RV32I-Lite Encoding Card

2,284 words · ~10 min read

VCA-CSA-101 cross-chapter quick-reference handout. Anchors: §4.5-§4.10; spec source: 5).

Purpose: complete RV32I-Lite ISA encoding reference for hand-encoding, hand-decoding, and assembler/disassembler verification. Print double-sided; pin to the wall during Labs 4.1-4.5, Ch 6 assembler work, and Ch 6a linker relocation work. Every byte your CPU fetches (on Tang Primer 25K canonical Phase-1 silicon, or Tang Nano 20K advanced-track silicon) is encoded by this card; every byte `riscv32-unknown-elf-objdump` disassembles agrees with it.

At a glance

Property	Value
Base	RV32I subset (deliberate; everything emitted is real RV32I-legal)
Instructions	**11 real** + **9 pseudo** (8 expand to one real instruction; `la` expands to two via R_VIRTUS_LA_GP12 reloc. See §The 9 pseudo-instructions)
Formats	**4** of RV32I's 6, R, I, S, B
Word size	**32 bits** (every instruction is exactly 4 bytes)
Endianness	Little-endian
Registers	**8** general-purpose (`x0` - `x7`); `x0` hardwired to zero
Alignment	All instructions 4-byte aligned; all data accesses word-aligned
Branch range	±4 KiB (B-format 13-bit signed offset, 2-byte units)
Compatibility	Bit-for-bit compatible with full RV32I. `riscv32-unknown-elf-as` accepts our source; `riscv32-unknown-elf-objdump` decodes our binaries

---

Register file

RV32I-Lite has **8 registers** (full RV32I has 32; we use a strict subset). The ABI mnemonics below match full-RV32I conventions, so RV32I-Lite source is directly readable as RV32I source.

Register	ABI name	Saver	Role in CSA-101 / Virtus OS v1
`x0`	`zero`	(hardwired)	Hardwired to 0; reads return 0; writes discarded
`x1`	`ra`	Caller	Return address (set by `jalr` linkage; consumed by `ret` pseudo)
`x2`	`sp`	Caller	Stack pointer; descending; word-aligned
`x3`	`gp`	-	Global pointer; reserved by ABI; **unused in Virtus OS v1**
`x4`	`tp`	-	Thread pointer; reserved by ABI; **unused in Virtus OS v1**
`x5`	`t0`	Caller	Temporary / argument 0 / scratch
`x6`	`t1`	Caller	Temporary / argument 1 / scratch
`x7`	`t2`	Caller	Temporary / argument 2 / scratch

Encoding: any register field (5 bits) takes values `00000`-`00111` for `x0`-`x7`. Values `01000`-`11111` are reserved for full-RV32I registers `x8`-`x31` and never emitted by RV32I-Lite code.

Register convention (CSA-201 forward-compatible)

The convention is committed in `vm/protocol.py:14-39` (canonical block comment; ground-truth source). Ch 8 §8.6.2 promotes it to chapter prose.

RV32I-Lite emits a strict subset of the wider RV32I ABI; the wider classes are forward-compatible reservations.

Class	Members in CSA-101 emit	Reserved (CSA-201+)	Caller's responsibility	Callee's res
Caller-clobbered ("temporary")	`t0`, `t1`, `t2`	`t3`-`t6`	Save before `call` if value needed after	None. May c freely
Callee-preserved ("saved")	(none in current emit)	`s0`-`s11`	None. Assume preserved across `call`	Save to stac use; restore `return`
Argument / return	(passed via stack. See vm-segment-cheat-sheet)	`a0`-`a7` (arg); `a0` (return)	Push args before `call`; pop result after	Read args fr push result l `return`
Stack pointer	`sp` (= `x2`)	-	None	Restore `sp` before `retu` 8 §8.7 step 4
Global pointer	`gp` (= `x3`)	-	None	**Never write** the linker pr (`linker/pr` R7.2-α) write code reads t segment-po `gp+0..0x10`
Return address	(delivered via stack push. See Ch 8 §8.6)	`ra` (= `x1`)	Push return-label before `call`	Pop into tem it
Thread pointer	(not used)	`tp` (= `x4`)	-	-

