From C to Bits: Everything Becomes an Instruction

Rodolfo Azevedo

Institute of Computing, University of Campinas (UNICAMP), Brazil

rodolfo.azevedo@unicamp.br

http://www.ic.unicamp.br/~rodolfo/mo801

Goal of this class

Module 2, Class 5: the compiler's job, reading disassembly, measuring performance, and the first KWS cycle estimate.

> Every line of C — an `if`, a `for`, a function call, an array access — becomes a sequence of RV32I instructions. Understanding that mapping is what lets you predict and measure performance, and it is what motivates the rest of this course.

At the end of this class, you should be able to:

- Trace how C constructs (conditionals, loops, function calls, array accesses) map to RV32I instruction sequences.
- Read and interpret RISC-V disassembly output from `objdump -d`.
- Estimate execution time from instruction count, instruction mix, and CPI.
- Measure cycle counts on real hardware using the `rdcycle` CSR and explain what the number means.

The compilation chain

```
main.c
  │
  ▼  riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -S
main.s       (human-readable assembly)
  │
  ▼  riscv64-unknown-elf-gcc -c
main.o       (relocatable object — binary, with symbol table)
  │
  ▼  riscv64-unknown-elf-ld -T link.ld
main.elf     (linked binary — addresses resolved)
  │
  ├─► riscv64-unknown-elf-objdump -d   (disassembly — back to text)
  └─► riscv64-unknown-elf-objcopy -O binary → main.bin → main.hex
```

At every stage, you can inspect the output. The disassembly is your most powerful debugging tool — it shows exactly what your processor will execute.

Startup code: crt0.S

Before `main()` runs, the processor needs a small assembly stub to set up the C runtime environment. In a bare-metal RV32I system, there is no OS to do this:

```
# crt0.S — minimal C runtime startup for RV32I
.section .text
.global _start
_start:
    # 1. Set the stack pointer to the top of DMEM
    la   sp, _stack_top      # _stack_top defined in linker script

    # 2. Zero the .bss section (uninitialised global variables)
    la   t0, __bss_start
    la   t1, __bss_end
bss_loop:
    bge  t0, t1, bss_done
    sw   zero, 0(t0)
    addi t0, t0, 4
    j    bss_loop
bss_done:
    # 3. Call main — never returns in our system
    call main
    # 4. If main returns, loop forever (ebreak in debug)
    ebreak
```

Without this: `sp` is undefined (garbage), `.bss` variables have random values, `main()` is never called.

The Makefile compiles `crt0.S` alongside all C files: `$(CC) crt0.S main.c -o main.elf`.

Linker script: placing code in memory

A linker script tells the linker where each section of the ELF binary lives in the target's address space:

```
/* link.ld — Harvard architecture: IMEM and DMEM are separate */
MEMORY {
  IMEM (rx)  : ORIGIN = 0x00000000, LENGTH = 4K
  DMEM (rw)  : ORIGIN = 0x00010000, LENGTH = 4K
}

SECTIONS {
  .text   : { *(.text*)   } > IMEM    /* code */
  .rodata : { *(.rodata*) } > DMEM    /* const data — must be in DMEM! */
  .data   : { *(.data*)   } > DMEM    /* initialized globals */
  .bss    : {                          /* zero-initialized globals */
    __bss_start = .;
    *(.bss*)
    __bss_end = .;
  } > DMEM

  _stack_top = ORIGIN(DMEM) + LENGTH(DMEM);  /* stack grows down */
}
```

Harvard architecture implication: there is no path from IMEM to DMEM at runtime. Traditionally, `.rodata` (string literals, `const` arrays) lives in flash and is read via the instruction bus — but here, our IMEM only executes instructions. **All data — even constants — must be in DMEM from the start**. The linker script enforces this by placing `.rodata` in the DMEM region.

ELF sections: what goes where

An ELF binary has named sections. The ones that matter for bare-metal:

Section	Content	In binary?	In memory?
`.text`	Machine instructions	✅ Yes	IMEM
`.rodata`	`const` arrays, string literals	✅ Yes	DMEM
`.data`	Initialised global variables	✅ Yes	DMEM
`.bss`	Uninitialised / zero-init globals	❌ No (just size)	DMEM (zeroed by crt0.S)
`.stack`	Stack space	❌ No	DMEM (reserved by linker)

`.bss` is not stored in the binary because it is always zero — `crt0.S` zeroes the memory region at startup. This is why `uint32_t arr[1024];` (global, uninitialized) adds 4KB to the memory footprint but **not** to the `.bin` file size.

static linkage: one copy per translation unit

A subtle C pitfall when sharing large arrays between files:

```
// weights.h — included by both kws_main.c and kws_kernel.c
static const int8_t conv_weights[4096] = { ... };  // 4 KB
```

Because the array is `static`, each `.c` file that `#include`s `weights.h` gets its **own private copy**. With two files including it: 8 KB used instead of 4 KB. With five files: 20 KB. In a processor with 4 KB of DMEM, this is fatal.

