GLYPH-X

a super regular RISC architecture

June 23, 2025 – 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.

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 that is a reimplementation of existing ideas; rather, it introduces a novel architecture with an instruction coding scheme optimized for efficient parallel decoding in hardware or software, combined with contemporary practice in instruction semantics.

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 [11] 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 [3] 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.

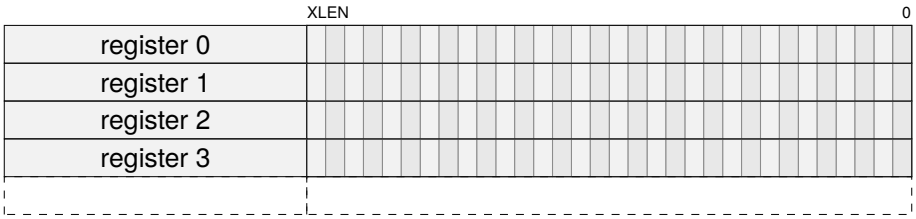


Figure 1.1.: organization of register storage.

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.

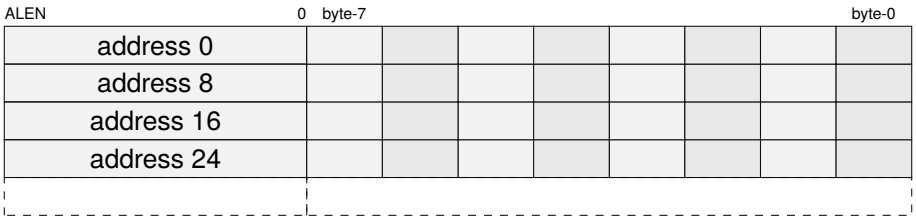


Figure 1.2.: organization of main-memory storage.

[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 [8], 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 1.4 provides details on the instruction template and instruction forms.

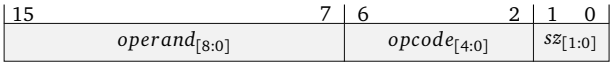


Figure 1.3.: 16-bit instruction containing one packet.

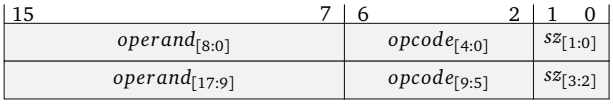


Figure 1.4.: 32-bit instruction containing two packets.

instruction words are packed together into instruction blocks containing instruction size information that is used to fuse them together to compose variable-length instructions [7]. instruction words can begin on any 16-bit aligned address. instruction memory is addressed with displacements from the *program counter* called *pc-relative* addresses.

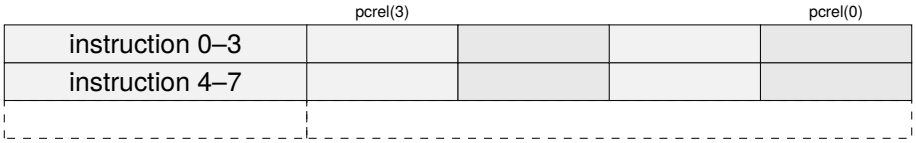


Figure 1.5.: organization of instruction-memory with 16-bit instructions.

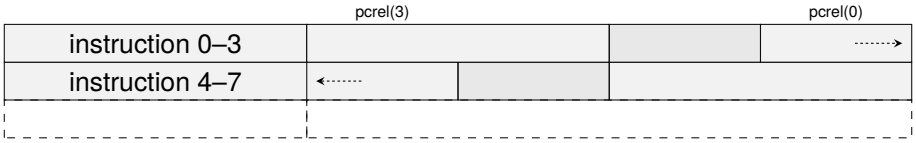


Figure 1.6.: organization of instruction-memory with 16-bit and 32-bit instructions.

1.1.5. constants

constant memory [10] 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 1.3 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.*

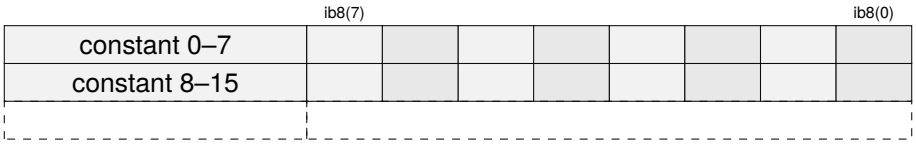


Figure 1.7.: organization of *ib8* constant-memory storage.

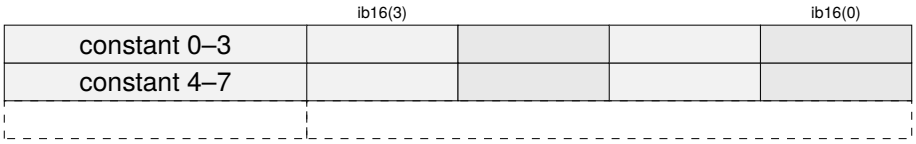


Figure 1.8.: organization of *ib16* constant-memory storage.