Why students rarely see this directly. Jack-emitted programs go compiler → translator → assembler → linker → silicon, and the register convention is consumed entirely between translator and assembler, by the time bytecode reaches the assembler, the register choices are already baked into emission templates. **The Jack programmer never picks `t0` or `t1`; the translator does.** The convention only becomes visible to the student in three places: Lab 8.2 (writing the translator that picks the registers), Lab 8.4 (gdb session against running silicon. `info registers` shows `t0`/`t1`/`t2` holding scratch values), and CSA-201's inline-asm / hand-rolled-extension labs (where the student honors the convention themselves).

Cross-references: Ch 8 §8.6.2 (chapter-prose enumeration); `vm/protocol.py:14-39` (canonical source); cross-chapter-vm-segment-cheat-sheet.md (saved-frame layout); cross-chapter-instr-mem-layout.md (gp+0..0x10 segment-pointer slots + gp+0x40..0x3FF la-ptr-table reserve).

The hardwired-zero trick. Because `x0` always reads as 0:

- `addi x0, x0, 0` ≡ `nop` (no operation)
- `addi rd, x0, n` ≡ `li rd, n` (load small immediate)
- `addi rd, rs, 0` ≡ `mv rd, rs` (move register)
- `sub rd, x0, rs` ≡ `neg rd, rs` (arithmetic negation)
- `beq rs, x0, label` ≡ `beqz rs, label` (branch if zero)

Each pseudo costs **zero opcodes**; the hardwired zero buys an entire family of conventional-looking instructions for free.

The 11 real instructions

R-format. Register-register arithmetic (5 instructions)

`opcode = 0110011` for all five. Differentiated by `funct3` and `funct7`.

Mnemonic	Action	funct7	funct3	opcode	Hex template
`add rd, rs1, rs2`	`rd = rs1 + rs2`	`0000000`	`000`	`0110011`	`0x00..0033`
`sub rd, rs1, rs2`	`rd = rs1 - rs2`	`0100000`	`000`	`0110011`	`0x40..0033`
`and rd, rs1, rs2`	`rd = rs1 & rs2`	`0000000`	`111`	`0110011`	`0x00..7033`
`or rd, rs1, rs2`	`rd = rs1 | rs2`	`0000000`	`110`	`0110011`	`0x00..6033`
`xor rd, rs1, rs2`	`rd = rs1 ^ rs2`	`0000000`	`100`	`0110011`	`0x00..4033`

Note: `add` and `sub` differ *only* in the high bit of `funct7`. In silicon: one bit selects the adder's carry-in (0 = add, 1 = subtract via two's-complement `rs1 + ~rs2 + 1`).

I-format. Register-immediate, loads, jalr (3 instructions)

Mnemonic	Action	funct3	opcode	Hex template
`addi rd, rs1, imm12`	`rd = rs1 + sext(imm12)`	`000`	`0010011`	`0x..0013`
`lw rd, imm12(rs1)`	`rd = M32[rs1 + sext(imm12)]`	`010`	`0000011`	`0x..2003`
`jalr rd, rs1, imm12`	`t = rs1 + sext(imm12); rd = PC+4; PC = t & ~1`	`000`	`1100111`	`0x..0067`

Immediate range: −2048 to +2047 (12-bit signed). Sign-extended at decode.

Word alignment: `lw`'s effective address (`rs1 + imm12`) must be a multiple of 4. Misaligned loads trap on real silicon (in CSA-101, behavior is undefined; CSA-201's PMP traps on misalignment).

`jalr` low-bit force-zero: the target address has its low bit cleared, regardless of `rs1+imm12`. This preserves 2-byte alignment for the C extension (which RV32I-Lite does not use, but the encoding preserves bit-for-bit).

