GLYPH-X

a super regular RISC architecture

Michael Clark

May 4, 2025 DRAFT

Contents

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[3] 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
[3] Alan M. Turing, Proceedings of the London Mathematical Society, Series 2, Volume 42, pp. 230–265.

1.1.1. load-store

a *load-store* architecture 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 $MMIO$[4] 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[5] 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.

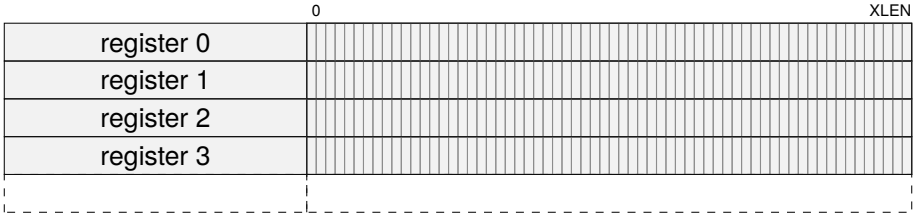


Figure 1.1.: organization of register storage.

1.1.3. memory

main memory is primary storage which in modern computers is most likely $DRAM$[6]. 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.

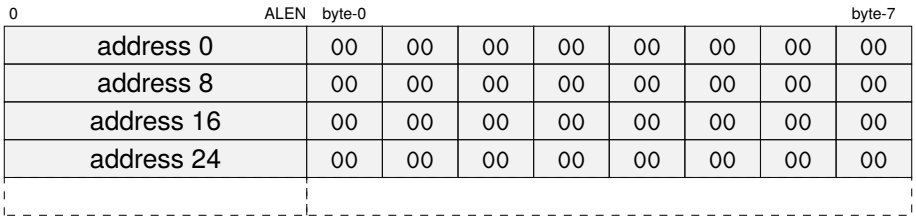


Figure 1.2.: organization of main-memory storage.

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

1.2. Overview

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[7].

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.
- `lib` *load-immediate-block* uses an unsigned displacement to access constants.
- `jalib` *jump-and-link-immediate-block* or *call* 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 *ret* 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.

[7] 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 $XLEN > ALEN \times 2$ it is possible to pack absolute addresses instead of relative addresses, as would be the case where *ALEN=64* and *XLEN=128*.

1.3. Instruction format

glyph has a variable length instruction format supporting 16, 32, 64, and 128-bit instruction packets. the instruction packet has been designed to use a *super regular* scheme, whereby successive instruction packets extend the fields in the previous packet.

1.3.1. 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 1.3.: 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 1.4.: 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 1.5.: 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 1.6.: instruction template — 128-bit.

1.3.2. instruction size encoding

the variable length instruction format has a *2-bit* size field in a fixed position in every 16-bit instruction packet, somewhat inspired by LEB128, to reduce the complexity of variable length instruction size decoding.

Instruction Size	Size Fields
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

1.3.3. instruction forms — 16-bit

the 16-bit 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 easier in software. the scheme is designed so that *1-bit* of coding space in the larger packet can be used to extend register sizes for the 16-bit ops.

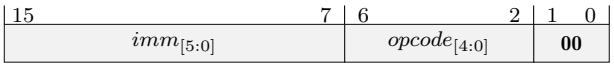


Figure 1.7.: 16-bit large immediate.

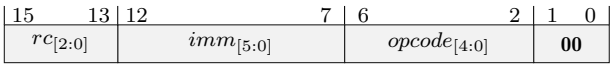


Figure 1.8.: 16-bit one operand with immediate.

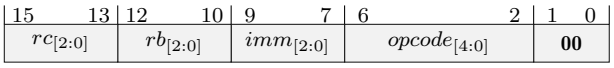


Figure 1.9.: 16-bit two operand with immediate.

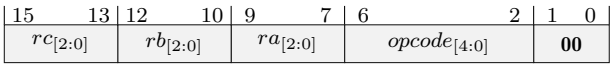


Figure 1.10.: 16-bit three operand.

