	description
`1 << 0`	I	system interrupt enable

Table 2.7.: trap control fields

trap scratch (`tscratch`)

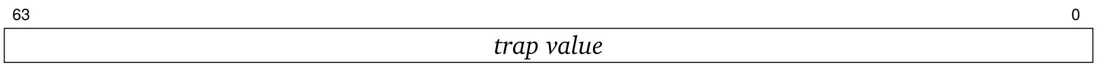
`tscratch` is a read-write save register for use during trap handling.



trap value (`tvalue`)

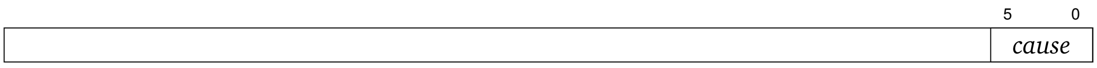
`tvalue` is a read-only register containing the identity of the trap and may contain:

- the faulting instruction word for *break* and *illegal instruction exceptions*.
- the faulting address for *misaligned*, *access* and *page faults*.
- the message value for *virtual interrupts*.
- the system clock time for *timer interrupts*.



trap cause (`tcause`)

`tcause` is a read-only register that contains the cause of the current trap.

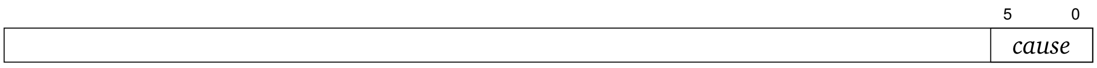


no.	**name**	**description**
system exceptions		
1	break-instruction	break instruction exception
2	illegal-instruction	illegal instruction exception
3	debug-monitor	debug monitor exception
4	misaligned-fetch	fetch misaligned
5	misaligned-load	load misaligned
6	misaligned-store	store misaligned
7	access-fault-fetch	fetch access fault
8	access-fault-load	load access fault
9	access-fault-store	store access fault
10	page-fault-fetch	fetch page fault
11	page-fault-load	load page fault
12	page-fault-store	store page fault
13	capability-fault	capability fault
14	domain-fault	domain fault
15	machine-fault	machine fault
system interrupts		
30	timer-interrupt	timer interrupt
31	virtual-interrupt	virtual interrupt
32...63	interrupt-n	interrupt pins 0 – 31

Table 2.8.: trap causes

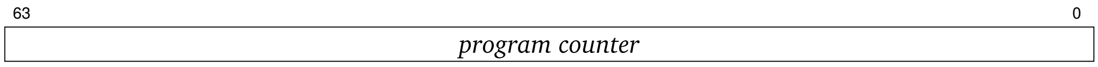
trap acknowledge (`tack`)

`tack` is a write-only register where the trap cause is written to acknowledge the trap so that interrupts are not delivered while state is being saved, and for double-faults to be detected.



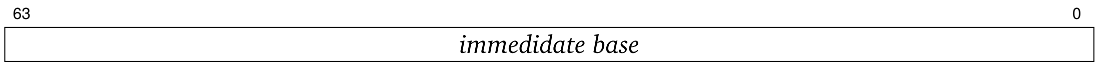
trap exception program counter (`tepc`)

`tepc` is a read-only register which contains the *program counter* address before the trap.



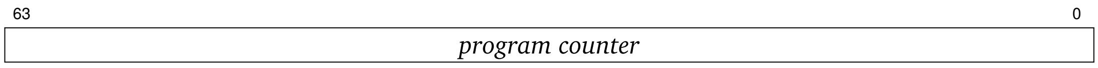
trap exception immedidate base (`teib`)

`teib` is a read-only register which contains the *immediate base* address before the trap.



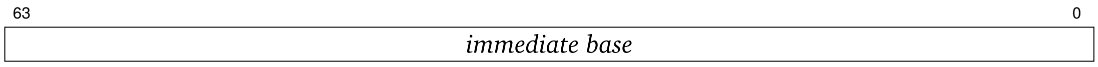
trap handler program counter (`thpc`)

`thpc` is a read-write register containing the address of the system trap handler routine.



trap handler immediate base (`thib`)

`thib` is a read-write register containing the address of the system trap handler constants.



2.2.2. privileged system registers

the section describes the privileged system registers.

system control (`scontrol`)

`scontrol` is a read-write register containing system control bits.

	4	3	2	1	0
	E	A	D	T	F

no.	name	description
`1 << 0`	F	floating-point unit enabled
`1 << 1`	T	user clock and frequency enabled
`1 << 2`	D	user address and domain enabled
`1 << 3`	A	user address space access disable enabled
`1 << 4`	E	user address space execute disable enabled

Table 2.9.: system control fields

system feature (`sfeature`)

`sfeature` is a read-only register containing system feature bits.

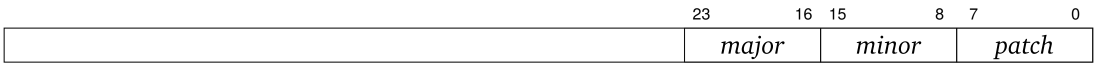
	4	3	2	1	0
	E	A	D	T	F

no.	name	description
`1 << 0`	F	floating-point unit feature
`1 << 1`	T	user clock and frequency feature
`1 << 2`	D	user address and domain feature
`1 << 3`	A	user address space access disable feature
`1 << 4`	E	user address space execute disable feature

Table 2.10.: system feature fields

system version (`sversion`)

`sversion` is a read-only register containing a specification version number described by its *major*, *minor*, and *patch* fields. the allowable values are specified in the following table.

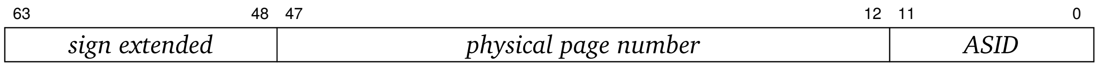


major	minor	patch	description
0	6	0	pre-release current

Table 2.11.: system version values

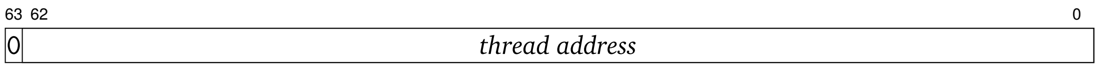
system page table root (`sptr`)

`sptr` is a read-write register containing the page table root physical address for address translation on this thread. the *ASID* field contains the address space identifier for this thread.



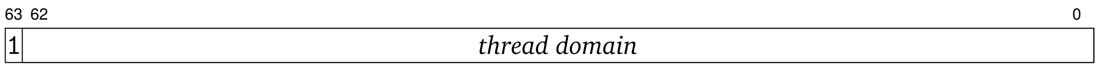
system thread address (`saddr`)

`saddr` is a read-only register containing the unique system-wide topological address for this thread. the system-wide topological address is used as a target for inter-thread virtual interrupts. this address is uniquely assigned during system initialization.



system thread domain (`sdomain`)

`sdomain` is a read-only register containing the system-wide domain address for this thread. the system-wide domain address is used as a target for inter-thread virtual interrupts. this address can alias. this address is assigned during context switching.



system target address (`starget`)

`starget` is a read-write register containing the system-wide target address of a thread to send a virtual interrupt. all-zeros is the *boot service processor*. all-ones is the *broadcast address*. zero in the *A* field is a *topological address*. one in the *A* field is a *domain address*.

63	62	0
A	*target address*	

system target message (`smessage`)

`smessage` is a write-only register that causes an edge-triggered virtual interrupt to be queued to the thread set in `starget`. virtual interrupts are marked pending in the `V` field of the `tstatus` register for the target thread.

63	0
message value	

system deadline timer (`stimer`)

`stimer` is a read-write register containing the deadline timer for this thread. when the system clock reaches the value in the register, a *timer interrupt* is triggered. timer interrupts are marked pending in the `T` field of the `tstatus` register.

63	0
clock time	

system interrupt enable (`sie`)

`sie` is a read-write register that contains interrupt enable flags for 32 wired interrupt pins.

	31	0
	interrupt enable 0-31	

system interrupt pending (`sip`)

`sip` is a read-only register that contains interrupt pending flags for 32 wired interrupt pins.

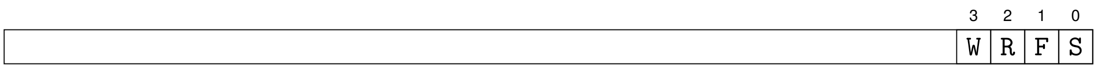
	31	0
	interrupt pending 0-31	

2.2.3. privileged debug registers

the section describes the privileged debug registers.

debug status (`dstatus`)

`dstatus` is a read-only register containing status bits for debug monitor execeptions.