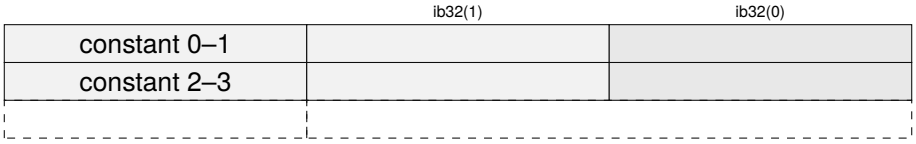


Figure 1.9.: organization of *ib32* constant-memory storage.

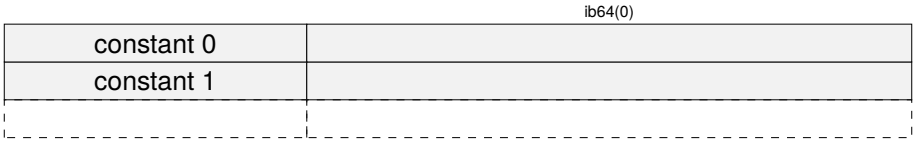


Figure 1.10.: organization of *ib64* constant-memory storage.

1.2. 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, 64, and 128-bit instructions.
- 16-bit compressed instruction packets 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 an unsigned displacement to access constants.
- `jalib` *jump-and-link-immediate-block* or *call link function* links address vector and adds constants to *(pc,ib)* and 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 *return link function* subtracts link vector from and adds constants to *(pc,ib)* and is used to branch the *program counter* and *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*.

1.3. Program streams

glyph seperates the stored programs into two streams, one with instructions and one with constants [10]. 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.

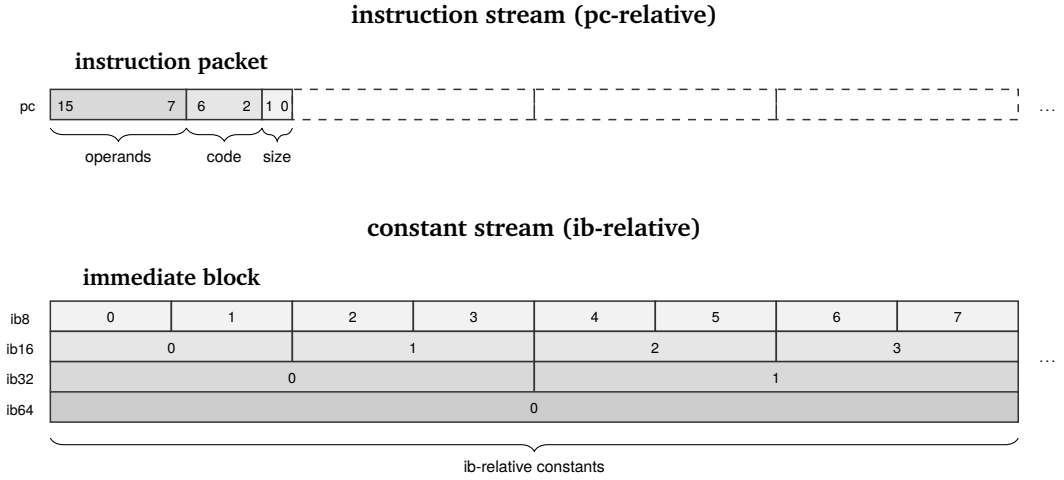


Figure 1.11.: program counter and immediate base register.

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.

1.4. Instruction format

instructions use a variable length format [7] 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 [6]. instruction words can begin on any 16-bit aligned address.

1.4.1. instruction templates

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

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

Figure 1.12.: instruction template — 16-bit.

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

Figure 1.13.: instruction template — 32-bit.

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

Figure 1.14.: instruction template — 64-bit.

15 7	6 2	1 0
$operand_{[8:0]}$	$opcode_{[4:0]}$	$sz_{[1:0]}$
$operand_{[17:9]}$	$opcode_{[9:5]}$	$sz_{[3:2]}$
$operand_{[26:18]}$	$opcode_{[14:10]}$	$sz_{[5:4]}$
$operand_{[35:27]}$	$opcode_{[19:15]}$	$sz_{[7:6]}$
$operand_{[44:36]}$	$opcode_{[24:20]}$	$sz_{[9:8]}$
$operand_{[53:45]}$	$opcode_{[29:25]}$	$sz_{[11:10]}$
$operand_{[62:54]}$	$opcode_{[34:30]}$	$sz_{[13:12]}$
$operand_{[71:63]}$	$opcode_{[39:35]}$	$sz_{[15:14]}$

Figure 1.15.: instruction template — 128-bit.

1.4.2. instruction sizes

the variable length instruction format has a *2-bit* size field inspired by LEB128 [4] in a fixed position in every instruction packet, to reduce the complexity of size-decoding for variable length instructions[7]. 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}
128-bit	{11,11,11,11,11,11,11,11}

Table 1.1.: variable-length instruction size fields

1.4.3. instruction forms

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.