1.4. Constant stream

glyph seperates the instruction stream into two streams, one with instructions and one with constants. the instruction stream is addressed with the *program counter (pc)* and the constant stream is addressed with the *immediate base (ib)* register.

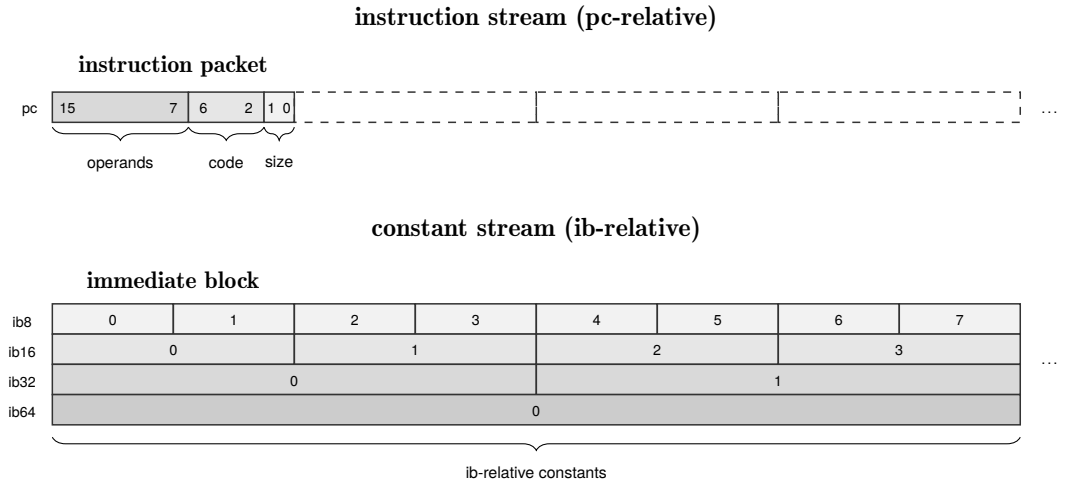


Figure 1.11.: program counter and immediate base register.

immediate blocks are aligned memory blocks addressed by the *immediate base register*.

immediate blocks can be navigated by branching the constant stream independently using the *immediate-block-jump* instruction, or together with the *program counter* using procedure call and return instructions that set and restore *(pc,ib)* via a link register that contains a packed relative address vector. the use of packed relative address vectors is for backward compatibility with a single link register.

for procedure calls and returns, the instruction and constant streams are set at the same time using the call and return instructions; *jump-and-link-immediate-block*, *jump-to-link-immediate-block*, which add and subtract relative address vectors to *(pc,ib)*, the *program counter* and the *immediate base* register. the *pack-indirect* instruction allows absolute *(pc,ib)* addresses to be packed into a relative address vector for indirect calls.

the instruction forms only use bonded register 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 with short references encoded inside of register slots for instructions that use an immediate block relative addressing mode.

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:

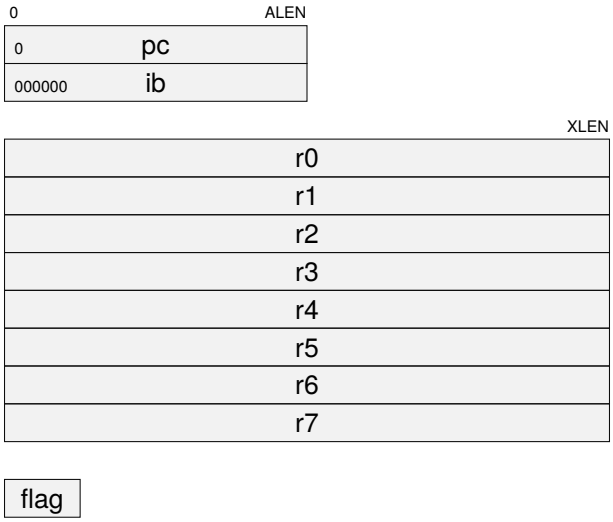


Figure 1.12.: register state accessible by 16-bit instruction packet.