S-format. Stores (1 instruction)

Mnemonic	Action	funct3	opcode	Hex template
`sw rs2, imm12(rs1)`	`M32[rs1 + sext(imm12)] = rs2`	`010`	`0100011`	`0x..2023`

Operand-order quirk: `sw rs2, offset(rs1)` lists the *source* register first in assembly source. (RV32I convention; not RV32I-Lite invention.)

Immediate split: the 12-bit immediate is split. `imm[11:5]` at bits [31:25], `imm[4:0]` at bits [11:7]. The split preserves the position of `rs1` and `rs2` so register-file read ports do not need format-aware muxing.

B-format. Conditional branches (2 instructions)

Mnemonic	Action	funct3	opcode	Hex template
`beq rs1, rs2, label`	`if rs1 == rs2 then PC += sext(imm13)`	`000`	`1100011`	`0x..0063`
`bne rs1, rs2, label`	`if rs1 != rs2 then PC += sext(imm13)`	`001`	`1100011`	`0x..1063`

13-bit signed offset, 2-byte units → ±4 KiB reachable from the branch instruction. Bit 0 of the offset is forced to zero (preserved for C extension compatibility).

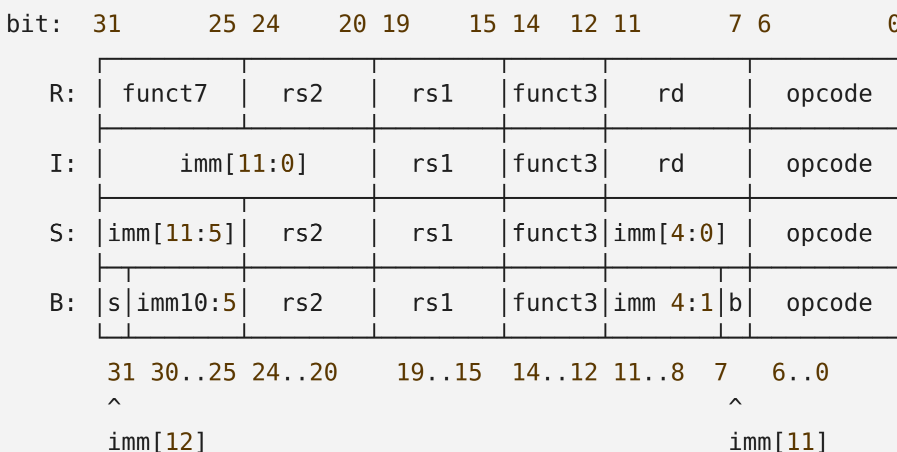
The B-format immediate shuffle is the most heavily-shuffled in RV32I:

- `imm[12]` → bit 31 (sign bit, aligned with S-format and I-format sign bits)
- `imm[10:5]` → bits 30:25
- `imm[4:1]` → bits 11:8
- `imm[11]` → bit 7 (isolated; **most-commonly-missed position**)

The shuffle preserves `rs1` at [19:15] and `rs2` at [24:20].

Bit-field layouts

Each format is a 32-bit word. Bit 31 (MSB) is on the left. Register fields are 5 bits; `funct3` is 3 bits; `funct7` is 7 bits; `opcode` is 7 bits.



Cross-format invariants (the entire reason CPU decode is cheap):

- `opcode` is always at bits [6:0]
- `funct3` (when present) is always at bits [14:12]
- `rs1` is always at bits [19:15]
- `rs2` (when present) is always at bits [24:20]
- `rd` (when present) is always at bits [11:7]

- The sign bit of any immediate is always at bit 31. Sign-extension hardware is driven *unconditionally* from bit 31

Hand-encoding checklist

For any instruction:

1. **Identify the format** from the opcode/family table above.
2. **Place each field** at its bit position in the format layout.
3. **Concatenate** to 32 bits.
4. **Group into nibbles** for hex.
5. **Verify** with `printf "<asm>" | riscv32-unknown-elf-as -march=rv32i -o /tmp/t.o -` then `riscv32-unknown-elf-objdump -d /tmp/t.o`.