16-bit instruction forms

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

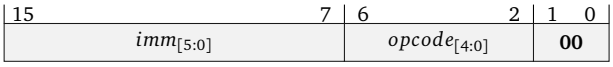


Figure 1.16.: 16-bit large immediate.

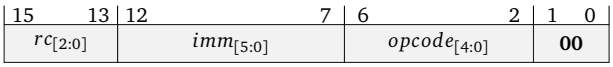


Figure 1.17.: 16-bit one operand with immediate.

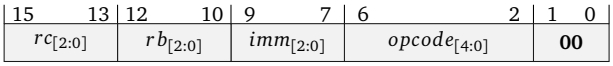


Figure 1.18.: 16-bit two operand with immediate.

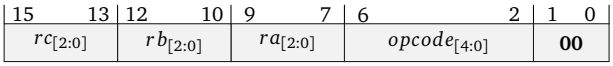


Figure 1.19.: 16-bit three operand.

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

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**

Figure 1.20.: 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**

Figure 1.21.: 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**

Figure 1.22.: 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**

Figure 1.23.: 32-bit three operand.

1.4.4. 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 1.2.: instruction decode combinations

1.5. Register file

the glyph register file is extensible due to the variable length instruction format and supports a different number of registers depending on the instruction size.

- 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.

the register state accessible by the 16-bit instruction packet is comprised of:

- *program counter* register (aligned to 2 bytes).
- *immediate base* register (aligned to 64 bytes).
- 8 ×general purpose registers (*r0* through *r7*).
- 1 ×predicate register (*flag*).

the following diagram shows the register state accessible by the 16-bit instruction packet:

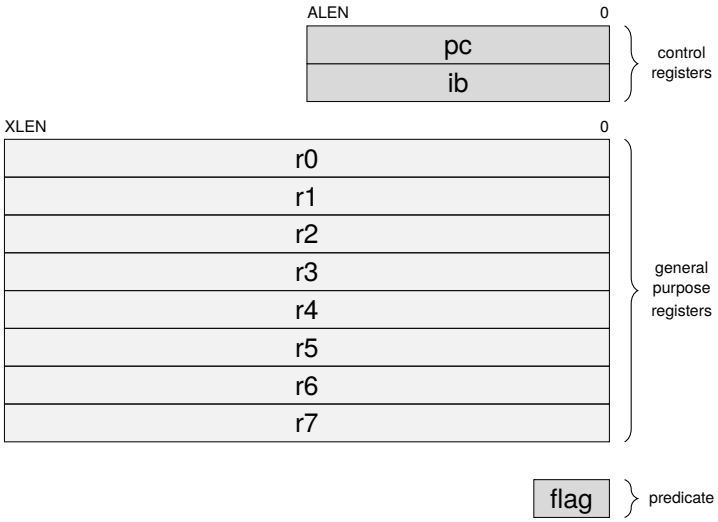


Figure 1.24.: register state accessible by 16-bit instruction packet.

the use of *ALEN*[8] and *XLEN*[9] parameters is to indicate that the width of addresses can be less than the width of the general purpose registers.

[8] *ALEN* refers to the width of addresses in bits.
[9] *XLEN* refers to the width of general purpose registers in bits.

1.6. Example pipeline

an illustrative micro-architecture is proposed based on the classic 5-stage RISC micro-architecture [9] 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.

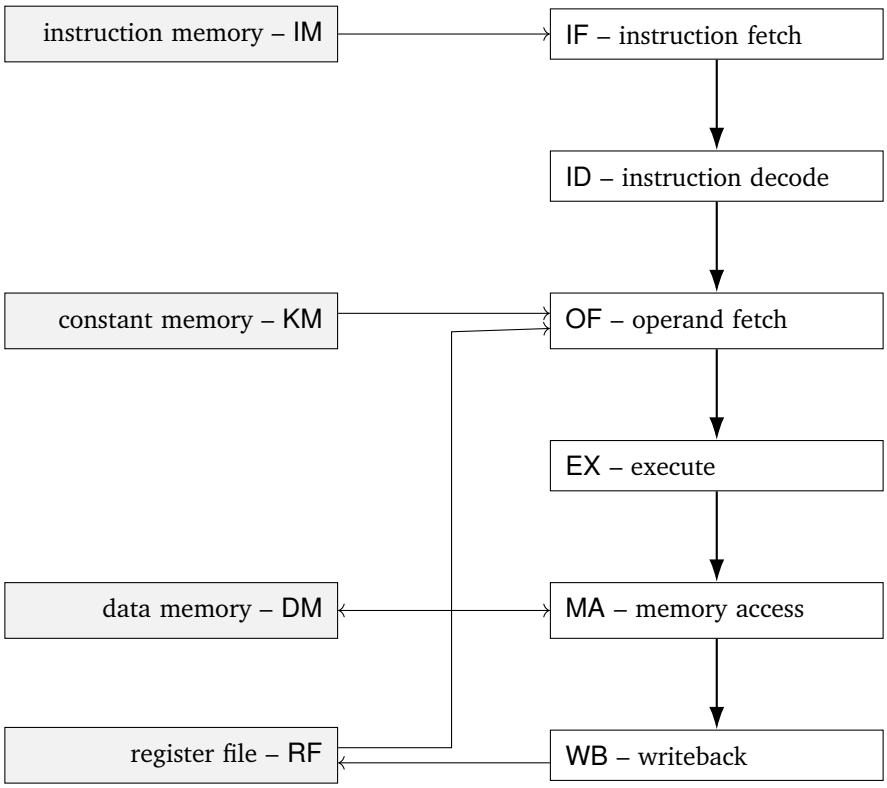


Figure 1.25.: sample 6-stage micro-architecture with support for constant memory.