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 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.

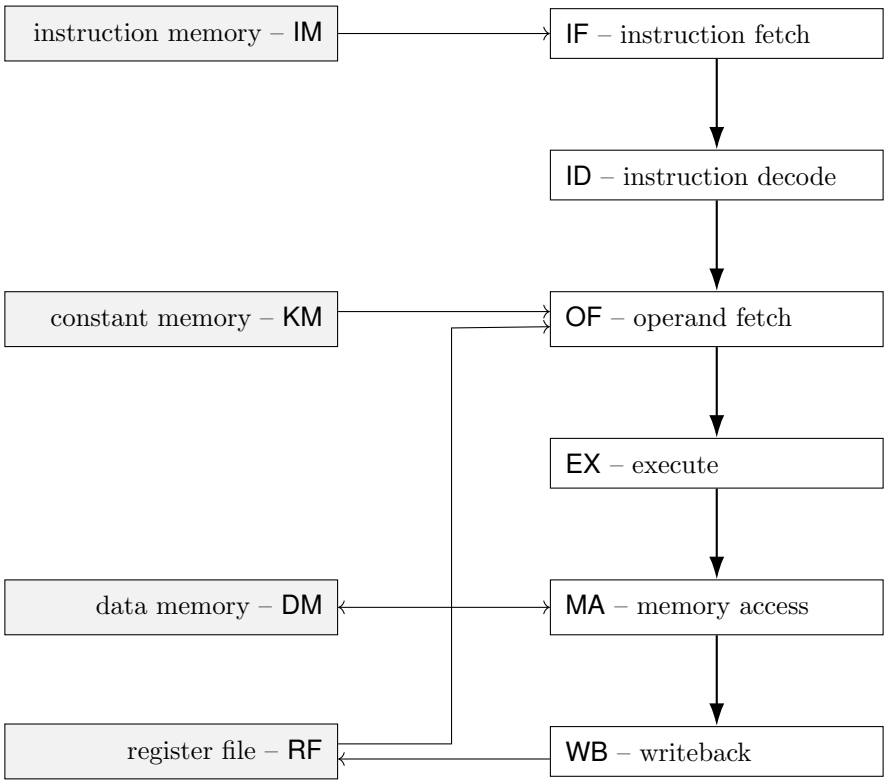


Figure 1.13.: sample 6-stage micro-architecture with support for constant memory.

2. Instructions

2.1. Instruction listing — 16-bit

2.1.1. 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.

2.1.2. j

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

j simm9

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 $[pc + simm9 \times 2 + 2]$.

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

2.1.3. b

15 … 7	6 … 2	1 … 0
imm9	*opcode*	*size*
simm	**00010**	**00**

b simm9

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 $[pc + simm9 \times 2 + 2]$, otherwise the *program counter* is advanced normally.

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

2.1.4. ibj

15 … 7	6 … 2	1 0
imm9	*opcode*	*size*
simm	**00011**	**00**

ibj simm9

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 += simm9 * 64
```

2.1.5. jalib.i64

15 … 13	12 … 7	6 … 2	1 0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**00100**	**00**

jalib rc, ib64(uimm6*8)

the *jalib* or *jump-and-link-immediate-block* 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)*, then saves the relative address vector in the *rc* register.

```
tmp = (i32x2)[ib + uimm6 * 8]
{rpc,rib} = (i32x2)tmp
pc += rpc + 2
ib += rib
r[rc] = tmp
```

2.1.6. jtlib.i64

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

jtlib rc, ib64(uimm6*8)

the *jtlib* or *jump-to-link-immediate-block* instruction loads a 64-bit constant addressed by $[ib + uimm6 \times 8]$ containing a *i32x2* relative address vector, then subtracts the *i32x2* relative address vector in the *rc* register from it, and adds the result to *(pc,ib)*.