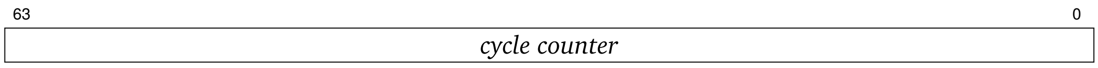


no.	name	description
`1 << 0`	S	stopping instruction
`1 << 1`	F	fetch monitor address
`1 << 2`	R	load monitor address
`1 << 3`	W	store monitor address

Table 2.12.: debug status fields

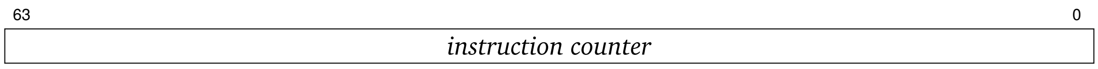
debug cycle counter (`dcycle`)

`dcycle` is a read-only register containing the number of cycles retired since power on.



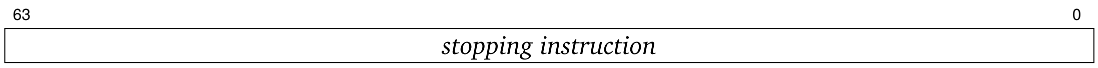
debug instruction counter (`dinst`)

`dinst` is a read-only register containing the number of instructions retired since power on.



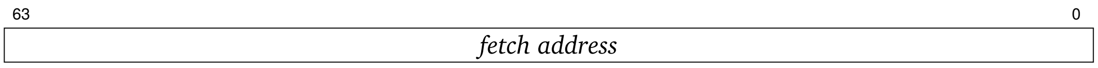
debug stop instruction (`dstop`)

`dstop` is a read-write register containing an instruction number to halt execution on and raise a *debug monitor exception*.



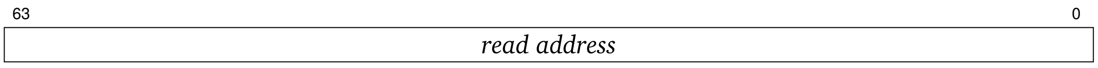
debug monitor fetch address (`dfetch`)

`dfetch` is a read-write register containing a memory fetch address to monitor for and halt execution on and raise a *debug monitor exception* for a matching memory fetch address.



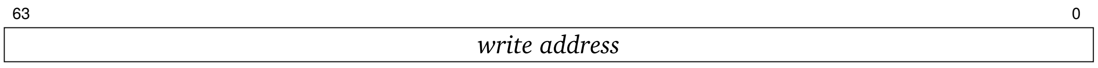
debug monitor read address (`dread`)

`dread` is a read-write register containing the memory load address to monitor for and halt execution on and raise a *debug monitor exception* for a matching memory load address.



debug monitor write address (`dwrite`)

`dwrite` is a read-write register containing the memory store address to monitor for and halt execution on and raise a *debug monitor exception* for a matching memory store address.



2.3. Capability extension

the architecture provides an optional capabilities extension that provides a capability check for permissions and capabilities enabled by *page table colors* in page table entries. color permissions mask pask page table permissions based on *page table color*. color capabilities restrict access to system, domain, capability and machine registers based on *page table color*.

the following tables list the privileged capability system parameters:

name	**description**
ncol	the number of architectural colors

Table 2.13.: capability parameters

capability registers

the following tables list the privileged capability system registers:

no.	**name**	**description**	**depends**
capability control registers			
`0x41`	ccontrol	capability control	
capability permission registers			
`0x43`	colorread	color read permissions	
`0x44`	colorwrite	color write permissions	
`0x45`	colorexec	color exec permissions	
capability role registers			
`0x46`	colorsys	color system role	
`0x47`	colordom	color domain role	domain
`0x48`	colorcap	color capability role	
`0x49`	colormac	color machine role	machine
capability matrix registers			
`0x100...0x10f`	colormatread[n]	color matrix read permissions	
`0x200...0x20f`	colormatwrite[n]	color matrix write permissions	
`0x300...0x30f`	colormatexec[n]	color matrix exec permissions	

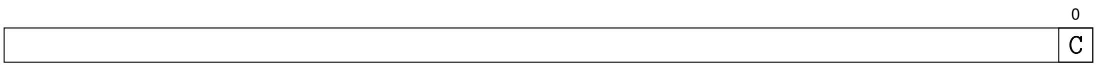
Table 2.14.: capability registers

2.3.1. capability control registers

this section describes the *capability control registers* which enable the capability extension.

capability control (`ccontrol`)

`ccontrol` is a read-write register containing capability control bits.



no.	name	description
`1 << 0`	C	capability check enable

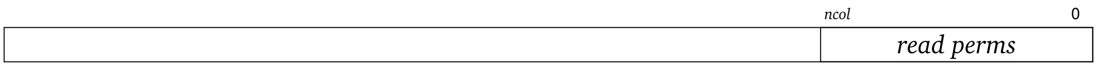
Table 2.15.: color control fields

2.3.2. capability permission registers

this section describes the *capability permission registers* which define *page table color* permission masks for page table entries. when `ccontrol.C` is set, the *capability permission registers* are combined with page table permissions to limit maximum *read*, *write*, or *execute* permission based on page table color.

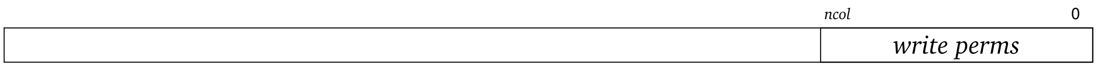
color read permission (`colorread`)

`colorread` is a read-write register containing an array of ***ncol*** bits containing read permissions for *color-x*, used to check loads from pages with a matching page table color.



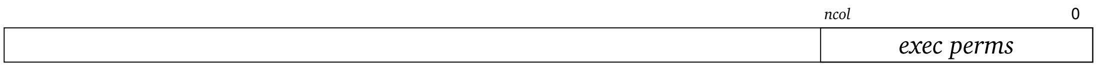
color write permission (`colorwrite`)

`colorwrite` is a read-write register containing an array of ***ncol*** bits containing write permissions for *color-x*, used to check stores to pages with a matching page table color.



color exec permission (`colorexec`)

`colorexec` is a read-write register containing an array of ***ncol*** bits containing execute permissions for *color-x*, used to check execution in pages with a matching page table color.

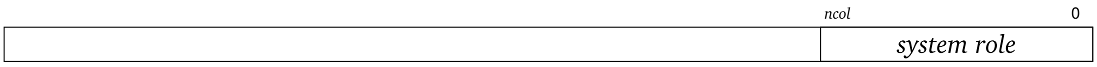


2.3.3. capability role registers

this section describes the *capability role registers*. the *capability role registers* describe access to privileged system registers and instructions based on *page table color*. when `ccontrol.C` is set, privileged system registers and instructions and will cause capability traps for illegal accesses. system registers and instructions are restricted based on the following roles: *system*, *domain*, *capability* and *machine*.

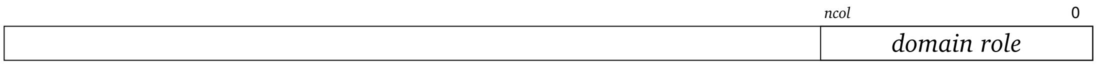
color system role (`colorsys`)

`colorsys` is a read-write register containing an array of ***ncol*** bits containing system role for *color-x*, used to check access to privileged trap registers, privileged system registers and privileged debug registers from executable pages with a matching page table color.



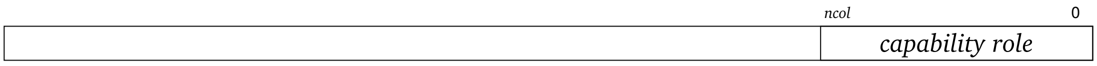
color domain role (`colordom`)

`colordom` is a read-write register containing an array of ***ncol*** bits containing domain role for *color-x* that are used to check access to privileged domain registers from executable pages with a matching page table color.



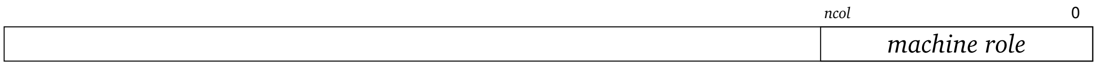
color capability role (`colorcap`)

`colorcap` is a read-write register containing an array of ***ncol*** bits containing capability role for *color-x* that are used to check access to privileged capability registers from executable pages with a matching page table color.



color machine role (`colormac`)

`colormac` is a read-write register containing an array of ***ncol*** bits containing machine role for *color-x* that are used to check access to privileged machine registers from executable pages with a matching page table color.