2. System

2.1. System registers

the architecture provides several control and status registers for floating-point control and status, time, trap handling, address translation, timers, inter-thread interrupts, and debug.

no.	name	description
user-level registers		
0x00	fpcontrol	floating-point control
0x01	fpstatus	floating-point status
0x02	ctime	clock time
0x03	cfreq	clock frequency
privileged trap registers		
0x10	tcontrol	trap control
0x11	tstatus	trap status
0x12	tscratch	trap scratch
0x13	tepc	trap exception program counter
0x14	teib	trap exception immediate base
0x15	tval	trap value
0x16	tcause	trap cause
0x17	tack	trap acknowledge
0x18	thpc	trap handler program counter
0x19	thib	trap handler immediate base
privileged system registers		
0x20	scontrol	system control
0x21	sstatus	system status
0x22	sfeature	system feature
0x23	sptr	system page table root
0x24	saddr	system thread address
0x25	starget	system target address
0x26	smessage	system message interrupt
0x27	stimer	system deadline timer
0x28	sie	system interrupt enable
0x29	sip	system interrupt pending
privileged debug registers		
0x31	dstatus	debug status
0x32	dcycle	debug cycle counter
0x33	dinst	debug instruction counter
0x34	dstop	debug stop instruction
0x35	dfetch	monitored fetch address
0x36	dread	monitored read address
0x37	dwrite	monitored write address

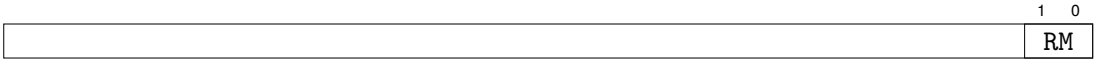
Table 2.1.: system registers

2.1.1. user-level registers

the section describes the user-level registers.

floating-point control (`fpcontrol`)

`fpcontrol` contains control information for floating-point operations. the `RM` field is a read-write and contains the current floating-point rounding mode.

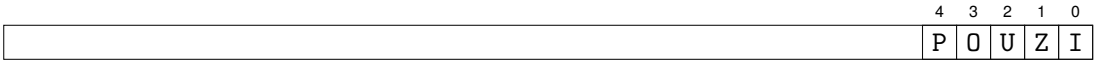


no.	name	description
0	RN	round to nearest (even)
1	RD	round down (towards $-\infty$)
2	RU	round up (towards $+\infty$)
3	RZ	round towards zero (truncate)

Table 2.2.: floating-point control round mode field

floating-point status (`fpstatus`)

`fpstatus` contains status information for floating-point operations. the `I, Z, U, O,` and `P` fields are read-write and contain accrued floating-point exceptions.



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

Table 2.3.: floating-point status flags

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`.

clock frequency (`cfreq`)

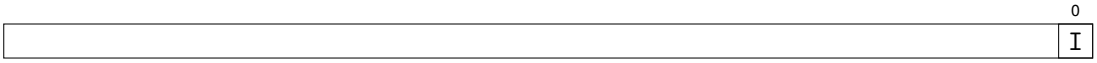
`cfreq` is a read-only register containing the wall-clock tick interval in picoseconds; $\frac{10^{12}}{f}$ where f is the frequency in Hertz (Hz).

2.1.2. privileged trap registers

the section describes the privileged trap registers.

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 for this hardware thread.

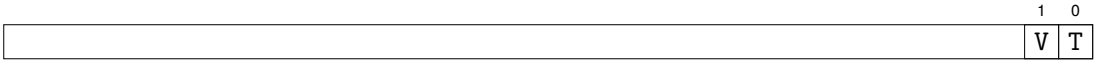


no.	name	description
`1 << 0`	I	system interrupt enable

Table 2.4.: trap control fields

trap status (`tstatus`)

`tstatus` contains status bits for the current trap. the `T` field is read-only and indicates whether a timer interrupt is pending. the `V` field is read-only and indicates whether a virtual interrupt is pending.