Worked: `addi x1, x0, 5`

- Format: I; `funct3 = 000`, `opcode = 0010011`
- Fields: `imm12 = 0x005`, `rs1 = x0 = 00000`, `rd = x1 = 00001`
- Place:

```
imm[11:0]    rs1   funct3  rd     opcode
000000000101 00000 000     00001  0010011
```

- Concatenate: `00000000010100000000000010010011`
- Hex: `0x00500093`
- Verify: `riscv32-unknown-elf-objdump -d` → `00500093 addi ra,zero,5`

Worked: `sw x6, 12(x2)`

- Format: S; `funct3 = 010`, `opcode = 0100011`
- Fields: `imm12 = 0x00C` → split `imm[11:5] = 0000000`, `imm[4:0] = 01100`; `rs1 = x2 = 00010`; `rs2 = x6 = 00110`
- Place:

```
imm[11:5] rs2   rs1   funct3 imm[4:0] opcode
0000000   00110 00010 010    01100    0100011
```

- Concatenate: `00000000011000010010011000100011`
- Hex: `0x00612623`

Worked: `beq x5, x0, +8` (branch forward 8 bytes)

- Format: B; `funct3 = 000`, `opcode = 1100011`
- Offset = +8 bytes → `imm13 = 0b0000000001000`
- Split per shuffle: `imm[12] = 0`, `imm[11] = 0`, `imm[10:5] = 000000`, `imm[4:1] = 0100`
- Fields: `rs1 = x5 = 00101`, `rs2 = x0 = 00000`
- Place:

```
imm[12] imm[10:5] rs2   rs1   funct3 imm[4:1] imm[11] opcode
0       000000    00000 00101 000    0100     0       1100011
```

- Concatenate: `00000000000000101000010001100011`
- Hex: `0x00028463`

The 9 pseudo-instructions

Eight pseudos expand to **one** real instruction at assembly time. The ninth (`la`) is a CSA-101-specific carve-out that expands to **two** instructions plus a linker relocation. See the note immediately after the table. No pseudo has its own opcode. The student writes the pseudo; the assembler emits the real bytes; the disassembler may display either form (per `--no-aliases` flag).

Pseudo	Expansion	Use
`nop`	`addi x0, x0, 0`	No-op; padding; pipeline stall
`mv rd, rs`	`addi rd, rs, 0`	Copy register (RV32I has no dedicated move)
`li rd, imm` *(imm fits in 12 bits)*	`addi rd, x0, imm`	Load small immediate (range −2048..+2047)
`neg rd, rs`	`sub rd, x0, rs`	Arithmetic negation: `rd = -rs`
`beqz rs, label`	`beq rs, x0, label`	Branch if `rs == 0`
`bnez rs, label`	`bne rs, x0, label`	Branch if `rs != 0`
`ret`	`jalr x0, x1, 0`	Return from subroutine
`jr rs`	`jalr x0, rs, 0`	Jump to register; no link
`la rd, sym`	`addi rd, gp, off12` + `lw rd, 0(rd)` *(see note below)*	Materialise the address of a `.data`-resident symbol via a linker-resolved 12-bit offset off `gp` (CSA-101 carve-out. Non-standard)

Note on `la` (CSA-101 carve-out. Non-standard expansion). Standard RV32I expands `la rd, sym` to `auipc rd, %hi(sym)` + `addi rd, rd, %lo(sym)`. CSA-101 has neither `auipc` nor `lui`, so it cannot synthesise a 32-bit absolute address from a 20-bit upper-immediate. Instead, the linker emits a **per-symbol pointer-table entry** in `.data`, anchored off `gp` (the global pointer). The assembler emits a placeholder `addi rd, gp, 0` + `lw rd, 0(rd)` pair carrying a single `R_VIRTUS_LA_GP12` relocation; the linker patches the 12-bit immediate to the symbol's pointer-table offset. The la-ptr-table itself lives at **`gp+0x40`** (= `.data + 0x40`); the leading 64 bytes (`gp+0x00..gp+0x3F`) are reserved for the VM segment-pointer region (`LCL_addr` / `ARG_addr` / `THIS_addr` / `THAT_addr` at `gp+0x00..gp+0x0F` + `temp[0..7]` at `gp+0x10..gp+0x2F` per `cross-chapter-vm-segment-cheat-sheet.md`), with a 16-byte alignment cushion at `gp+0x30..gp+0x3F` for future segment additions. This places the slot capacity at `(2048 − 0x40) / 4 = 496` la-references per program. Adequate for any CSA-101 program. This is a CSA-101 simplification, CSA-201 reverts to the standard `auipc`+`addi` expansion once U-format lands. *The pointer-table approach (Option A) was selected; the alternative of a dedicated la-base register (Option B) was deferred.*