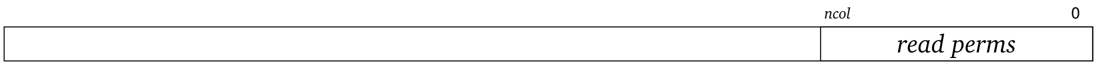


2.3.4. capability matrix registers

this section describes the color matrix registers. the *capability matrix registers* are permission registers that protect access to memory based on a combination of *source* and *destination* page table color and *read*, *write*, or *execute* permissions. when `ccontrol.C` is set, memory accesses and branches to memory addresses in pages with matching *source* and *destination* page table colors, and wider permissions, will cause capability traps.

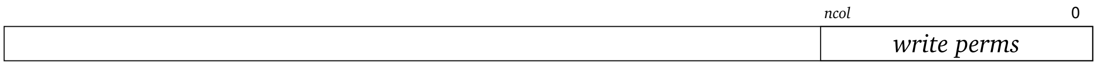
color matrix read permissions (`colormatread`[*ncol*])

`colormatread`[*ncol*] are ***ncol*** read-write registers containing an array of ***ncol*** bits with read permissions, used to check loads from executable pages with a matching register *page table color* to memory addresses in memory pages with a matching *page table color* index.



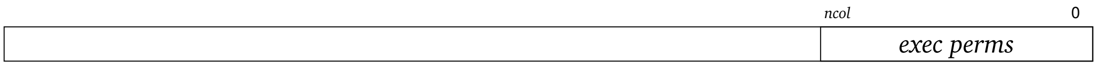
color matrix write permissions (`colormatwrite`[*ncol*])

`colormatwrite`[*ncol*] are ***ncol*** read-write registers containing an array of ***ncol*** bits with write permissions, used to check stores from executable pages with a matching register *page table color* to memory addresses in memory pages with a matching *page table color* index.



color matrix exec permissions (`colormatexec`[*ncol*])

`colormatexec`[*ncol*] are ***ncol*** read-write registers containing an array of ***ncol*** bits with exec permissions, used to check branches from executable pages with a matching register *page table color* to branch addresses in executable pages with a matching *page table color* index.



2.4. Domain extension

the architecture provides an optional domain [3] extension for virtual machines. the extension adds registers for virtual domain addresses, trap routing, page translation, and context switching for system traps, domain traps, capability traps and machine traps.

the following tables list the privileged domain system parameters:

name	description
ndom	the number of architectural domains

Table 2.16.: domain parameters

domain registers

the following tables list the privileged domain system registers:

no.	name	description	depends
domain control registers			
0x51	domcontrol	domain control	
0x53	domlast	domain number last	
0x54	domcurr	domain number current	
0x55	domnext	domain number next	
domain routing registers			
0x500...0x50f	dommatrix[*n*]	domain matrix	
0x600...0x60f	domsys[*n*]	domain system default domain	
0x700...0x70f	domdom[*n*]	domain domain default domain	
0x800...0x80f	domcap[*n*]	domain capability default domain	capability
0x900...0x90f	dommac[*n*]	domain machine default domain	machine
domain context registers			
0xa00...0xa0f	domaddr[*n*]	domain address	
0xb00...0xb0f	domptr[*n*]	domain page table root	
0xc00...0xc0f	domthpc[*n*]	domain trap handler program counter	
0xd00...0xd0f	domthib[*n*]	domain trap handler immediate base	

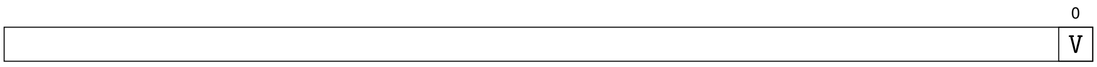
Table 2.17.: domain registers

2.4.1. domain control registers

this section describes the *domain control registers* which enable the domain extension and provide access to last and current domain number and control the next domain number.

domain control (`domcontrol`)

`domcontrol` is a read-write register containing domain control bits.

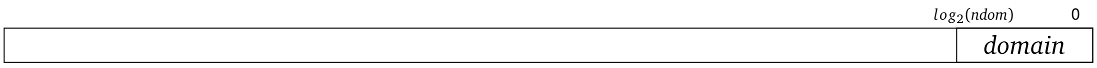


no.	name	description
`1 << 0`	V	virtual domains enable

Table 2.18.: domain control fields

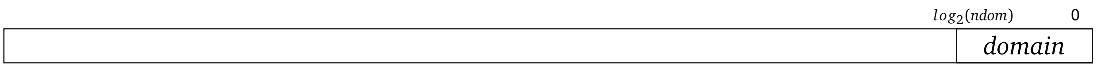
domain number last (`domlast`)

`domlast` is a read-only register containing the domain number of the last domain.



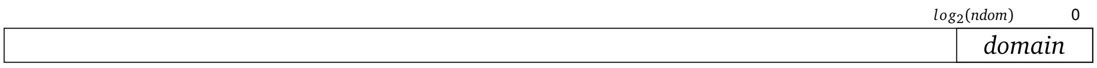
domain number current (`domcurr`)

`domcurr` is a read-only register containing the domain number of the current domain.



domain number next (`domnext`)

`domnext` is a read-write register containing the domain number of the next domain.

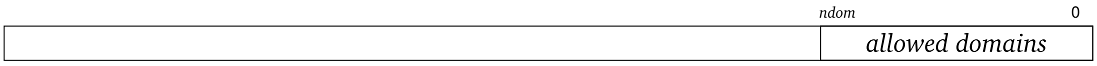


2.4.2. domain routing registers

this section describes the *domain routing registers* that control valid domain transitions and routing for context switching of system, domain, capability and machine traps.

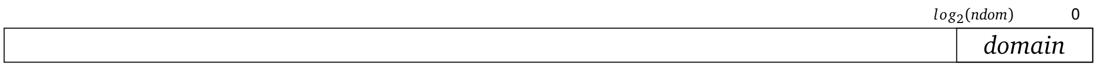
domain matrix (`dommatrix`[*ndom*])

`dommatrix`[*ndom*] are ***ndom*** read-write registers containing a matrix of the domains that the current domain is allowed to transition to as the next domain.



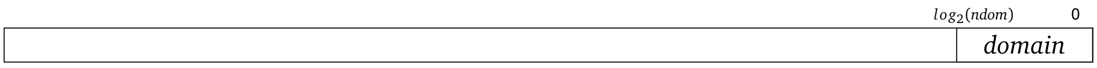
domain system default domain (`domsys`[*ndom*])

`domsys`[*ndom*] are ***ndom*** read-write registers containing the domain number of the domain that system traps are routed to.



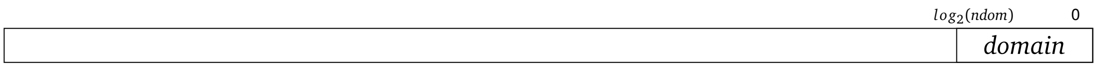
domain domain default domain (`domdom`[*ndom*])

`domdom`[*ndom*] are ***ndom*** read-write registers containing the domain number of the domain that domain traps are routed to.



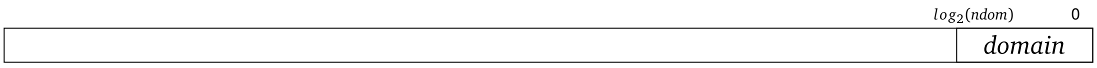
domain capability default domain (`domcap`[*ndom*])

`domcap`[*ndom*] are ***ndom*** read-write registers containing the domain number of the domain that capability traps are routed to.



domain machine default domain (`dommac`[*ndom*])

`dommac`[*ndom*] are ***ndom*** read-write registers containing the domain number of the domain that machine traps are routed to.



2.4.3. domain context registers

this section describes the *domain context registers* that control virtual interrupt routing, page translation and trap entry functions for domain context switching.

domain address (`domaddr`[*ndom*])

`domaddr`[*ndom*] are ***ndom*** read-write registers containing an array of the domain addresses associated with each domain. domain context switches copy the value for the target domain into `sdomain` to enable virtual interrupt delivery.

63	62 … 0
1	*domain address*

domain page table root (`domptr`[*ndom*])

`domptr`[*ndom*] are ***ndom*** read-write registers containing an array of the page table root physical addresses for address translation in each domain. domain context switches copy the value for the target domain into `sptr` to enable address translation.

63 … 48	47 … 12	11 … 0
sign extended	*physical page number*	*ASID*

domain trap handler program counter (`domthpc`[*ndom*])

`domthpc`[*ndom*] are ***ndom*** read-write registers containing an array of the addresses for the domain trap handler routines in each domain. the address is a virtual address interpreted in relation to the page table root for the domain. domain context switches copy the value for the target domain into `thpc` and set the *program counter* register.

63 … 0
program counter

domain trap handler immediate base (`domthib`[*ndom*])

`domthib`[*ndom*] are ***ndom*** read-write registers containing an array of the addresses for the domain trap handler constants in each domain. the address is a virtual address interpreted in relation to the page table root for the domain. domain context switches copy the value for the target domain into `thib` and set the *immediate base* register.