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

Table 2.5.: trap status fields

trap scratch (`tscratch`)

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

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 value (`tval`)

`tval` is a read-only register with a word indicating the identity of the trap, and may contain:

- the faulting instruction word for *break* and *illegal instruction exceptions*.
- the faulting address for *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, and may contain:

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
system interrupts		
30	timer-interrupt	timer interrupt
31	virtual-interrupt	virtual interrupt
32	interrupt-0	interrupt pin 0
...	...	...
63	interrupt-31	interrupt pin 31

Table 2.6.: 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 handler program counter (`thpc`)

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

trap handler immediate base (`thib`)

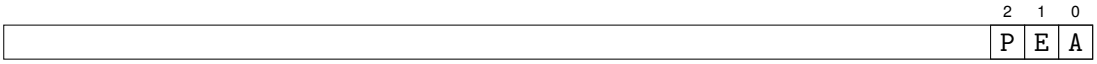
`thib` is a read-write register containing the address of the system trap handler constants.

2.1.3. privileged system registers

the section describes the privileged system registers.

system control (`scontrol`)

`scontrol` is a read-write register containing system control information. the `A` field controls whether the user pages can be read or written by the supervisor. the `E` field controls whether user pages can be executed by the supervisor. the `P` field controls whether page table pages need physical maps and translate permissions. if enabled, then physical addresses must be self-mapped with the `T` permission set on entries for pages that contain page tables.

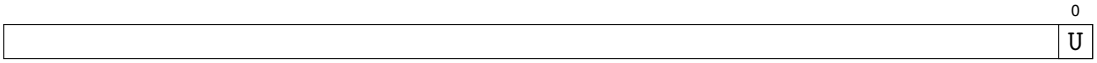


no.	name	description
1 << 0	A	supervisor access to user pages disabled
1 << 1	E	supervisor execute of user pages disabled
1 << 2	P	physical permissions feature enabled

Table 2.7.: system control fields

system status (`sstatus`)

`sstatus` is a read-only register containing system status. the `U` field indicates the processor is operating in user-mode. this register cannot be read in user-mode so it will return zero.

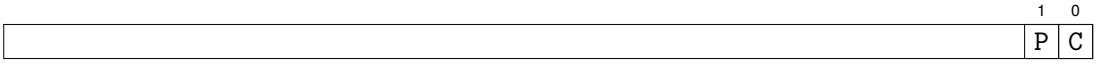


no.	name	description
1 << 0	U	user mode

Table 2.8.: system status fields

system feature (`sfeature`)

`sfeature` is a read-only register containing system features. the `C` field indicates the capabilities feature is present. the `P` field indicates the physical permissions feature is present.

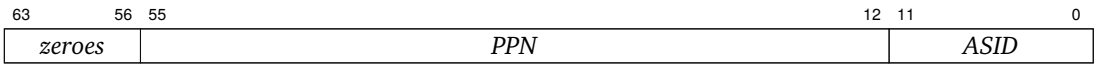


no.	name	description
1 << 0	C	capabilities check feature
1 << 0	P	physical permissions feature

Table 2.9.: system feature fields

system page table root (sptr)

sptr is a read-write register containing the page table root for address translation on this hardware thread. the *ASID* field contains the address space identifier for the current thread. the *PPN* field contains the physical page number of the page table root structure.



system address (saddr)

saddr is a read-only register containing the unique system-wide address for this hardware thread. the system-wide address is used as the target for inter-thread virtual interrupts.

system target (starget)

starget is a read-write register containing the system-wide address for a hardware thread that is the target of an edge-triggered virtual interrupt. there are two special addresses:

- a target address of all-zeros is the boot service processor.
- a target address of all-ones is the broadcast address for all threads.

system message (smessage)

smessage is a write-only register that causes an edge-triggered virtual interrupt to be queued to the hardware thread set in starget. virtual interrupts are marked pending in the V field of the tstatus register. if interrupts are not enabled on the target hardware thread, the virtual interrupt is delivered when interrupts are next enabled.

system timer (stimer)

stimer is a read-write register containing the deadline timer for this hardware 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. if interrupts are not enabled on this hardware thread, the timer interrupt is delivered when interrupts are next enabled.

system interrupt enable (sie)

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

system interrupt pending (sip)

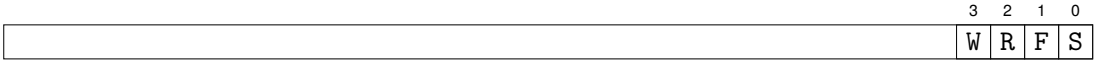
sip is a read-only register that contains interrupt pending flags for 32 wired interrupt pins.

2.1.4. 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 2.10.: 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 (`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 (`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 (`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.

2.2. 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 supervisor virtual addresses as well as physical addresses.

2.2.1. 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 2.11.: 64-bit page table dimensions

2.2.2. 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 along with several permission and metadata bits. the physical page number either points to the next page table level for *pointer* entries or the translated physical address for *leaf* entries.

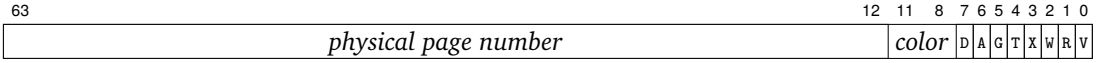


Figure 2.1.: 64-bit page table entry structure.