Fix: declare in the `.h` file with `extern`, define once in a `.c` file:

```
// weights.h
extern const int8_t conv_weights[4096];

// weights.c  (compiled and linked once)
const int8_t conv_weights[4096] = { ... };
```

This bug was encountered in the course reference implementation: ~16 KB of KWS weights were duplicated across translation units, filling all available DMEM. Diagnosed via `riscv-none-elf-nm main.elf | sort -k2 -rn` (shows symbol sizes).

Reading a disassembly: simple arithmetic

C source:

```c
int add(int a, int b) { return a + b; }
```

Compiled with `-O0` (no optimization):

```
00000000 <add>:
   0: fd010113   addi  sp, sp, -48    # allocate stack frame
   4: 02112623   sw    ra, 44(sp)     # save return address
   8: 02812423   sw    s0, 40(sp)     # save frame pointer
   c: 03010413   addi  s0, sp, 48    # set frame pointer
  10: fea42623   sw    a0, -20(s0)   # store argument a
  14: feb42423   sw    a1, -24(s0)   # store argument b
  18: fec42703   lw    a4, -20(s0)   # reload a
  1c: fe842783   lw    a5, -24(s0)   # reload b
  20: 00f707b3   add   a5, a4, a5    # a + b
  24: 00078513   mv    a0, a5        # return value in a0
  28: ...        (restore, ret)
```

18 instructions for `a + b`. The compiler with `-O0` stores every variable on the stack and reloads it — there is no optimization.

The same function with `-O2`

```
00000000 <add>:
   0: 00b50533   add   a0, a0, a1    # a0 = a0 + a1 (result in a0)
   4: 00008067   ret
```

2 instructions. Arguments arrive in `a0`/`a1` (ABI convention), the result goes back in `a0`, and the compiler knows it does not need to spill anything to the stack. The optimization level is not a minor tuning knob — it changes the *structure* of the code.

The lesson: **performance estimates based on C source are meaningless without knowing the optimization level.** Always measure on compiled binaries.

What the compiler generates for common patterns

C construct	Typical RV32I instructions
`a + b`, `a - b`	`add`, `sub` (1 instruction)
`a * b` (no M ext.)	Library call: dozens of instructions
`a[i]`	`slli`, `add`, `lw` (index × 4 + base → load)
`if (a < b)`	`blt` or `slt` + `beq`
`for (i=0; i<N; i++)`	Initialize → `bge` / `blt` loop back
Function call	`jal ra, target`
Return	`jalr x0, ra, 0` (alias: `ret`)
`struct` field access	`lw` with immediate offset

Key insight: **memory access is the bottleneck.** `a[i]` is always at least two instructions (address compute + load). A convolution over a tensor is mostly loads and stores surrounding a single `add`.

The cycle counter: measuring elapsed time

The RV32I privileged spec defines a `mcycle` CSR (Control and Status Register) — a 64-bit counter incremented every clock cycle. Reading it before and after a computation gives you the exact cycle count:

```
static inline uint32_t read_cycle(void) {
    uint32_t c;
    __asm__ volatile ("csrr %0, mcycle" : "=r"(c));
    return c;
}

int main(void) {
    int a[8] = {1,2,3,4,5,6,7,8};
    int b[8] = {1,1,1,1,1,1,1,1};
    int sum = 0;

    uint32_t t0 = read_cycle();
    for (int i = 0; i < 8; i++) sum += a[i] * b[i];
    uint32_t t1 = read_cycle();

    // t1 - t0 = cycles to compute dot product
    __asm__ volatile ("ebreak");
}
```

For Project 1: implement `mcycle` as a 32-bit register that increments every cycle (the low 32 bits are sufficient for most measurements at 27 MHz).

Case study: int8 dot product cycle count

A dot product of two 8-element `int8` arrays, compiled with `-O0` on your RV32I core (no `M` extension):

```
Cycle count (8 elements, int8, -O0):
  Loop overhead (init, branch, increment):  ~5 cycles/iteration
  Array index + load (×2):                  ~6 cycles/iteration
  Multiply (software, no M ext.):           ~30–50 cycles/iteration
  Accumulate:                               ~1 cycle/iteration
  ──────────────────────────────────────────────────────────────
  Total per element:                        ~42–62 cycles
  Total for 8 elements:                     ~336–496 cycles
```

With `-O2` and the `M` extension (`mul` instruction):

```
  ~4–6 cycles/element → ~32–48 cycles total
```

A single `mul` instruction vs. a software multiply routine: **8–10× speedup from the extension alone**.

KWS inference cycle estimate

The `tiny_conv` model for keyword spotting:

- Input: 49×40 = **1,960** int8 values (MFCC features)
- Conv2D layer: 49×40 input, kernel K×K, N filters (exact dims from reference impl.)
- Each output value: K×K×C_in multiply-accumulate (MAC) operations
- Total model MACs: several hundred thousand (dominated by the Conv2D layer)

Cycle estimate on your Project 1 core (`-O0`, no Zmmul), at ~50 cycles/MAC:

$$\text{Total MACs} \times 50 \text{ cycles/MAC} = \text{several million cycles}$$

At 27 MHz: **several hundred milliseconds per inference**. KWS needs to respond within 1 second. With a naive implementation on a plain RV32I core, you may already be close to the budget.