Disassembler display. `objdump` recognises the pseudo patterns and displays them by default:

```
$ printf 'ret\n' | riscv32-unknown-elf-as -march=rv32i -o /tmp/t.o -
$ riscv32-unknown-elf-objdump -d /tmp/t.o
   0:   00008067    ret           # default: pseudo form
$ riscv32-unknown-elf-objdump -d --no-aliases /tmp/t.o
   0:   00008067    jalr  zero,ra,0   # literal underlying instruction
```

Ghidra always shows the literal underlying instruction. *Decompiler is conservative; pseudos are display-layer convenience, not semantic content.*

What's deliberately NOT in RV32I-Lite

The full RV32I base ISA has **47 instructions**; RV32I-Lite has 11. Each missing instruction is a deliberate teaching choice, the cost of its absence is felt in CSA-101, recovered in CSA-201.

Missing instructions (from full RV32I base)

Instruction(s)	What it does	Cost in CSA-101	Returns in CSA-201
`jal rd, imm21`	J-format unconditional jump-and-link with 20-bit PC-relative offset	All control transfer must use `jalr` (register-indirect) or `beq x0, x0` (always-taken branch)	CSA-201 §1. Adds J-format
`lui rd, imm20`	Load upper immediate (high 20 bits)	Cannot materialise 32-bit immediates in one instruction; large constants go through `.data`-resident pointers via `la` pseudo	CSA-201 §1. Adds U-format
`auipc rd, imm20`	Add upper immediate to PC; produces PC-relative 32-bit address	Cannot do PC-relative 32-bit addressing; cross-section calls use linker-resolved indirection	CSA-201 §1. Adds U-format
`slt`, `slti`, `sltu`, `sltiu`	Set if less than (signed/unsigned, register/ immediate)	`lt`/`gt` comparisons require `sub` + sign-bit extraction + branch (~12 instructions per `lt`/`gt`)	CSA-201 §2. Adds set-less-than family
`blt`, `bge`, `bltu`, `bgeu`	Signed/unsigned compare-and-branch (less-than, greater-or-equal)	Inequality branches synthesised from `sub` + `beq`/`bne` (or compose via `slt` once it lands)	CSA-201 §2
`xori`, `andi`, `ori`	Immediate bitwise (XOR/AND/OR with sign-extended 12-bit immediate)	Cannot mask/toggle immediate bits in one instruction; must materialise constant first	CSA-201 §1
`slli`, `srli`, `srai`	Shift left/right (logical/arithmetic) by immediate	"Shift by k" implemented as `k` self-adds (`add rd, rd, rd` repeated); 1-cycle shift becomes `k`-cycle loop	CSA-201 §1
`sll`, `srl`, `sra`	Shift left/right by register	Same. Software-loop shift	CSA-201 §1
`lh`, `lhu`, `lb`, `lbu`	Half-word and byte loads (signed/ unsigned)	All loads are word loads; byte access requires `lw` + mask + shift	CSA-201 §1