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 2.12.: 64-bit page table entry fields

2.2.3. 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 supervisor virtual addresses as well as physical addresses. the page table translation and lookup virtual address structure is as follows:

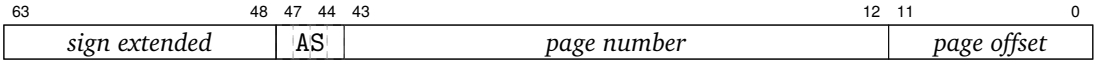


Figure 2.2.: 64-bit translation address structure.

no.	name
0b0.	user
0b10.	supervisor
0b110.	physical

Table 2.13.: 1—4 bit translation address space prefixes.

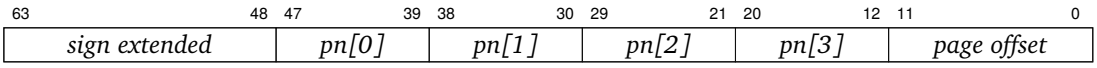


Figure 2.3.: 64-bit lookup virtual address structure.

2.2.4. 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 *sfeature.P* is present and *scontrol.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

```
1: function FIND_PAGE(type, level, ppn, ta)
2:     shift ← level × pn_bits + po_bits                          ▷ calculate page number shift
3:     tpn ← (ta >> shift) ∧ ((1 << pn_bits) − 1)                 ▷ calculate translation page number
4:     pte ← LOAD((ppn << po_bits) + tpn × sizeof(PTE))           ▷ load page table entry
5:     leaf ← pte.R ∨ pte.W ∨ pte.X                               ▷ page table entry node type
6:     sm ← PTE_R ∨ PTE_W ∨ PTE_X                                 ▷ default self map page table entry
7:     if sfeature.P ∧ scontrol.P ∧ type = virtual then
8:         pa ← pte.ppn << po_shift
9:         (sm, level) ← FIND_PAGE(physical, num_levels − 1, sptr.ppn, pa)   ▷ load self mapping
10:        if (leaf ∧ pte.ppn ≠ sm.ppn) ∨ (¬leaf ∧ ¬sm.T) then
11:            raise fault                                        ▷ invalid self mapping
12:        end if
13:    end if
14:    if ¬leaf ∧ level > 0 then
15:        return FIND_PAGE(type, level − 1, pte.ppn, ta)         ▷ follow pointer entry
16:    else if ¬leaf then
17:        raise fault                                            ▷ leaf entry not found
18:    end if
19:    return (pte, sm, level)                                    ▷ return page table entry
20: end function
```

Algorithm 2 translate virtual address to physical address

```
1: function TRANSLATE(op, va)
2:     if ¬CHECK_CANONICAL(va, va_bits) then                      ▷ check address is sign-extended
3:         raise fault
4:     end if
5:     (pte, sm, level) ← FIND_PAGE(virtual, num_levels − 1, sptr.ppn, va)   ▷ find page table entry
6:     mask ← ((1 << (level × pn_bits + po_bits)) − 1)
7:     caps ← colorperms[pte.color]
8:     user ← ¬((va >> (va_bits − 1)) ∧ 1)
9:     if (¬pte.V) ∨ (pte.W ∧ ¬pte.R) then
10:        raise fault                                            ▷ invalid write must have read
11:    else if (op.R ∧ ¬pte.R) ∨ (op.W ∧ ¬pte.W) ∨ (op.X ∧ ¬pte.X) then
12:        raise fault                                            ▷ invalid permissions
13:    else if (sm.R ∧ ¬pte.R) ∨ (sm.W ∧ ¬pte.W) ∨ (sm.X ∧ ¬pte.X) then
14:        raise fault                                            ▷ invalid self map
15:    else if (cap.R ∧ pte.R) ∨ (cap.W ∧ pte.W) ∨ (cap.X ∧ pte.X) then
16:        raise fault                                            ▷ invalid capabilities
17:    else if (op.R ∧ ¬pte.A) ∨ (op.W ∧ ¬pte.D) then
18:        raise fault                                            ▷ invalid accessed or dirty
19:    else if ¬sstatus.U ∧ ¬scontrol.E ∧ user ∧ op.X then
20:        raise fault                                            ▷ invalid user page execute
21:    else if ¬sstatus.U ∧ ¬scontrol.A ∧ user ∧ op.(R ∨ W) then
22:        raise fault                                            ▷ invalid user page access
23:    else if (pte.ppn << po_bits) ∧ mask ≠ 0 then
24:        raise fault                                            ▷ invalid superpage alignment
25:    end if
26:    return (pte.ppn << po_bits) + (va ∧ mask)
27: end function
```

2.3. Capabilities

the architecture provides an optional capabilities extension that includes several control and status registers for permissions and capabilities enabled by *color* bits in page table entries.

no.	name	description
capability registers		
0x40	colorperms	color permissions
0x41	colorcaps	color capabilities
capability matrix		
0x50	colormatrix0	color matrix for color 0
...	...	...
0x5f	colormatrix15	color matrix for color 15

Table 2.14.: capability registers

color permissions (`colorperms`)

`colorperms` is a read-write register containing an array of 16 elements holding 4 page table permission mask bits (negated). when set these bits mask page table permission bits based on the page table entry *color* field. setting bits masks permissions. the default of zero as unmasked is to provide full access by default for non capability aware system software.