```
{rpc,rib} = (i32x2)r[rc]
{dpc,dib} = (i32x2)[ib + uimm6 * 8]
pc += dpc - rpc
ib += dib - rib
```

2.1.7. movib.i64

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

movib.i64 rc, ib64(uimm6*8)

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

```
r[rc] = (i64)[ib + uimm6 * 8]
```

2.1.8. movi.i64

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.

```
r[rc] = simm6
```

2.1.9. addi.i64

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.

```
r[rc] += simm6
```

2.1.10. srli.i64

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.

```
r[rc] = (u64)r[rc] >> uimm6
```

2.1.11. srai.i64

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.

```
r[rc] = (i64)r[rc] >> uimm6
```

2.1.12. slli.i64

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.

```
r[rc] = r[rc] << uimm6
```

2.1.13. addib.i64

15 13	12 7	6 2	1 0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**01100**	**00**

addib.i64 rc, ib32(uimm6*4)

the *addib* or *add-immediate-block-constant* 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.

```
r[rc] += (i32)[ib + uimm6 * 4]
```

2.1.14. leapc.i64

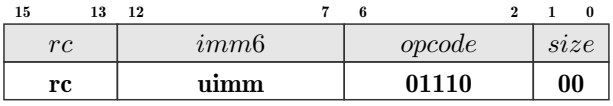
15 13	12 7	6 2	1 0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**01101**	**00**

leapc.i64 rc, ib32(uimm6*4)(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.

```
r[rc] = pc + (i32)[ib + uimm6 * 4]
```

2.1.15. loadpc.i64



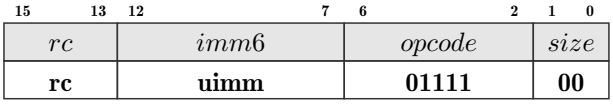
15–13	12–7	6–2	1–0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**01110**	**00**

loadpc.i64 rc, ib32(uimm6*4)(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.

```
r[rc] = (i64)[pc + (i32)[ib + uimm6 * 4]]
```

2.1.16. storepc.i64



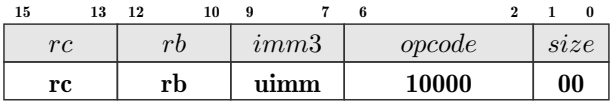
15–13	12–7	6–2	1–0
rc	*imm6*	*opcode*	*size*
rc	**uimm**	**01111**	**00**

storepc.i64 rc, ib32(uimm6*4)(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.

```
(i64)[pc + (i64)[ib + uimm6 * 4]] = r[rc]
```

2.1.17. load.i64



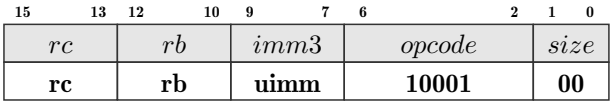
15–13	12–10	9–7	6–2	1–0
rc	*rb*	*imm3*	*opcode*	*size*
rc	**rb**	**uimm**	**10000**	**00**

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.

```
r[rc] = (i64)[r[rb] + uimm3 * 8]
```

2.1.18. store.i64



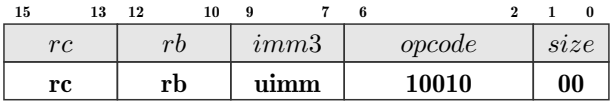
15–13	12–10	9–7	6–2	1–0
rc	*rb*	*imm3*	*opcode*	*size*
rc	**rb**	**uimm**	**10001**	**00**

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.

```
(i64)[r[rb] + uimm3 * 8] = r[rc]
```

2.1.19. compare.i64



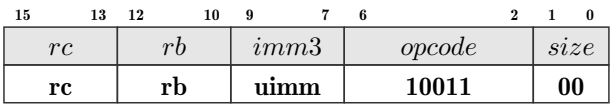
15–13	12–10	9–7	6–2	1–0
rc	*rb*	*imm3*	*opcode*	*size*
rc	**rb**	**uimm**	**10010**	**00**

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: *0 →less than (signed)*, *1 →greather or equal (signed)*, *2 →equal*, *3 →not equal*, *4 →less than (unsigned)*, *5 →greater or equal (unsigned)*, or *6 →conditional move*.