Instruction(s)	What it does	Cost in CSA-101	Returns in CSA-201
`sh`, `sb`	Half-word and byte stores	All stores are word stores; byte modify requires read-modify-write	CSA-201 §1 (the framebuffer's RMW hazard from Ch 12 §12.6.5 lands here)
`fence`, `fence.i`	Memory ordering / instruction-fence	Single-threaded, single-core, no caches. Fences are no-ops in CSA-101	CSA-201 §6 (when preemption arrives)
`ecall`, `ebreak`	Environment call (syscall trap) / debugger trap	No supervisor mode; OS services called via direct `jalr`	CSA-201 §2. Adds privilege levels + traps
CSR instructions (`csrrw` etc.)	Control-status register read/write	No CSRs in CSA-101 (no privilege state, no machine-mode bookkeeping)	CSA-201 §2

Missing extensions

Extension	What it adds	When it returns
M extension (`mul`, `mulh`, `div`, `rem`)	Hardware multiply and divide	CSA-201 §3, Math.lib's ~1000-cycle software `multiply` becomes a 1-cycle `mul`; the gap is the speedup the student personally measures
A extension (`lr.w`, `sc.w`, atomic RMW)	Atomic load-reserved/ store-conditional; atomic arithmetic	CSA-201 §6, when preemption arrives and the framebuffer's RMW hazard becomes real
F extension (single-precision FP)	IEEE 754 binary32 add/ sub/mul/div/sqrt	con-101 (Virtus Console retro-FPU course), RV32I-Lite has no FPU
D extension (double-precision FP)	IEEE 754 binary64	(not in CSA-201 either; advanced electives only)
C extension (compressed)	16-bit instruction encodings interleaved with 32-bit	Out of scope across CSA-101 + CSA-201 (the encoding *preserves* C-extension compatibility (branches are 2-byte aligned) but no 16-bit instructions are emitted)
Zicfilp/Zicfiss (control-flow integrity)	Forward-edge type tags + backward-edge shadow stack	CSA-201 §8. Closes the CFI gap from Ch 12 §12.11

Pseudo-instructions deferred (need missing instructions)

Pseudo	Expansion	Why deferred
`not rd, rs`	`xori rd, rs, -1`	Needs `xori`
`seqz rd, rs`	`sltiu rd, rs, 1`	Needs `sltiu`
`snez rd, rs`	`sltu rd, x0, rs`	Needs `sltu`
`blez rs, l`	`bge x0, rs, l`	Needs `bge`
`bgez rs, l`	`bge rs, x0, l`	Needs `bge`
`bltz rs, l`	`blt rs, x0, l`	Needs `blt`
`bgtz rs, l`	`blt x0, rs, l`	Needs `blt`
`j label`	`jal x0, label`	Needs `jal` (J-format)
`call label`	`auipc x1, ...` + `jalr x1, ...`	Needs `auipc` (U-format)
`tail label`	`auipc x6, ...` + `jalr x0, ...`	Needs `auipc`
`nop`-N (multi-cycle pad)	N × `addi x0, x0, 0`	Available in CSA-101 (just N nops); not a single pseudo

(`la rd, sym` is a CSA-101 pseudo. See §The 9 pseudo-instructions above. Its CSA-101 expansion is non-standard.)

Identifier syntax (labels, symbol names)

The assembler accepts the following character classes in identifiers (label definitions, symbol references, directive arguments):

Position	Allowed characters
First character	`A-Z`, `a-z`, `_`, `.`
Continuation	`A-Z`, `a-z`, `0-9`, `_`, `.`, `$`

Why `$` is admitted in continuation. The Ch 7/8 VM translator emits per-function label namespacing in the form `<funcname>$<label>`. E.g. `Main.run$IF_FALSE_0`, `Main.run$WHILE_TOP_2`. This is the Jack/Hack convention preserved in the Virtus VM (Ch 8

§8.2): the `$` character disambiguates a function-local label from the function's own name without requiring per-function symbol tables in the assembler. Since Ch 4 doesn't pin identifier syntax (Lab 4.x labels are simple alphanumeric) and Ch 6 introduced labels without enumerating `$`, this carve-out was confirmed when the VM translator's `$`-bearing labels first reached the assembler (control-flow inside `function`).