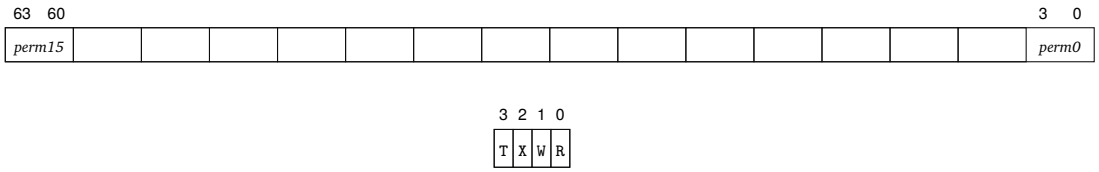


Figure 2.4.: color permission structure.

no.	mnem	name	description
1 << 0	R	~read	read permission for *color-x*
1 << 1	W	~write	write permission for *color-x*
1 << 2	X	~execute	execute permission for *color-x*
1 << 3	P	~physical	physical permission for *color-x*

Table 2.15.: color permission descriptions.

color capabilities (colorcaps)

colorcaps is a read-write register containing an array of 16 elements holding 4 capability bits (negated). when set these bits mask access to the *debug, system, capabilites,* and *hypervisor* sets of registers and instructions from program text pages based on the page table entry *color* field. setting bits masks capabilities. the default of zero as unmasked is to provide full access by default for non capability aware system software.

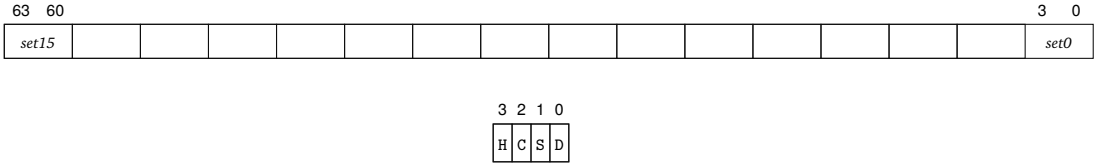


Figure 2.5.: color capabilities structure.

no.	mnem	name	description
1 << 0	D	~debug	debug access from *color-x*
1 << 1	S	~system	system access from *color-x*
1 << 2	C	~capability	capability access from *color-x*
1 << 3	H	~hypervisor	hypervisor access from *color-x*

Table 2.16.: color permission descriptions.

color matrix (colormatrix0...colormatrix15)

colormatrix0...colormatrix15 are read-write registers containing arrays of 16 elements holding 4 (negated) permission bits. when set these bits mask page table permission bits or branch execute permissions from executable pages based on the source page table entry *color-x*, with source color in the register name, for memory access or branch to target pages with page table entry *color-y*, with target color in the array index within each register.

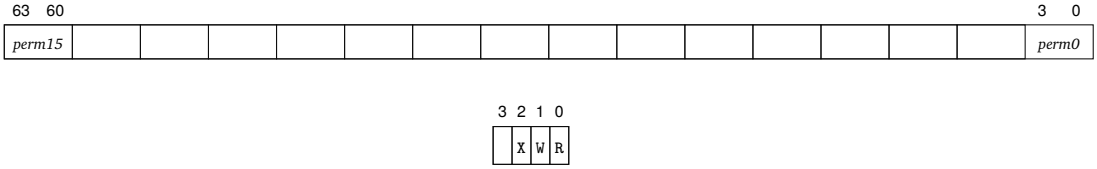


Figure 2.6.: color matrix structure.

no.	mnem	name	description
1 << 0	R	~read	load permission from *color-x* to *color-y*
1 << 1	W	~write	store permission from *color-x* to *color-y*
1 << 2	X	~execute	branch permission from *color-x* to *color-y*

Table 2.17.: color matrix desciptions.

3. Instructions

3.1. Instruction listing — 16-bit

3.1.1. break

15 … 7	6 … 2	1 … 0
imm9	*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
imm9	*opcode*	*size*
simm	**00001**	**00**

j $simm9 \times 2$

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
imm9	*opcode*	*size*
simm	**00010**	**00**

b $simm9 \times 2$

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
imm9	*opcode*	*size*
simm	**00011**	**00**

ibj $simm9 \times 64$

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	*imm6*	*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:

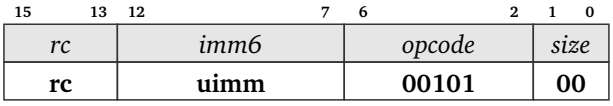
value	mnemonic	description	link-type	dst-link	src-link
0	jib	jump	jump	-	-
1	-	-	-	-	-
2	jalib	jump-and-link	call	r6	-
3	jalib	jump-and-link	call	r7	-
4	jtlib	jump-to-link	ret	-	r6
5	jtlib	jump-to-link	ret	-	r7
6	jalaib	jump-and-link-add	tail	r6	-
7	jalaib	jump-and-link-add	tail	r7	-

