ECE-6913 Computer System Architecture
Homework 3

New York University, Fall 2024

Due on 10/19, 11:59 PM

Name: Raman Kumar Jha
NYU ID: N13866145

Part I: RISC-V Programming

This document contains solutions to four RISC-V assembly programming problems, designed for the 32-bit RISC-V simulator (Venus). Each solution includes the problem description, the full assembly code, and a discussion of the implementation choices.

---

1. Sum of the Squares of Odd Numbers

Problem: Write a RISC V program using instructions in the RISC V ISA to calculate the sum of the squares of all odd numbers between 0 and +N where N is an integer ¡ 100.

RISC-V Code

```
.data
N: .word 10 # Example: N=10. Odds: 1, 3, 5, 7, 9
           # Squares: 1, 9, 25, 49, 81. Sum = 165

.text
main:
    lw   x5, N(x0)     # x5 (t0) = N (the limit)
    li   x6, 1         # x6 (t1) = i (current odd number)
    li   x7, 0         # x7 (t2) = sum (accumulator)

loop:
    bgt  x6, x5, done  # if (i > N) goto done
    mul  x8, x6, x6    # x8 (t3) = i * i
    add  x7, x7, x8    # sum = sum + (i*i)
    addi x6, x6, 2    # i = i + 2 (get next odd number)
    j    loop

done:
    # Print the final sum
    mv   a0, x7        # Move result to a0 for printing
    li   a7, 1         # ecall for print integer
    ecall

    # Exit program
    li   a7, 10        # ecall for exit
    ecall
```

Discussion

- **Register Allocation:**
  - `x5 (t0)`: Holds the limit `N`.
  - `x6 (t1)`: Serves as the loop counter `i`. It is initialized to 1, the first odd number.
  - `x7 (t2)`: The accumulator for the `sum`, initialized to 0.
  - `x8 (t3)`: A temporary register to hold the square of `i`.
- **Logic:** The code avoids checking every number. It starts the counter `i` at 1 and in each iteration, it increments `i` by 2 (`addi x6, x6, 2`). This ensures that only odd numbers (1, 3, 5, ...) are processed.
- **Efficiency:** This approach is more efficient than iterating 1-by-1 and checking for oddness, as it halves the number of loop iterations and removes a conditional branch (or modulo) from the loop.
- **Instructions:** The `mul` instruction (from the 'M' extension) is used for squaring. The loop terminates with `bgt` (Branch if Greater Than) when the counter `i` exceeds `N`.
- **Verification:** The `ecall` instructions are included to print the final result to the simulator console and then exit, demonstrating the code's correctness. For N=10, the code correctly outputs 165.

---

2. Factorial of an Integer

Problem: Write a RISC V program using instructions in the RISC V ISA to calculate the factorial of any positive integer N ¡ 10.

RISC-V Code

```
.data
N: .word 5 # Example: N=5. 5! = 120

.text
main:
    lw   x5, N(x0)     # x5 (t0) = N (our counter i)
    li   x6, 1         # x6 (t1) = result
    li   x8, 1         # x8 (t3) = constant 1 for comparison

loop:
    ble  x5, x8, done  # if (i <= 1) goto done
    mul  x6, x6, x5    # result = result * i
    addi x5, x5, -1   # i = i - 1
    j    loop

done:
    # Print the final result
    mv   a0, x6        # Move result to a0 for printing
    li   a7, 1         # ecall for print integer
    ecall

    # Exit program
    li   a7, 10        # ecall for exit
    ecall
```

Discussion

- **Register Allocation:**
  - `x5 (t0)`: Holds `N` and is used as the countdown register `i`.

  - `x6 (t1)`: The accumulator for the `result`, initialized to 1.
  - `x8 (t3)`: Holds the constant 1, used for the loop-termination check.

- **Logic:** This program uses a simple countdown loop. The result is initialized to 1. The loop multiplies the `result` by the current value of `i` (`x5`), then decrements `i`.
- **Efficiency:** The loop condition `ble x5, x8, done` (Branch if Less Than or Equal) checks if `i <= 1`. This correctly terminates the loop, as multiplying by 1 is unnecessary. This logic also inherently handles the edge cases:
  - If N=1, the loop is skipped, and the result is 1.
  - If N=0, the loop is skipped, and the result is 1.
- **Instructions:** This implementation is minimal and requires only 8 instructions for the main logic. For N=5, the code correctly outputs 120.

---

3. Sum of Prime Numbers

Problem: Write a RISC V program using instructions in the RISC V ISA to calculate the sum of all prime numbers less than a given integer N where N ¡ 100.