63 … 0
immediate base

2.5. Machine extension

the architecture provides an optional machine extension that includes machine-level registers for physical memory protection and software-defined memory management.

machine registers

the following tables list the privileged machine registers:

no.	**name**	**description**
machine control registers		
`0x61`	mcontrol	machine control
`0x62`	mfeature	machine feature
machine memory registers		
`0x63`	mtlbkey	machine translation lookaside buffer key
`0x64`	mtlbfmt	machine translation lookaside buffer format
`0x65`	mtlbent	machine translation lookaside buffer entry

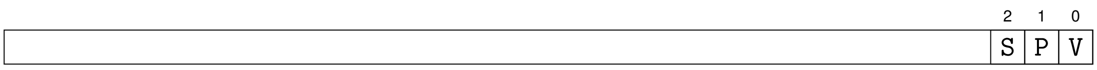
Table 2.19.: machine registers

2.5.1. machine control registers

the section describes the privileged machine registers for machine control and features.

machine control (`mcontrol`)

`mcontrol` is a read-write register containing machine control bits.

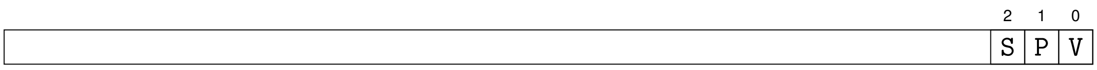


no.	name	description
`1 << 0`	V	virtual address translation enable
`1 << 1`	P	physical permissions check enable
`1 << 2`	S	software page translation enable

Table 2.20.: machine control fields

machine feature (`mfeature`)

`mfeature` is a read-only register containing machine feature bits.



no.	name	description
`1 << 0`	V	virtual address translation feature
`1 << 1`	P	physical permissions check feature
`1 << 2`	S	software page translation feature

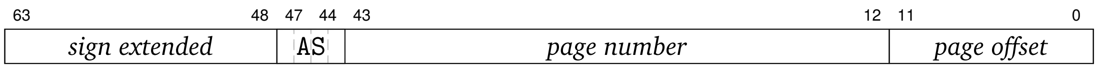
Table 2.21.: machine feature fields

2.5.2. machine memory registers

the section describes the privileged machine registers for address translation. the registers in this section enable a *machine-fault* trap handler to perform address translation by walking the page table structures and populating the TLB[1] as part of a software-defined MMU[2].

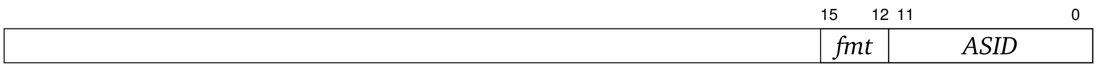
machine translation lookaside buffer key (`mtlbkey`)

`mtlbkey` is a read-only register containing the machine fault translation virtual address for the current page translation miss fault.



machine translation lookaside buffer format (`mtlbfmt`)

`mtlbfmt` is a read-write register containing the machine fault address space identifier and page table format for the current page translation miss fault.



no.	size	description
0	2^{12}	v48 page table entry 4KiB page
1	2^{21}	v48 page table entry 2MiB page
2	2^{30}	v48 page table entry 1GiB page

Table 2.22.: 64-bit machine translation lookaside buffer format field

machine translation lookaside buffer entry (`mtlbent`)

`mtlbent` is a read-write register containing the machine fault translation lookaside buffer entry for the current page translation miss fault. this register is populated by the page translation miss fault handler with a physical page number, a page color, plus several permission and metadata bits in the same format as the metadata bits defined for page table entries. table 2.26 in the address translation section describes the metadata bits.

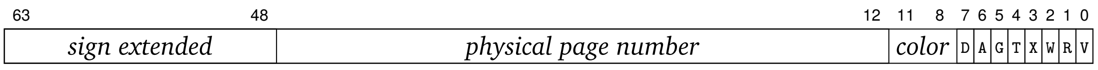


Figure 2.1.: 64-bit machine translation lookaside buffer entry structure.

[1]Translation Lookaside Buffer
[2]Memory Management Unit

2.6. Environment extension

the architecture provides an optional environment extension that includes machine-level registers for diagnostics, warm reset, cold reset, and power off.

environment registers

the following tables list the privileged environment registers:

no.	**name**	**description**
environment control registers		
`0x71`	econtrol	environment control

Table 2.23.: environment registers

2.6.1. environment control registers

the section describes the privileged registers for environment control.

environment control (`econtrol`)

`econtrol` is a read-write register used to issue machine-level environment control commands. the `OP` field contains an opcode with values for: *diagnostic*, *warm reset*, *cold reset* and *power off*. the *cause code* field is recorded in an environment log.

63 … 56	55 … 0
`OP`	*cause code*

no.	description
0	diagnostic
1	warm reset
2	cold reset
3	power off

Table 2.24.: environment control opcode field

2.7. Address translation

the architecture provides for page-based virtual to physical address translation using a page table *trie structure* composed of index pages containing arrays of page table entries. a page walker reads the structure from the root pointer to leaf entries to translate virtual addresses into physical addresses. the architecture introduces the concept of a *translation address* which is an address boxed with an *address space prefix* (`AS`) designed to provide a canonical address form for user and system virtual addresses as well as physical addresses.

2.7.1. page table structure

the section describes the dimensions for memory address translation and the *trie-based* page table structure pointed to by the *system page table root* (`sptr`) register.

Page Table Levels	**Page Number Bits**	**Page Offset Bits**	**Virtual Address Bits**	**Page Index Entries**	**Page Sizes**
4	9	12	44 — 48	512	4KiB, 2Mib, 1GiB

Table 2.25.: 64-bit page table dimensions

2.7.2. page table entries

page table entries are grouped into arrays within index pages which are selected by an index derived from a portion of the virtual address. each entry contains a physical page number, a page color, plus several permission and metadata bits. the physical page number points to the next page table level for *pointer* entries or the translated physical address for *leaf* entries.

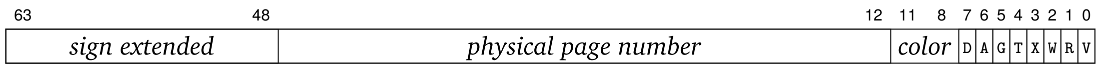


Figure 2.2.: 64-bit page table entry structure.

no.	**name**	**description**
`1 << 0`	V	valid
`1 << 1`	R	read
`1 << 2`	W	write
`1 << 3`	X	execute
`1 << 4`	T	translate
`1 << 5`	G	global
`1 << 6`	A	accessed
`1 << 7`	D	dirty

Table 2.26.: 64-bit page table entry fields

2.7.3. page table addresses

the page table translation system has three types of addresses: translation addresses, virtual addresses and physical addresses. page table translation addresses have a 1 — 4 bit address space prefix to allow them to contain user and system virtual addresses as well as physical addresses. the page table translation and lookup virtual address structure is as follows:

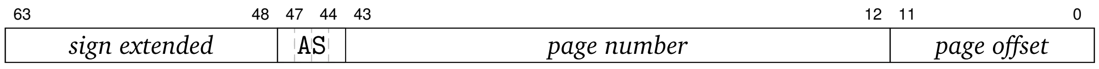


Figure 2.3.: 64-bit translation address structure.

no.	name
`0b0.`	user
`0b10.`	system
`0b110.`	physical

Table 2.27.: 1—4 bit translation address space prefixes.

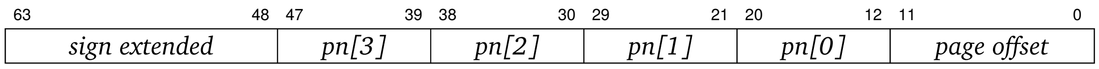


Figure 2.4.: 64-bit lookup virtual address structure.