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



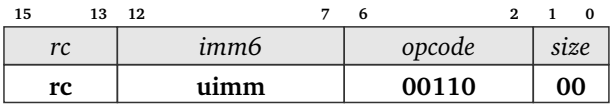
15–13	12–7	6–2	1–0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**00101**	**00**

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



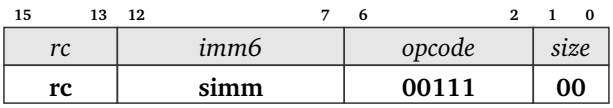
15–13	12–7	6–2	1–0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**00110**	**00**

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



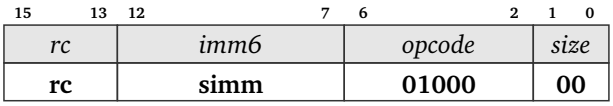
15–13	12–7	6–2	1–0
rc	*imm6*	*opcode*	*size*
rc	**simm**	**00111**	**00**

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



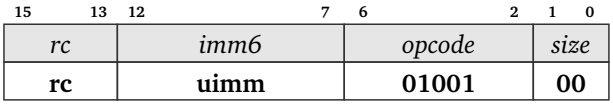
15–13	12–7	6–2	1–0
rc	*imm6*	*opcode*	*size*
rc	**simm**	**01000**	**00**

addi.i64 $rc, simm6$

the *addi* or *add-immediate* instruction sign-extends the immediate value in *simm6* then adds it to the *rc* register.

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

3.1.10. srli



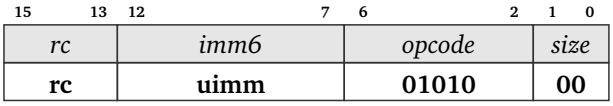
15 13	12 7	6 2	1 0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**01001**	**00**

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



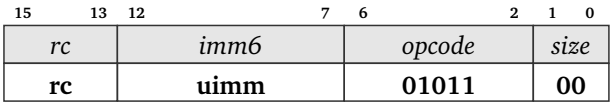
15 13	12 7	6 2	1 0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**01010**	**00**

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



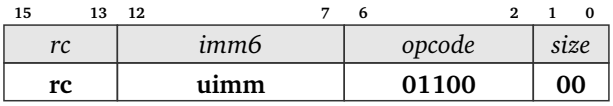
15 13	12 7	6 2	1 0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**01011**	**00**

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	*imm6*	*opcode*	*size*
rc	**uimm**	**01100**	**00**

addh.i64 $rc, ib32(uimm6)$

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.

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

3.1.14. leapc

15 13	12 7	6 2	1 0
rc	*imm6*	*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	*imm6*	*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	*imm6*	*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*	*imm3*	*opcode*	*size*
rc	**rb**	**uimm**	**10000**	**00**

load.i64 $rc, (uimm3 \times 8)(rb)$

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*	*imm3*	*opcode*	*size*
rc	**rb**	**uimm**	**10001**	**00**

store.i64 $rc, (uimm3 \times 8)(rb)$

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*	*imm3*	*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*	*imm3*	*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

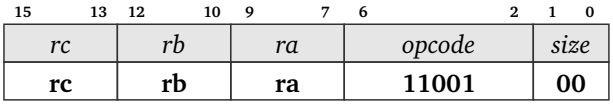
15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11000**	**00**

add.i64 rc, rb, ra

the *add* instruction adds the *rb* register to the *ra* register and saves the result in the *rc* register.

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

3.1.26. srl



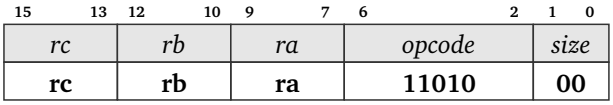
15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11001**	**00**

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



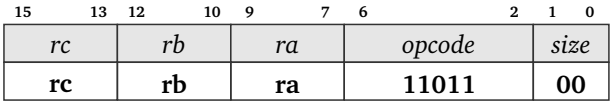
15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11010**	**00**

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



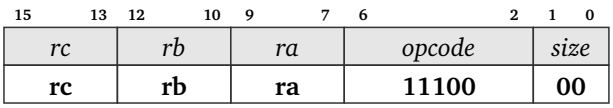
15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11011**	**00**

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

the *sub* instruction subtracts the *ra* register from the *rb* register and saves the result in the *rc* register.

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

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	
imm9	*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 [5], 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** $simm9 \times 2$
simm			**00010**	**00**	**b** $simm9 \times 2$
simm			**00011**	**00**	**ibj** $simm9 \times 64$
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** $rc, simm6$
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** $rc, ib32(uimm6)$
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** $rc, (uimm3 \times 8)(rb)$
rc	**rb**	**uimm**	**10001**	**00**	**store.i64** $rc, (uimm3 \times 8)(rb)$
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
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
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] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.