RISC-V Code

```
.data
N: .word 20 # Example N=20. Primes: 2, 3, 5, 7, 11, 13, 17, 19
           # Sum = 77

.text
main:
    lw   x5, N(x0)     # x5 (t0) = N (the limit)
    li   x8, 0         # x8 (t3) = sum = 0
    li   x6, 2         # x6 (t1) = i (outer loop, number to check)

outer_loop:
    bge  x6, x5, done  # if (i >= N) goto done
    li   x7, 2         # x7 (t2) = j (inner loop, divisor)

inner_loop:
    # If j == i, we've checked all divisors, so it's prime
    beq  x7, x6, add_prime

    rem  x10, x6, x7   # x10 (t5) = i % j
    beq  x10, x0, next_i # if (rem == 0), i is divisible, not prime
                         # so skip to the next i

    addi x7, x7, 1    # j++
    j    inner_loop

add_prime:
    add  x8, x8, x6    # sum = sum + i (it's prime)

next_i:
    addi x6, x6, 1    # i++
    j    outer_loop

done:
    # Print the final sum
    mv   a0, x8
    li   a7, 1
```

```
    ecall

    # Exit program
    li   a7, 10
    ecall
```

Discussion

- **Register Allocation:**
  - `x5 (t0)`: Holds the limit `N`.
  - `x6 (t1)`: The outer loop counter `i`, which iterates from 2 to N-1.
  - `x7 (t2)`: The inner loop counter `j`, which iterates from 2 to `i`.
  - `x8 (t3)`: The accumulator for the `sum` of primes.
  - `x10 (t5)`: A temporary register to hold the remainder of `i % j`.
- **Logic:** A nested loop structure is used. The `outer_loop` iterates through all candidate numbers `i` to be checked. The `inner_loop` checks if `i` is prime by attempting to divide it by every integer `j` from 2 up to `i-1`.
- **Efficiency:**
  - The `rem` instruction (from the 'M' extension) is used to find the remainder.
  - If a remainder is 0 (`beq x10, x0, next_i`), the number is not prime, and we immediately break the inner loop and jump to the next `i`.
  - If the inner loop completes without finding a divisor (i.e., `j` reaches `i`), the `beq x7, x6, add_prime` branch is taken, and `i` is added to the sum.
  - This implementation avoids using a separate "is_prime" flag register by branching directly, which saves instructions.
- **Verification:** For N=20, the code correctly outputs 77.

---

4. Sum of a Geometric Series

Problem: Write a RISC V program that calculates the sum of N terms in a geometric series where a = 1 and r = -3.

RISC-V Code

```
.data
N: .word 4 # Example: N=4. Sum = 1 + (-3) + 9 + (-27) = -20

.text
main:
    lw   x5, N(x0)     # x5 (t0) = N (number of terms)
    li   x6, 0         # x6 (t1) = i (loop counter)
    li   x7, 0         # x7 (t2) = sum
    li   x8, 1         # x8 (t3) = current term (starts at a=1)
    li   x9, -3        # x9 (t4) = r (the ratio)

loop:
    bge  x6, x5, done  # if (i >= N) goto done
    add  x7, x7, x8    # sum = sum + term
    mul  x8, x8, x9    # term = term * r (for next iteration)
    addi x6, x6, 1    # i = i + 1
    j    loop

done:
```

```
# Print the final sum
mv   a0, x7
li   a7, 1
ecall

# Exit program
li   a7, 10
ecall
```

Discussion

- **Register Allocation:**
  - `x5 (t0)`: Holds the number of terms, `N`.
  - `x6 (t1)`: The loop counter `i`, starting from 0.
  - `x7 (t2)`: The accumulator for the `sum`.
  - `x8 (t3)`: Holds the current `term` of the series (ar^i). It is initialized to $a = 1$.
  - `x9 (t4)`: Holds the constant ratio $r = -3$.
- **Logic:** This code directly implements the iterative summation $S_N = \sum_{i=0}^{N-1} ar^i$.
- **Efficiency:** The loop runs `N` times (from `i=0` to `N-1`). In each iteration, it adds the current `term` to the `sum`, and then calculates the `term` for the *next* iteration by multiplying by `r`. This is a very straightforward and minimal implementation.
- **Verification:** For N=4, the series is $1 + (-3) + 9 + (-27)$. The code correctly calculates and prints the sum, -20.

Part II: Analysis of "Computing's Energy Problem"

This section provides answers to questions based on the ISSCC 2014 Keynote by Professor Mark Horowitz, "Computing's Energy Problem (and what we can do about it)".