```
match fun3
| lt  -> flag = (i64)r[rc] <  (i64)r[rb]
| ge  -> flag = (i64)r[rc] >= (i64)r[rb]
| eq  -> flag =      r[rc] =       r[rb]
| ne  -> flag =      r[rc] !=      r[rb]
| ltu -> flag = (u64)r[rc] <  (u64)r[rb]
| geu -> flag = (u64)r[rc] >= (u64)r[rb]
| mov -> if (flag) r[rc] = r[rb]
```

2.1.20. logic.i64



15–13	12–10	9–7	6–2	1–0
rc	*rb*	*imm3*	*opcode*	*size*
rc	**rb**	**uimm**	**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: *0 →move*, *1 →logical not*, *2 →negate*, *3 →bswap*, *4 →count trailing zeros*, *5 →count leading zeros*, *6 →count population*, or *7 →sign extend*.

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

2.1.21. pin.i64

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 + 2*, 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*.

```
rpc = (i32)(pc - r[ra] + 2)
rib = (i32)(ib - r[rb]    )
r[rc] = (i32x2){rpc,rib}
```

2.1.22. and.i64

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*.

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

2.1.23. or.i64

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*.

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

2.1.24. xor.i64

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*.

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

2.1.25. add.i64

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.

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

2.1.26. srl.i64

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.

```
r[rc] = (u64)r[rb] >> r[ra]
```

2.1.27. sra.i64

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.

```
r[rc] = (i64)r[rb] >> r[ra]
```

2.1.28. sll.i64

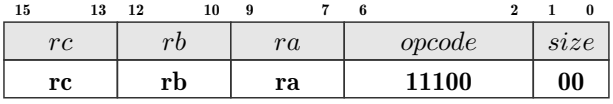
15 13	12 10	9 7	6 2	1 0
rc	*rb*	*ra*	*opcode*	*size*
rc	**rb**	**ra**	**11011**	**00**

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.

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

2.1.29. sub.i64

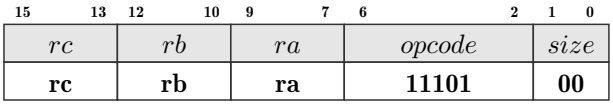


15–13	12–10	9–7	6–2	1–0	
rc	*rb*	*ra*	*opcode*	*size*	
rc	**rb**	**ra**	**11100**	**00**	**sub.i64** rc, rb, ra

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

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

2.1.30. mul.i64

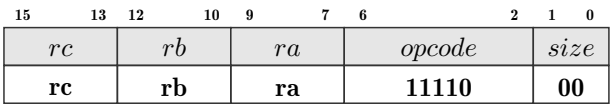


15–13	12–10	9–7	6–2	1–0	
rc	*rb*	*ra*	*opcode*	*size*	
rc	**rb**	**ra**	**11101**	**00**	**mul.i64** rc, rb, ra

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

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

2.1.31. div.i64

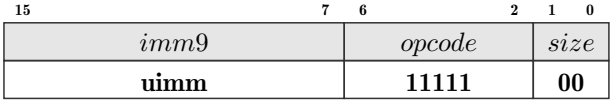


15–13	12–10	9–7	6–2	1–0	
rc	*rb*	*ra*	*opcode*	*size*	
rc	**rb**	**ra**	**11110**	**00**	**div.i64** rc, rb, ra

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 a divide by zero exception.