2.7.4. page table translation

the page table walker performs page table lookups to translate virtual addresses into physical addresses. the page table walker reads the page table root pointer then walks the *trie-based* page table structure to find a leaf entry containing a physical page address.

page table entries with the *read, write* and *exec* permission bits clear are interpreted as *pointer* entries and contain a pointer to the next-level index page, otherwise they are interpreted as *leaf* entries with a final translation. entries with the *write* bit set must have the *read* bit set. entries without the *accessed* bit set will fault on read accesses. entries without the *dirty* bit set will fault on write accesses. the *global* bit prevents *sptr.ASID* from being associated with TLB entries. the *color* field is reserved for use by system software and the capabilities extension.

if the physical permissions feature *mfeature.P* is present and *mcontrol.P* is enabled, then physical pages must be present in the page table with a self-mapping of their physical address prefixed with ($AS = physical$) leading to a leaf page table entry containing its own address. this is to allow physical page permissions to be checked. in addition to this, the physical self-mappings for physical pages containing page table pages must have the *translate* bit set.

the page table lookup and virtual to physical address translation process are as follows:

Algorithm 1 find page table entry for translation address

```
function FIND_PAGE($type, level, ppn, ta$)
    $shift \leftarrow level \times pn\_bits + po\_bits$    ▷ calculate page number shift
    $tpn \leftarrow (ta >> shift) \wedge ((1 << pn\_bits) - 1)$    ▷ calculate translation page number
    $pte \leftarrow$ LOAD$((ppn << po\_bits) + tpn \times sizeof(PTE))$    ▷ load page table entry
    $leaf \leftarrow pte.R \vee pte.W \vee pte.X$    ▷ page table entry node type
    $sm \leftarrow R \vee W \vee X$    ▷ default self map page table entry
    if $mfeature.P \wedge mcontrol.P \wedge type = virtual$ then
        $pa \leftarrow pte.ppn << po\_shift$
        $(sm, level) \leftarrow$ FIND_PAGE$(physical, num\_levels - 1, sptr.ppn, pa)$    ▷ load self mapping
        if $(leaf \wedge pte.ppn \neq sm.ppn) \vee (\neg leaf \wedge \neg sm.T)$ then
            raise fault    ▷ invalid self mapping
        end if
    end if
    if $\neg leaf \wedge level > 0$ then
        return FIND_PAGE$(type, level - 1, pte.ppn, ta)$    ▷ follow pointer entry
    else if $\neg leaf$ then
        raise fault    ▷ leaf entry not found
    end if
    return $(pte, sm, level)$    ▷ return page table entry
end function
```

Algorithm 2 translate virtual address to physical address

```
function TRANSLATE($op, va$)
    if $\neg$CHECK_CANONICAL$(va, va\_bits)$ then    ▷ check address is sign-extended
        raise fault
    end if
    $(pte, sm, level) \leftarrow$ FIND_PAGE$(virtual, num\_levels - 1, sptr.ppn, va)$    ▷ find page table entry
    $mask \leftarrow ((1 << (level \times pn\_bits + po\_bits)) - 1)$
    $caps \leftarrow (R \wedge colorread[pte.color]) \vee (W \wedge colorwrite[pte.color]) \vee (X \wedge colorexec[pte.color])$
    $user \leftarrow \neg((va >> (va\_bits - 1)) \wedge 1)$
    if $(\neg pte.V) \vee (pte.W \wedge \neg pte.R)$ then
        raise fault    ▷ invalid write must have read
    else if $(op.R \wedge \neg pte.R) \vee (op.W \wedge \neg pte.W) \vee (op.X \wedge \neg pte.X)$ then
        raise fault    ▷ invalid permissions
    else if $(sm.R \wedge \neg pte.R) \vee (sm.W \wedge \neg pte.W) \vee (sm.X \wedge \neg pte.X)$ then
        raise fault    ▷ invalid self map
    else if $(cap.R \wedge pte.R) \vee (cap.W \wedge pte.W) \vee (cap.X \wedge pte.X)$ then
        raise fault    ▷ invalid capabilities
    else if $(op.R \wedge \neg pte.A) \vee (op.W \wedge \neg pte.D)$ then
        raise fault    ▷ invalid accessed or dirty
    else if $\neg tstatus.U \wedge \neg scontrol.E \wedge user \wedge op.X$ then
        raise fault    ▷ invalid user page execute
    else if $\neg tstatus.U \wedge \neg scontrol.A \wedge user \wedge op.(R \vee W)$ then
        raise fault    ▷ invalid user page access
    else if $(pte.ppn << po\_bits) \wedge mask \neq 0$ then
        raise fault    ▷ invalid superpage alignment
    end if
    return $(pte.ppn << po\_bits) + (va \wedge mask)$
end function
```

3. Instructions

3.1. Instruction listing — 16-bit

3.1.1. break

15 — 7	6 — 2	1 — 0
imm	*opcode*	*size*
uimm	**00000**	**00**

break *uimm9*

the *break* instruction causes a debugger trap. *program counter* and trap cause are saved to privileged registers for the operating system to dispatch to a debugger and the *program counter* is set to a trap vector address.

3.1.2. j

15 — 7	6 — 2	1 — 0
imm	*opcode*	*size*
simm	**00001**	**00**

j *simm9* × 2

the *j* or *jump* instruction is an unconditional branch instruction that adds a relative immediate address to the *program counter*. the resulting *program counter* address is $[pc + simm9 \times 2]$.

```
pc = pc + simm9 * 2
```

3.1.3. b

15 — 7	6 — 2	1 — 0
imm	*opcode*	*size*
simm	**00010**	**00**

b *simm9* × 2

the *b* or *branch* instruction is a conditional branch instruction that adds a relative immediate address to the *program counter*. if the *flag* register has been set by a compare instruction, the resulting *program counter* address is $[pc + simm9 \times 2]$, otherwise the *program counter* is advanced normally.

```
if flag:
    pc = pc + simm9 * 2
```

3.1.4. ibj

15 7	6 2	1 0
imm	*opcode*	*size*
simm	**00011**	**00**

ibj *simm9* × 64

the *ibj* or *immediate-block-jump* instruction adds a 64-bit relative address to the *immediate base* register. the resulting *immediate base* address is $[ib + simm9 \times 64]$.

```
ib = ib + simm9 * 64
```

3.1.5. link

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
fun	**uimm**	**00100**	**00**

link.i64 *fun3*, *ib64*(*uimm6*)

the *link* instruction loads a 64-bit constant addressed by $[ib + uimm6 \times 8]$ containing a *i32x2* relative address vector, which it adds it to *(pc,ib)*, conditionally subtracts the source link register *(r6 or r7)*, then conditionally adds the difference to the destination link register *(r6 or r7)*, depending on the value of *fun3*. the type of linkage in *fun3* can be one of:

value	mnemonic	description	link-type	dst-link	src-link
0	jib	jump	jump	-	-
1	-	-	-	-	-
2	jalib	jump-and-link	call	r6	-
3	jalib	jump-and-link	call	r7	-
4	jtlib	jump-to-link	ret	-	r6
5	jtlib	jump-to-link	ret	-	r7
6	jalaib	jump-and-link-add	tail	r6	-
7	jalaib	jump-and-link-add	tail	r7	-

Table 3.1.: Link instruction functions

```
lr = 0b110 + (fun3 & 1)
cval = const-mem<i32x2>[ib + uimm6 * 8]
dval = fun3 == jalaib ? cval + auth-decrypt(reg<i32x2>[lr]) : cval
lval = fun3 == jtlib  ?        auth-decrypt(reg<i32x2>[lr]) : 0
(dpc,dib) = dval
(lpc,lib) = lval
pc = pc + dpc - lpc
ib = ib + dib - lib
if fun3 == jalib or fun3 == jalaib:
    reg[lr] = auth-encrypt(reg<i32x2>(dpc,dib))
```

the *auth-encrypt* and *auth-decrypt* functions are placeholders for authenticated encryption, and decryption of relative address vectors. authentication failures should generate a trap for the operating system to dispatch to a control flow integrity trap handler.

3.1.6. movh

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**00101**	**00**

movh.i64 *rc*, *ib32*(*uimm6*)

the *movh* or *move-half-immediate-block* instruction loads a 32-bit constant addressed by $[ib + uimm6 \times 4]$, which it sign-extends to 64-bits, then saves to the *rc* register.

```
reg[rc] = const-mem<i32>[ib + uimm6 * 4]
```

3.1.7. movw

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**00110**	**00**

movw.i64 *rc*, *ib64*(*uimm6*)

the *movw* or *move-word-immediate-block* instruction loads a 64-bit constant addressed by $[ib + uimm6 \times 8]$ then saves it to the *rc* register.

```
reg[rc] = const-mem<i64>[ib + uimm6 * 8]
```

3.1.8. movi

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**simm**	**00111**	**00**

movi.i64 *rc*, *simm6*

the *movi* or *move-immediate* instruction sign-extends the immediate value in *simm6* then saves the result in the *rc* register.

```
reg[rc] = simm6
```

3.1.9. addi

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**simm**	**01000**	**00**

addi.i64 *rc*, *simm6*; *flag*

the *addi* or *add-immediate* instruction sign-extends the immediate value in *simm6* then adds it to the *rc* register, and carry is saved to the *flag* register.

```
result = reg[rc] + simm6
flag = result < reg[rc]
reg[rc] = result
```

3.1.10. srli

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**01001**	**00**

srli.i64 *rc, uimm6*

the *srli* or *shift-right-logical-immediate* instruction performs a logical right shift by *uimm6* bits of the value in the *rc* register. zeros are copied into the left most bits.

```
reg[rc] = reg<u64>[rc] >> uimm6
```