1.1 How does Technology Scaling decrease the cost of Computing? How do reductions in the cost of manufacturing a transistor enable widespread use of computing devices?

Technology scaling has decreased the cost of computing by making it possible to fabricate chips with millions of transistors at nearly no cost. This massive reduction in cost per transistor has made computing so inexpensive that it can be "included in almost anything". As a direct result, computing devices have become widespread, surrounding us in our homes, cars, workplaces, and daily appliances.

1.2 Why did scaling processor clock frequency become more difficult in the last 15 years? How did Power dissipation become the primary constraint on server CPU performance?

Scaling processor clock frequency became difficult because processors "hit the power wall" for air cooling (around 100W) in the early 2000s. This power wall was a result of two factors:

1. Power density had been growing exponentially because clock frequencies were scaled up faster than constant-field scaling dictated.
2. Voltage scaling, a key technique for managing power, also slowed down because it was no longer possible to lower the transistor threshold voltage without causing massive leakage currents.

Since the chips could no longer be cooled effectively at higher frequencies, power dissipation became the "principal constraint on performance", forcing designers to stop increasing frequencies and instead move to multi-core designs.

1.3 Why is Moore's Law slowing down? Why did Dennard Scaling end?

The paper explains that **Dennard Scaling** (which stated power density remains constant as transistors get smaller) ended because it became impossible to continue scaling (lowering) the transistor's threshold voltage as its physical dimensions shrank. Attempting to lower the threshold voltage further resulted in unacceptably high leakage currents. This breakdown in voltage scaling is what stopped the trend of constant power density. The paper notes that even as Dennard scaling broke down, the number of transistors continued to follow **Moore's Law**, but the inability to manage power (due to the end of Dennard scaling) is what created the energy problem.

1.4 Why is the energy consumption by Memory substantial?

Memory energy consumption is substantial for two main reasons: on-die caches and off-die DRAM.

- **On-Die Caches:** Caches and register files can dissipate over 50% of the processor's total die energy. The leakage power of a modern last-level cache alone can be "larger than the power of a simple core running full out".
- **Off-Die DRAM:** Accessing external DRAM is "a couple of orders-of-magnitude" more energy-intensive than a simple processor operation (like a 10pJ functional operation). A large part of this cost comes from the "very energy-inefficient I/O" systems used by DRAM, which consume both dynamic power (over 20pJ/bit) and static power to remain active.

1.5 What solutions to Computing's Energy Problem does Professor Mark Horowitz's envision?

Professor Horowitz envisions a fundamental shift away from general-purpose computing and towards **specialization**.

- The key solution is to create "applications and hardware which are better matched to the task and each other". This means using **application-specific accelerators** (like GPUs, image processors, or codecs), which are specialized hardware engines that can be 2 to 3 orders-of-magnitude more energy-efficient than a general-purpose processor for a given task.
- To make this specialization possible, he states that "our principal task will be to enable a larger group of application experts to participate" in hardware design. This requires the creation of new **design tools**, such as hardware generators (like Chisel) and domain-specific languages (DSLs), that allow people who are not hardware experts to build their own efficient, specialized systems.

Problem 2:

This assignment requires you to review several references on RISCV beginning with a summary transcript [2] of the Debate on Proprietary Vs Open Source Instruction Sets at the 4th Workshop on Computer Architecture Research Directions, June 2015 sponsored by the ACM. This Debate between Professor David Patterson (author of the textbook you are using) and Dave Christie of AMD highlights all of the key technical and business arguments for and against an Open-Source ISA such as RISC V as of 2015 (the same year the RISC V Foundation was established). A Technical Report from EECS UC Berkeley highlights the technical reasons for Open ISAs [3] providing a more detailed discussion on the advantages offered by open source ISAs

(1) Articulate your views on the topics debated in [2]. Justify your views.

After reviewing the 2015 debate between Dave Christie and David Patterson, my view aligns with Professor Patterson's position that the future of computing, especially in the "21st-century" era of Systems-on-a-Chip (SoCs), is best served by a free, open-source, and modular ISA. While Mr. Christie's arguments about the stability and formidable power of existing ecosystems are valid, they primarily represent a defense of the status quo rather than a compelling vision for future innovation.