```
r[rc] = r[rb] / r[ra]
```

2.1.32. 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.

3. Assembler

3.1. Introduction

this glyph assembly language reference begins with an introduction to assembler and linker concepts, followed by sections describing the glyph assembler directives, and pseudo-instruction aliases. section 2 contains a complete listing of instruction.

3.2. Concepts

this section covers assembler high level concepts required to understand the concepts involved in assembling and linking executable code from source files.

3.2.1. assembly file

an assembly file contains assembly language directives, macros and instructions. it can be emitted by a compiler or it can be handwritten. an assembly file is the input file to the assembler. the extensions for assembly files are `.s`.

3.2.2. relocatable object file

relocatable object files contain compiled object code and data emitted by the assembler. an object file cannot be run, rather it is used as input to the linker as a step towards producing an executable file. the extension for object files is `.o`.

3.2.3. file header

an assembler file has a file header that contains magic to indicate how the file is formatted, the architecture of the binary, the endianness of the binary; *little-endian* in the case of glyph, the file type *(relocatable object, executable, shared library)*, the number of program headers and their offsets in the file, the number of section headers and their offsets in the file, plus fields indicating the file format version and various other details.

3.2.4. program header

program headers provide size and offsets of loadable segments within an executable or shared library along with protection attributes used by the operating system *(read, write and exec)*. program headers are not present in relocatable object files and are primarily for use by the operating system to and dynamic linker to map code and data into memory.

3.2.5. section header

section headers provide size, offset, type, alignment and flags for the sections contained within the binary file. section headers are not required to execute a static binary but are necessary for dynamic linking and program linking. various section types refer to the location of the symbol table, relocations and dynamic symbols in the binary file.

3.2.6. sections

an object file is made up of multiple sections, with each section corresponding to distinct types of executable code or data. there are a variety of different section types. this list contains the four most common sections:

- `.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

3.2.7. program linking

program linking is the process of reading multiple relocatable object files, merging the sections from each of the source files, calculating the new addresses for symbols and applying relocation fixups to text or data that is pointed to in relocation entries.

3.2.8. linker script

linker scripts are text source files that are optionally input to the linker containing rules for the linker to use when calculating the load address and alignment of the various sections when creating an executable output file. the extension for linker scripts is `.ld`.

3.3. Directives

the assembler implements a number of directives that control the assembly of instructions into object files. these directives give 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 3.1.: Assembler directives

3.4. 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` *ibcall-reloc(text-label,const-label)*	procedure call
`ret`	`jtlib` *ibret-reloc(block-entry-label)*	procedure return
`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,* `mov`	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 3.2.: Pseudo instructions

3.5. Calling convention

3.5.1. calling convention — 16-bit

the 16-bit instruction packet, while intended to be used in conjunction with the 32-bit opcodes, is designed as a complete subset, so there is an ABI variant that targets a subset using only 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	t0	temporary register 0	caller
r5	a0	argument register 0	caller
r6	a1	argument register 1	caller
r7	ra	return address / (pc,ib) link vector	caller

Table 3.3.: 16-bit register assignment

A. Appendix

A.1. Opcode summary — 16-bit

15 13	12 10	9 7	6 2	1 0	
uimm			**00000**	**00**	**break** uimm9
simm			**00001**	**00**	**j** simm9
simm			**00010**	**00**	**b** simm9
simm			**00011**	**00**	**ibj** simm9
rc	**uimm**		**00100**	**00**	**jalib.i64** rc, ib64(uimm6*8)
rc	**uimm**		**00101**	**00**	**jtlib.i64** rc, ib64(uimm6*8)
rc	**uimm**		**00110**	**00**	**movib.i64** rc, ib64(uimm6*8)
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**	**addib.i64** rc, ib32(uimm6*4)
rc	**uimm**		**01101**	**00**	**leapc.i64** rc, ib32(uimm6*4)(pc)
rc	**uimm**		**01110**	**00**	**loadpc.i64** rc, ib32(uimm6*4)(pc)
rc	**uimm**		**01111**	**00**	**storepc.i64** rc, ib32(uimm6*4)(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**	**fun3**	**10010**	**00**	**compare.i64** rc, rb, fun3
rc	**rb**	**fun3**	**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