3.1.11. srai

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**01010**	**00**

srai.i64 *rc, uimm6*

the *srai* or *shift-right-arithmetic-immediate* instruction performs an arithmetic right shift by *uimm6* bits of the value in the *rc* register. the sign is copied into the left most bits.

```
reg[rc] = reg<i64>[rc] >> uimm6
```

3.1.12. slli

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**01011**	**00**

slli.i64 *rc, uimm6*

the *slli* or *shift-left-logical-immediate* instruction performs a logical left shift by *uimm6* bits of the value in the *rc* register. zeros are copied into the right most bits.

```
reg[rc] = reg[rc] << uimm6
```

3.1.13. addh

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**01100**	**00**

addh.i64 *rc, ib32(uimm6);flag*

the *addh* or *add-half-immediate-block* instruction loads a 32-bit constant addressed by $[ib + uimm6 \times 4]$, which it sign-extends to 64-bits, then adds to the *rc* register, and carry is saved to the *flag* register.

```
result = reg[rc] + const-mem<i32>[ib + uimm6 * 4]
flag = result < reg[rc]
reg[rc] = result
```

3.1.14. leapc

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**01101**	**00**

leapc.i64 *rc*, *ib32*(*uimm6*)(*pc*)

the *leapc* or *load-effective-address-pc* instruction loads a 32-bit constant addressed by $[ib + uimm6 \times 4]$, which it sign-extends to 64-bits, adds it to the *program counter*, and saves the result in the *rc* register.

```
reg[rc] = pc + const-mem<i32>[ib + uimm6 * 4]
```

3.1.15. loadpc

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**01110**	**00**

loadpc.i64 *rc*, *ib32*(*uimm6*)(*pc*)

the *loadpc* instruction loads a 32-bit constant addressed by $[ib + uimm6 \times 4]$ which it sign-extends to 64-bit, adds it to the *program counter* to form an address, then loads a 64-bit value from memory at that address and saves the result in the *rc* register.

```
reg[rc] = mem<i64>[pc + const-mem<i32>[ib + uimm6 * 4]]
```

3.1.16. storepc

15 13	12 7	6 2	1 0
rc	*imm*	*opcode*	*size*
rc	**uimm**	**01111**	**00**

storepc.i64 *rc*, *ib32*(*uimm6*)(*pc*)

the *storepc* instruction loads a 32-bit constant addressed by $[ib + uimm6 \times 4]$ which it sign-extends to 64-bit, adds it to the *program counter* to form an address, then stores to memory at that address a 64-bit value from the *rc* register.

```
mem<i64>[pc + const-mem<i32>[ib + uimm6 * 4]] = reg[rc]
```

3.1.17. load

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*imm*	*opcode*	*size*
rc	**rb**	**uimm**	**10000**	**00**

load.i64 *rc*, (*uimm3* × 8)(*rb*)

the *load* instruction computes the address $[rb + uimm3 \times 8]$ then loads a 64-bit value from memory at that address and saves the result in the *rc* register.

```
reg[rc] = mem<i64>[reg[rb] + uimm3 * 8]
```

3.1.18. store

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*imm*	*opcode*	*size*
rc	**rb**	**uimm**	**10001**	**00**

store.i64 *rc*, (*uimm3* × 8)(*rb*)

the *store* instruction computes the address $[rb + uimm3 \times 8]$ then stores a 64-bit value to memory at that address containing a 64-bit value from the *rc* register.

```
mem<i64>[reg[rb] + uimm3 * 8] = reg[rc]
```

3.1.19. compare

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*imm*	*opcode*	*size*
rc	**rb**	**fun**	**10010**	**00**

cmp.i64 *rc*, *rb*, *fun3*

the *compare* instruction performs a comparison between the value in *rb* and *rc* then saves the result in the *flag* register. the *compare* opcode is also used to perform conditional move whereby the *rb* register is copied into the *rc* if the *flag* register is set. the type of comparsion in *fun3* can be one of:

value	mnemonic	description
0	*lt*	*less than (signed)*
1	*ge*	*greather or equal (signed)*
2	*eq*	*equal*
3	*ne*	*not equal*
4	*ltu*	*less than (unsigned)*
5	*geu*	*greater or equal (unsigned)*
6	*cmov*	*conditional move*
7	*ncmov*	*negated conditional move*

Table 3.2.: Compare instruction functions

```
match fun3
| lt    -> flag = reg<i64>[rc] <  reg<i64>[rb]
| ge    -> flag = reg<i64>[rc] >= reg<i64>[rb]
| eq    -> flag =      reg[rc] =       reg[rb]
| ne    -> flag =      reg[rc] !=      reg[rb]
| ltu   -> flag = reg<u64>[rc] <  reg<u64>[rb]
| geu   -> flag = reg<u64>[rc] >= reg<u64>[rb]
| cmov  -> if flag:
               reg[rc] = reg[rb]
| ncmov -> if not flag:
               reg[rc] = reg[rb]
```

3.1.20. logic

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*imm*	*opcode*	*size*
rc	**rb**	**fun**	**10011**	**00**

logic.i64 *rc, rb, fun3*

the *logic* instruction performs a logic operation on the value in the *rb* register then stores the result in the *rc* register. the type of logic operations in *fun3* can be one of:

value	mnemonic	description
0	*mv*	*move*
1	*not*	*logical not*
2	*neg*	*negate*
3	*bswap*	*bswap*
4	*ctz*	*count trailing zeros*
5	*clz*	*count leading zeros*
6	*ctpop*	*count population*
7	*sext*	*sign extend*

Table 3.3.: Logic instruction functions

```
match fun3
| mov   -> reg[rc] = reg[rb]
| not   -> reg[rc] = ~reg[rb]
| neg   -> reg[rc] = -reg[rb]
| bswap -> reg[rc] = bswap(reg[rb])
| ctz   -> reg[rc] = ctz(reg[rb])
| clz   -> reg[rc] = clz(reg[rb])
| ctpop -> reg[rc] = ctpop(reg[rb])
| sext  -> reg[rc] = sext(reg[rb])
```

3.1.21. pin

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**10100**	**00**

pin.i64 *rc, rb, ra*

the *pin* or *pack-indirect* instruction packs two absolute addresses as an *i32x2 (pc,ib)* relative address vector. the register *ra* is subtracted from the *program counter*, and the register *rb* is subtracted from the *immediate base* register, and the results are packed into an *i32x2* relative address vector and saved to the register *rc*.

```
lpc   = int<i32>(pc - reg[ra])
lib   = int<i32>(ib - reg[rb])
lval  = (lpc,lib)
reg[rc] = auth-encrypt(lval)
```

3.1.22. and

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**10101**	**00**

and.i64 *rc*, *rb*, *ra*

the *and* instruction performs a *logical-and* of the register *rb* and the register *ra* and saves the result in the register *rc*.

```
reg[rc] = reg[rb] & reg[ra]
```

3.1.23. or

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**10110**	**00**

or.i64 *rc*, *rb*, *ra*

the *or* instruction performs a *logical-or* of the register *rb* and the register *ra* and saves the result in the register *rc*.

```
reg[rc] = reg[rb] | reg[ra]
```

3.1.24. xor

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**10111**	**00**

xor.i64 *rc*, *rb*, *ra*

the *xor* instruction performs a *logical-exclusive-or* of the register *rb* and the register *ra* and saves the result in the register *rc*.

```
reg[rc] = reg[rb] ^ reg[ra]
```

3.1.25. add

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11000**	**00**

add.i64 *rc*, *rb*, *ra*; *flag*

the *add* instruction adds the *rb* register to the *ra* register and saves the result in the *rc* register, and carry is saved to the *flag* register.

```
result = reg[rb] + reg[ra]
flag = result < reg[rb]
reg[rc] = result
```

3.1.26. srl

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11001**	**00**

srl.i64 *rc, rb, ra*

the *srl* or *shift-right-logical* instruction performs a logical right shift of the value in the *rb* register by the number of bits in register *ra* then saves the result in the *rc* register. zeros are copied into the right most bits.

```
reg[rc] = reg<u64>[rb] >> reg[ra]
```

3.1.27. sra

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11010**	**00**

sra.i64 *rc, rb, ra*

the *sra* or *shift-right-arithmetic* instruction performs an arithmetic right shift of the value in the *rb* register by the number of bits in register *ra* then saves the result in the *rc* register. sign is copied into the right most bits.

```
reg[rc] = reg<i64>[rb] >> reg[ra]
```

3.1.28. sll

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11011**	**00**

sll.i64 *rc, rb, ra*

the *sll* or *shift-left-logical* instruction performs a logical left shift of the value in the *rb* register by the number of bits in register *ra* then saves the result in the *rc* register.

```
reg[rc] = reg[rb] << reg[ra]
```