Why `.` is admitted everywhere. Ch 11 stdlib service mangling (`Output.printString`, `Math.multiply`) uses `.` as the class-method delimiter; the linker resolves these symbols verbatim against the stdlib roster. Same convention as full RV32I.

Cross-format common-bit-position reference

For the decoder you build in Ch 5 (and for hand-decoding in Lab 4.2):

```
Read these bits unconditionally, every cycle, regardless of format:

  bit [6:0]    → opcode             (always)
  bit [11:7]   → rd  OR  imm[4:0]   (R/I have rd; S has imm-low; B has imm[4:1]
+imm[11])
  bit [14:12]  → funct3              (when format has it)
  bit [19:15]  → rs1                 (always - even formats with no rs1 read here
harmlessly)
  bit [24:20]  → rs2                 (always - same)
  bit [31:25]  → funct7 OR imm[11:5] (R has funct7; S has imm-high; B has imm[12]
+imm[10:5])
  bit [31]     → sign bit of any immediate (sign-extender driven unconditionally
from this bit)
```

Format-disambiguation by opcode (the only field whose meaning is format-independent):

opcode	Format	Family
`0110011`	R	Register-register arithmetic (`add` / `sub` / `and` / `or` / `xor`)
`0010011`	I	Register-immediate arithmetic (`addi`)
`0000011`	I	Loads (`lw`)
`0100011`	S	Stores (`sw`)
`1100011`	B	Branches (`beq` / `bne`)
`1100111`	I	`jalr`
`0000000`	(illegal in RV32I-Lite)	The all-zeros word, Lab 4.2's HALT-trap test case

---

Verification. Bit-identical against the GNU toolchain

Every byte RV32I-Lite emits is **bit-identical** to what `riscv32-unknown-elf-as -march=rv32i` emits for the same source. This is the chapter's central claim and the chapter's reproducibility test.

```
# Encode the same instruction by hand and via GNU as
$ printf 'addi x1, x0, 5\n' | riscv32-unknown-elf-as -march=rv32i -o /tmp/gnu.o -
$ riscv32-unknown-elf-objdump -d /tmp/gnu.o
   0:   00500093    addi  ra,zero,5

# Hand-encoded: 0x00500093 - match
```

The chapter's Lab 4.4 ("hand-encoded vs GNU-emitted bit-comparison") is graded on this property. **The Virtus subset is real RISC-V, not a Virtus fork**. Disassembly works in `objdump`, in Ghidra, and in Compiler Explorer.

Where to read more

- **Ch 4** *Machine Language*. Full encoding walkthrough; the chapter this card distills.
- **Ch 5** *Computer Architecture*, the silicon decoder that consumes these bytes.
- **Ch 6** *Assembler*, the program that emits these bytes from text source.

- **Ch 6a** *Static Linker*. `R_VIRTUS_32` and `R_VIRTUS_BRANCH13` relocations that patch fields into already-encoded instructions (see VOF v1 layout reference handout).
- **Ch 8** *VM II*, the call protocol that uses `jalr` for control transfer (RV32I-Lite's substitute for the missing `jal`).
- **Findings §16**, the canonical spec source; this card derives from it.
- **`riscv-spec.pdf`** (RISC-V International, official spec). Full RV32I and the M / A / F / D / C / Zicfilp / Zicfiss extensions referenced in the "Missing Extensions" table.
- **`cross-chapter-silicon-level-reading-guide.md`**: once you have encoded an RV32I-Lite instruction by hand, you may want to see what the actual silicon executing instructions at this abstraction level looks like. The silicon-level reading guide walks through the MOS 6502 and Z80 at transistor scale using the Visual6502 interactive simulator and Ken Shirriff's die-shot annotations at righto.com. The same fetch-decode-execute sequence your encoded instruction triggers in your Tang Primer 25K CPU is visible, transistor by transistor, in 1975 and 1976 fabricated silicon. See also `cross-chapter-fpga-cell-to-silicon-bridge.md` for how your synthesized LUT4 cells and flip-flops correspond to physical silicon structures.

© Virtus Cyber Academy. Generated 2026-05-10.