My justification is based on the following points from the debate:

1. The Need for Architectural Innovation Christie's argument that the ISA is secondary to the ecosystem overlooks a critical shift in the industry. As Patterson points out, Moore's Law is ending, meaning performance and efficiency gains must now come from architectural innovation, not just from semiconductor physics. This innovation is taking the form of domain-specific accelerators and coprocessors. This is precisely where proprietary, monolithic ISAs fail. Patterson highlights that ARMv8, for example, has no defined coprocessor interface, which is a "glaring omission" for the future of architecture. An open, modular ISA like RISC-V, which is "designed for the 21st century", provides a minimal base and reserved space for exactly this kind of domain-specific extension.

2. Efficiency and Modularity vs. Legacy Baggage Christie praises the "responsible stewardship" of proprietary ISAs, which maintain backward compatibility. However, Patterson provides a powerful counter-argument that this leads to "monolithic" ISAs that only grow over time. His example—that the ARMv8 manual is 5,400 pages long and still lacks 16-bit compressed instructions, leading to larger code size than x86—is a striking critique. A "minimal modular ISA" like RISC-V, which has fewer than 50 base instructions and a standard compact extension, is a more technically sound and efficient foundation for a diverse range of devices, from tiny $1 chips to large servers.

3. Democratizing Design and Security The most compelling argument is the democratization of innovation. Christie questions the demand for custom core design, but Patterson's stance is that the high cost and restrictive nature of proprietary licenses *prevent* innovation and competition. By removing these financial and legal barriers, an open ISA allows academics, startups, and engineers worldwide to design and share cores. This also enables transparency. Patterson's point that an open-source core allows "many more eyeballs" to look for errors or "government... backdoors" is a far superior security model to the "security by obscurity" common in proprietary designs.

In conclusion, the proprietary model was successful for the "monolithic microprocessor era", but the rise of SoCs, the end of Moore's Law, and the need for domain-specific accelerators have created a clear technical and business need for the open, modular, and extensible approach that RISC-V provides.

---

(2) Review and summarize technical reasons for Open-Source ISAs in [3].

The UC Berkeley technical report, "Instruction Sets Should Be Free: The Case For RISC-V," argues that there is "no good technical reason" for ISAs to remain proprietary. It outlines the technical failings of proprietary models and the specific technical advantages of a modern, open-source ISA like RISC-V.

Problems with Proprietary ISAs

The report first argues that the reasons for ISAs being proprietary are purely for business and historical, not technical, reasons:

- **Block Innovation:** Proprietary ISAs stifle innovation and competition by using expensive licenses ($1M-$10M) and long negotiations (6-24 months) that exclude academics and small companies.
- **Ecosystem Argument is Flawed:** The claim that ISA owners are needed to build the software ecosystem is false; "outsiders build almost all of the software" for them.
- **No Technical Superiority:** The most popular ISAs (x86, ARM) are not "ISA exemplars"; they are just commercially successful.
- **Compatibility is Solvable:** Open organizations have long been able to develop mechanisms to ensure compatibility with hardware standards (e.g., IEEE 754 floating point, Ethernet).
- **Risk of Obsolescence:** A proprietary ISA is tied to the fate of a single company and can be terminated if that company fails (e.g., DEC's Alpha and VAX).

Technical Advantages of an Open-Source ISA (RISC-V)

The report presents RISC-V as a "clean-slate" design that embodies the technical advantages of an open ISA built for modern computing:

- **Modularity (Base-plus-Extension):** RISC-V is not a single, monolithic ISA. It is designed as a small, simple, and stable base integer ISA (RV32I) with a set of standard, optional extensions that can be added as needed (e.g., 'M' for Multiply-Divide, 'F'/'D' for Floating Point).

- **Extensibility for Specialization:** The ISA explicitly reserves opcode space for "entirely new opcodes". This allows companies to add their own custom, application-specific accelerators without conflicting with the standard ISA, a crucial feature for SoCs.
- **Clean Design:** Being a new design, RISC-V learns from the mistakes of older RISC ISAs, such as avoiding "microarchitectural designs to affect the ISA" (e.g., delayed branches).
- **Code Density (Compact Encoding):** The ISA was designed to solve the code size issue of early RISCs. It features a standard 'C' extension that provides compressed 16-bit instructions, offering code size "smaller than 80x86".
- **Future-Proof Addressing:** The ISA was designed from the start to support 32-bit, 64-bit, and 128-bit address spaces. This avoids the "one ISA mistake from which it is hard to recover", which is outgrowing an address space.
- **Transparency and Shared Effort:** An open ISA enables shared open core designs, which leads to "shorter time to market, lower cost from reuse, fewer errors given many more eyeballs" and transparency that makes it difficult to insert "secret trap doors".