3.1.29. sub

15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11100**	**00**

sub.i64 *rc, rb, ra; flag*

the *sub* instruction subtracts the *ra* register from the *rb* register and saves the result in the *rc* register, and carry *not borrow* is saved to the *flag* register. **note:** this differs from borrow semantics; flag is *true carry* and can be used with *adc* or *sbc* instructions without inversion.

```
result = reg[rb] - reg[ra]
flag = reg[rb] >= reg[ra]
reg[rc] = result
```

3.1.30. mul

15 13	12 10	9 7	6 2	1 0	
rc	*rb*	*ra*	*opcode*	*size*	
rc	**rb**	**ra**	**11101**	**00**	**mul.i64** *rc*, *rb*, *ra*

the *mul* instruction performs signed multiplication of the *rb* register with the *ra* register and saves the result in the *rc* register.

```
reg[rc] = reg[rb] * reg[ra]
```

3.1.31. div

15 13	12 10	9 7	6 2	1 0	
rc	*rb*	*ra*	*opcode*	*size*	
rc	**rb**	**ra**	**11110**	**00**	**div.i64** *rc*, *rb*, *ra*

the *div* instruction performs signed division of the *rb* register by the *ra* register and saves the result in the *rc* register. division by zero causes the flag to be set and zero to be stored in the *rc* register. a subsequent branch can handle the division by zero.

```
flag = reg[ra] == 0
if flag:
    reg[rc] = 0
else:
    reg[rc] = reg[rb] / reg[ra]
```

3.1.32. illegal

15 7	6 2	1 0	
imm	*opcode*	*size*	
uimm	**11111**	**00**	**illegal** *uimm9*

the *illegal* instruction causes an illegal instruction trap. *program counter* and trap cause are saved to privileged registers for the operating system to dispatch to an illegal instruction handler and the *program counter* is set to a trap vector address.

4. Assembly

4.1. Introduction

this glyph assembly language reference begins with an introduction to assembler and linker concepts and terminology, followed by sections describing the glyph assembler directives, and pseudo-instruction aliases. section 3 contains a complete listing of instruction.

4.2. Concepts

this section covers assembler high level concepts required to understand the concepts involved in assembling and linking executable code from source files. the terminology used in this section is applicable to the PE/COFF, ELF and Mach-O file formats.

4.2.1. assembly

assembly files contains assembly language directives, macros and instructions describing program code and data. they can be handwritten or emitted by a compiler. an assembly file is the input file to the assembler and the output from the assembler is an object file.

4.2.2. object

object files contain compiled relocatable object code and data emitted by the assembler. an object file cannot be run, rather it is used as input to the linker in a program linking step which combines them to produce an executable file or shared library.

4.2.3. archive

archive files contain collections of relocatable object files and are typically referred to as static libraries. archive files use a simple format that appends together a set of object files with an index listing the object files contained in the archive. thin archive files just contain the index listing the object files without any associated data.

4.2.4. executable

executable files contain compiled relocatable object code and data that has been linked together by a linker in a program linking step using multple object files, archive files and shared libraries as input. executable files can be statically or dynamically linked. dynamically linked executables files express dependencies on shared libraries.

4.2.5. shared library

shared library files contain compiled relocatable object code and data that have been linked together by a linker in a program linking step using multple object files, archive files and shared libraries as input. shared libraries are dynamically linked by a runtime linker, and they express dependencies on other shared libraries which need to be loaded at runtime.

4.2.6. section

a section is a name for a region of code or data in an object file, executable file or shared library file. the files can be made up of multiple sections where each section corresponds to several types of executable code or data. this list contains the most common types:

- `.text` is a read-only section containing executable code
- `.const` is a read-only section containing immediate blocks
- `.data` is a read-write section containing global or static variables
- `.rodata` is a read-only section containing read-only variables
- `.bss` is a read-write section containing uninitialized data

4.2.7. segment

segments are loadable regions of code or data in an an executable or shared library. segments describe virtual addresses, file offsets and memory access permissions for mapped sections. in ELF, a segment can map to one or more sections. in PE/COFF sections are mapped directly. in Mach-O, sections are contained within a small set of specific segment types.

4.2.8. symbol

symbols are metadata table entries that contains a name with a mapping to an address. symbols are present in object files and dynamic symbols are present in shared libraries and executables. they can be referred to in relocation entries and debugging metadata.

4.2.9. relocation

relocations are metadata table entries used to update relocatable addresses during linking. relocations contains a type, a file offset pointing to text or data, and a pointer to a symbol name whose address needs to be updated during the linking step. relocations can be present in object files and dynamic relocations are present in shared libraries and executables.

4.2.10. program linking

program linking is the process of combining multiple relocatable object files by merging and aligning sections, resolving symbol references across files, assigning final symbol addresses, applying relocation fixups using relocation entries, and adding debug metadata.

4.3. Directives

the assembler implements a number of directives that control the assembly of instructions into object files. these directives are based on the AT&T System V assembler [2] and the GNU assembler [6], with some additions. they provide the ability to include arbitrary data, align data, export symbols, switch sections, define constants, and emit metadata.

the following table lists glyph assembler directives:

Directive	Arguments	Description
Data directives		
`.byte`	*expression-list*	8-bit comma separated words
`.short`	*expression-list*	16-bit comma separated words
`.long`	*expression-list*	32-bit comma separated words
`.quad`	*expression-list*	64-bit comma separated words
`.octa`	*expression-list*	128-bit comma separated words
`.string`	*"string"*	emit string
`.zero`	*integer*	emit zeroes
Alignment directives		
`.align`	*pow2 [,pad_val=0] [,max]*	align to power of 2
`.balign`	*bytes [,pad_val=0]*	byte align
Symbol directives		
`.globl`	*symbol_name,const_name*	emit symbol *(global scope)*
`.local`	*symbol_name,const_name*	emit symbol *(local scope)*
Section directives		
`.text`		emit `.text` section or make current
`.const`		emit `.const` section or make current
`.data`		emit `.data` section or make current
`.rodata`		emit `.rodata` section or make current
`.bss`		emit `.bss` section or make current
`.common`	*symbol_name,size,align*	emit common object to `.bss` section
`.section`	*section_name*	emit section (default `.text`) or make current
Miscellaneous directives		
`.equ`	*name, value*	constant definition
`.file`	*"filename"*	emit filename symbol
`.ident`	*"string"*	emit identification string
`.size`	*symbol,symbol*	emit symbol size
`.type`	*symbol,@function*	emit symbol type

Table 4.1.: Assembler directives

4.4. Pseudo-instructions

the assembler implements a number of convenience psuedo-instruction aliases that are formed from regular instructions, but have implicit or deduced arguments.

the following table lists glyph assembler pseudo instruction aliases:

Pseudo-instruction	**Expansion**	**Description**
`nop`	`or.i64 r0,r0,r0`	no-operation
`li` *rc, expression*	(several expansions)	load immediate
`la` *rc, symbol*	(several expansions)	load address
`call` *symbol*	`jalib.i64` *ibcall(text-label,const-label)*	procedure call
`ret`	`jtlib.i64` *ibret(text-label,const-label)*	procedure return
`jib.i64` *ib64(uimm6)*	`link.i64` *ib64(uimm6),* `jib`	jump-immmediate-block
`jalib.i64` *rc, ib64(uimm6)*	`link.i64` *rc, ib64(uimm6),* `jalib`	jump-and-link-immmediate-block
`jtlib.i64` *rc, ib64(uimm6)*	`link.i64` *rc, ib64(uimm6),* `jtlib`	jump-to-link-immmediate-block
`jalaib.i64` *rc, ib64(uimm6)*	`link.i64` *rc, ib64(uimm6),* `jalaib`	jump-and-link-add-immmediate-block
`cmp.lt.i64` *rc, rb*	`compare.i64` *rc, rb,* `lt`	compare less than (signed)
`cmp.gt.i64` *rc, rb*	`compare.i64` *rb, rc,* `lt`	compare greater than (signed)
`cmp.le.i64` *rc, rb*	`compare.i64` *rb, rc,* `ge`	compare less or equal (signed)
`cmp.ge.i64` *rc, rb*	`compare.i64` *rc, rb,* `ge`	compare greater or equal (signed)
`cmp.eq.i64` *rc, rb*	`compare.i64` *rc, rb,* `eq`	compare equal
`cmp.ne.i64` *rc, rb*	`compare.i64` *rc, rb,* `ne`	compare not equal
`cmp.ltu.i64` *rc, rb*	`compare.i64` *rc, rb,* `ltu`	compare less than (unsigned)
`cmp.gtu.i64` *rc, rb*	`compare.i64` *rb, rc,* `ltu`	compare greater than (unsigned)
`cmp.leu.i64` *rc, rb*	`compare.i64` *rb, rc,* `geu`	compare less or equal (unsigned)
`cmp.geu.i64` *rc, rb*	`compare.i64` *rc, rb,* `geu`	compare greater or equal (unsigned)
`cmov.i64` *rc, rb*	`compare.i64` *rc, rb,* `cmov`	conditional move
`ncmov.i64` *rc, rb*	`compare.i64` *rc, rb,* `ncmov`	negated conditional move
`mov.i64` *rc, rb*	`logic.i64` *rc, rb,* `mov`	copy register
`not.i64` *rc, rb*	`logic.i64` *rc, rb,* `not`	logical not
`neg.i64` *rc, rb*	`logic.i64` *rc, rb,* `neg`	signed negate
`bswap.i64` *rc, rb*	`logic.i64` *rc, rb,* `bswap`	byte swap
`ctz.i64` *rc, rb*	`logic.i64` *rc, rb,* `ctz`	count trailing zeros
`clz.i64` *rc, rb*	`logic.i64` *rc, rb,* `clz`	count leading zeroes
`ctpop.i64` *rc, rb*	`logic.i64` *rc, rb,* `ctpop`	count population
`sext.i64` *rc, rb*	`logic.i64` *rc, rb,* `sext`	sign extend