This is why Project 3 exists.

In-class exercise — predict before you measure

Before running the compiler, **predict** the cycle count for this function with `-O0` and with `-O2`:

```
int dot8(int8_t *a, int8_t *b, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += (int)a[i] * (int)b[i];
    return sum;
}
```

With `n = 16` on your RV32I core (no M extension, no cache):

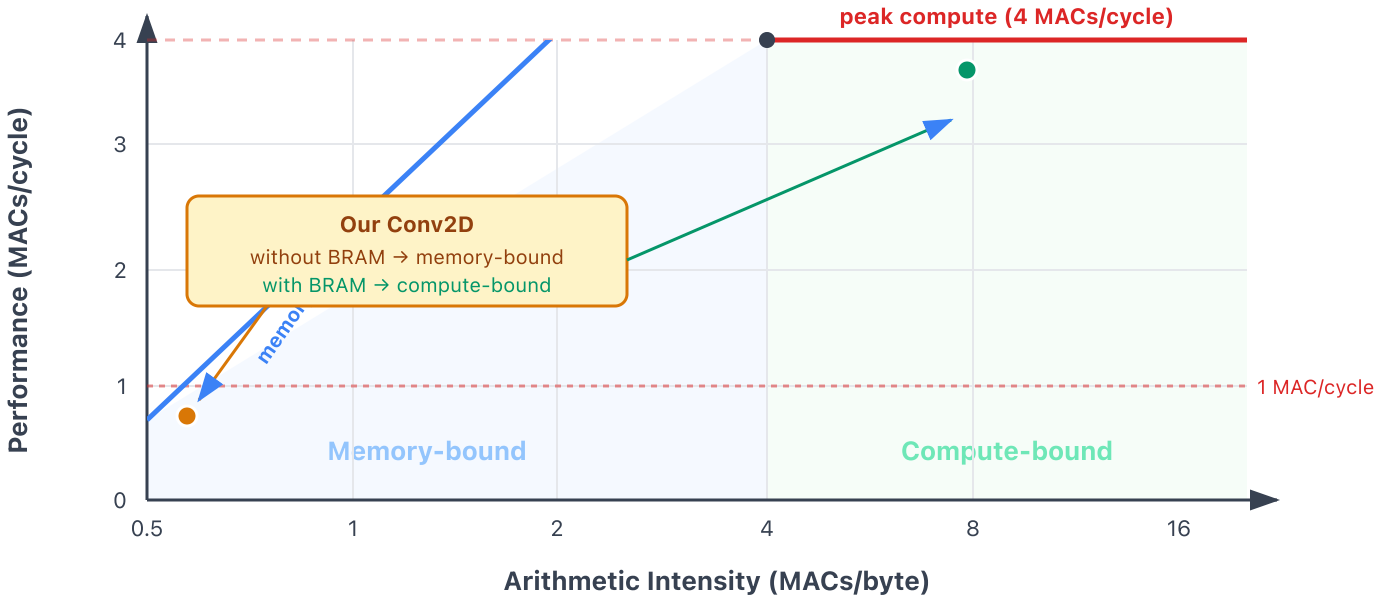
Think through:

1. With `-O0`, how many instructions does one loop iteration generate? (Hint: load, sign-extend cast, multiply via `mul`-absent fallback, add, increment, branch — count them.)
2. With `-O2`, which instructions does the compiler eliminate or combine?
3. If you had the `M` extension (`mul` instruction): how many cycles would `i * j` cost?

> After Lab 3, you will fill in the "Measured" row. Keep your prediction — comparing it to the measurement is more valuable than the number itself.

The roofline preview

A useful mental model: performance is bounded by either compute or memory bandwidth.



- **Low arithmetic intensity**: touching many bytes per MAC → memory bandwidth is the bottleneck.
- **High arithmetic intensity**: reusing data heavily → compute is the bottleneck.

A dot product reuses nothing — arithmetic intensity ≈ 0.5 MACs/byte. A matrix multiply reuses rows and columns — intensity = O(N). We will revisit this in Module 5 when designing the accelerator.

Lab 3 out — measure `-O0` vs. `-O2` on the board

Goal: measure the cycle count of a multiply loop on your Tang Nano 9K running Project 1.

```
// Compile twice: once with -O0, once with -O2
int result = 0;
uint32_t t0 = read_cycle();
for (int i = 0; i < 64; i++)
    result += (int8_t)a[i] * (int8_t)b[i];
uint32_t t1 = read_cycle();
// Store (t1-t0) in x10 (a0), then ebreak
// Read x10 from your testbench or from LEDs
```

Report:

1. Cycle count with `-O0` (no optimization).
2. Cycle count with `-O2`.
3. Ratio and explanation: which instructions disappeared and why.
4. (Optional) Add the `M` extension (`mul` instruction) and report the speedup.

Next module

Module 3 — HW/SW Interfaces: once you can run C programs on your core, the next question is how software talks to hardware peripherals — memory-mapped I/O, a minimal on-chip bus, and Zmmul + CMAC as examples of ISA extension and custom instruction. **Project 2 begins.**