Table 4.2.: Pseudo instructions

4.5. Calling convention

4.5.1. calling convention — 16-bit

the 16-bit instruction packet, while intended to be used in conjunction with the larger opcodes, is designed as a complete subset, so there is an ABI variant that targets a subset of the instruction set architecture that only uses the 16-bit opcodes.

the register assignment for the 16-bit subset was chosen with this rationale:

- 2 blocks of 4 contiguous non-volatile *callee-save* and volatile *caller-save* registers.
- 3 special registers, 2 argument registers, 1 temporary register, and 3 save registers.
- 3 save registers to avoid excessive spilling around function calls.
- 1 temporary register to avoid spilling arguments to free a temporary.

the calling convention for the 16-bit subset is as follows:

- *immediate base* `ib` is set by `call` instructions and must point to a valid immediate block on function entry. function symbols are exported with two labels; one in the `.text` section, and one in the `.const` section. *immediate base* must be restored to the entry value in the function *epilogue* before it can be restored by `ret`.
- *argument registers* `a0` and `a1` are used for the first two arguments, and the remaining arguments are passed on the stack. *return value* is places in `a0` and `a1`, *temporary register* `t0` is a volatile register, and *frame pointer* (if enabled) uses `s0`. there are two more non-volatile *callee-save* registers, `s1` and `s2`.

the following table outlines the 16-bit register allocation showing register name alias, description, and non-volatile *callee-save* or volatile *caller-save* status.

name	**alias**	**description**	**save**
ib		immediate base	callee
r0	sp	stack pointer	callee
r1	s0/fp	saved register 0 / frame pointer	callee
r2	s1	saved register 1	callee
r3	s2	saved register 2	callee
r4	a0	argument register 0	caller
r5	a1	argument register 1	caller
r6	t0	temporary register 0	caller
r7	ra	return address / (pc,ib) link vector	caller

Table 4.3.: 16-bit register assignment

A. Appendix

A.1. Opcode summary — 16-bit

15 13 12 10 9 7 6 2 1 0

operand			*opcode*	*size*	
uimm			00000	00	**break** *uimm9*
simm			00001	00	**j** *simm9* × 2
simm			00010	00	**b** *simm9* × 2
simm			00011	00	**ibj** *simm9* × 64
fun	uimm		00100	00	**link.i64** *fun3*, *ib64*(*uimm6*)
rc	uimm		00101	00	**movh.i64** *rc*, *ib32*(*uimm6*)
rc	uimm		00110	00	**movw.i64** *rc*, *ib64*(*uimm6*)
rc	simm		00111	00	**movi.i64** *rc*, *simm6*
rc	simm		01000	00	**addi.i64** *rc*, *simm6*; *flag*
rc	uimm		01001	00	**srli.i64** *rc*, *uimm6*
rc	uimm		01010	00	**srai.i64** *rc*, *uimm6*
rc	uimm		01011	00	**slli.i64** *rc*, *uimm6*
rc	uimm		01100	00	**addh.i64** *rc*, *ib32*(*uimm6*); *flag*
rc	uimm		01101	00	**leapc.i64** *rc*, *ib32*(*uimm6*)(*pc*)
rc	uimm		01110	00	**loadpc.i64** *rc*, *ib32*(*uimm6*)(*pc*)
rc	uimm		01111	00	**storepc.i64** *rc*, *ib32*(*uimm6*)(*pc*)
rc	rb	uimm	10000	00	**load.i64** *rc*, (*uimm3* × 8)(*rb*)
rc	rb	uimm	10001	00	**store.i64** *rc*, (*uimm3* × 8)(*rb*)
rc	rb	fun	10010	00	**compare.i64** *rc*, *rb*, *fun3*
rc	rb	fun	10011	00	**logic.i64** *rc*, *rb*, *fun3*
rc	rb	ra	10100	00	**pin.i64** *rc*, *rb*, *ra*
rc	rb	ra	10101	00	**and.i64** *rc*, *rb*, *ra*
rc	rb	ra	10110	00	**or.i64** *rc*, *rb*, *ra*
rc	rb	ra	10111	00	**xor.i64** *rc*, *rb*, *ra*
rc	rb	ra	11000	00	**add.i64** *rc*, *rb*, *ra*; *flag*
rc	rb	ra	11001	00	**srl.i64** *rc*, *rb*, *ra*
rc	rb	ra	11010	00	**sra.i64** *rc*, *rb*, *ra*
rc	rb	ra	11011	00	**sll.i64** *rc*, *rb*, *ra*
rc	rb	ra	11100	00	**sub.i64** *rc*, *rb*, *ra*; *flag*
rc	rb	ra	11101	00	**mul.i64** *rc*, *rb*, *ra*
rc	rb	ra	11110	00	**div.i64** *rc*, *rb*, *ra*
uimm			11111	00	**illegal** *uimm9*

References

[1] Howard H. Aiken and Grace Hopper. *A Manual of Operation for the Automatic Sequence Controlled Calculator*. Tech. rep. Describes the Harvard Mark I architecture with dedicated instruction memory. Harvard University Computation Laboratory, 1946.

[2] AT&T Bell Laboratories. *UNIX® System V Release 4: Assembler and Machine-Level Debugging Guide*. Describes AT&T assembler directives and syntax used in UNIX® System V Release 4. Prentice Hall, 1990. ISBN: 0-13-947116-4.

[3] Paul Barham et al. "Xen and the Art of Virtualization". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY, USA: ACM, 2003, pp. 164–177. DOI: `10.1145/945445.945462`.

[4] J. Cocke and V. Markstein. "The Evolution of RISC Technology at IBM". In: *IBM Journal of Research and Development* 34.1 (1990). Reviews the IBM 801's architecture and its influence on later RISC systems, pp. 4–11. DOI: `10.1147/rd.341.0004`.

[5] DWARF Debugging Information Format Workgroup. *LEB128 (Little Endian Base 128) Encoding*. Section 7.6: Variable Length Data. Free Standards Group, 2006. URL: `https://dwarfstd.org/doc/Dwarf3.pdf`.

[6] Free Software Foundation. *Using as: The GNU Assembler*. Part of GNU Binutils. GNU Project. 1991. URL: `https://sourceware.org/binutils/docs/as/`.

[7] Advanced Micro Devices Inc. "Accessing an extended register set in an extended register mode". Patent US6877084B1. Filed 2001-04-02. Granted 2005-04-05. Expired 2023-06-18. Apr. 2005. URL: `https://patents.google.com/patent/US6877084B1`.

[8] Cray Research LLC. "Data processing system for processing one and two parcel instructions". Patent US5717881A. Filed 1995-06-07. Granted 1998-02-10. Expired 2015-02-10. Feb. 1998. URL: `https://patents.google.com/patent/US5717881A`.

[9] John von Neumann. *First Draft of a Report on the EDVAC*. Tech. rep. Seminal report introducing the stored-program concept, later known as the Von Neumann architecture. Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[10] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface*. 5th. Describes the classic 5-stage RISC pipeline: fetch, decode, execute, memory, write-back. Morgan Kaufmann, 2013. ISBN: 9780124077263.

[11] Ray J. Solomonoff. "A Formal Theory of Inductive Inference. Parts I and II". In: *Information and Control* 7.1-2 (1964). Introduces the three-tape Universal Prefix Turing Machine model, pp. 1–22, 224–254. DOI: `10.1016/S0019-9958(64)90223-2`.

[12] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society*. 2nd ser. 42 (1936). A theoretical model of computation, pp. 230–265. DOI: `10.1112/plms/s2-42.1.230`.