# Orca Lisp Processor

*A RISC-V based microcomputer for LISP*

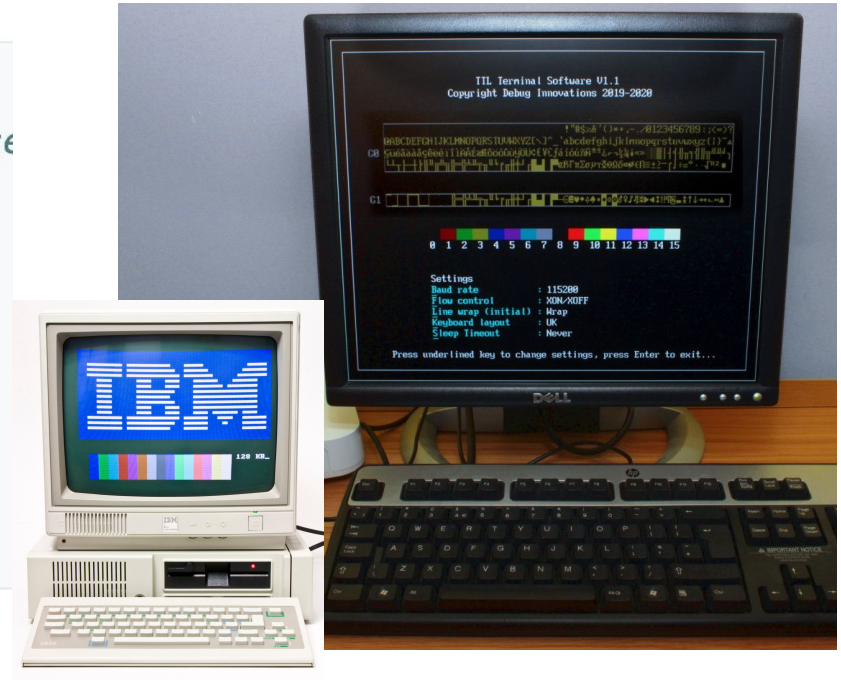
Haoran Wen and Kosi Nwabueze

# Microcomputers + LISP = Orca!

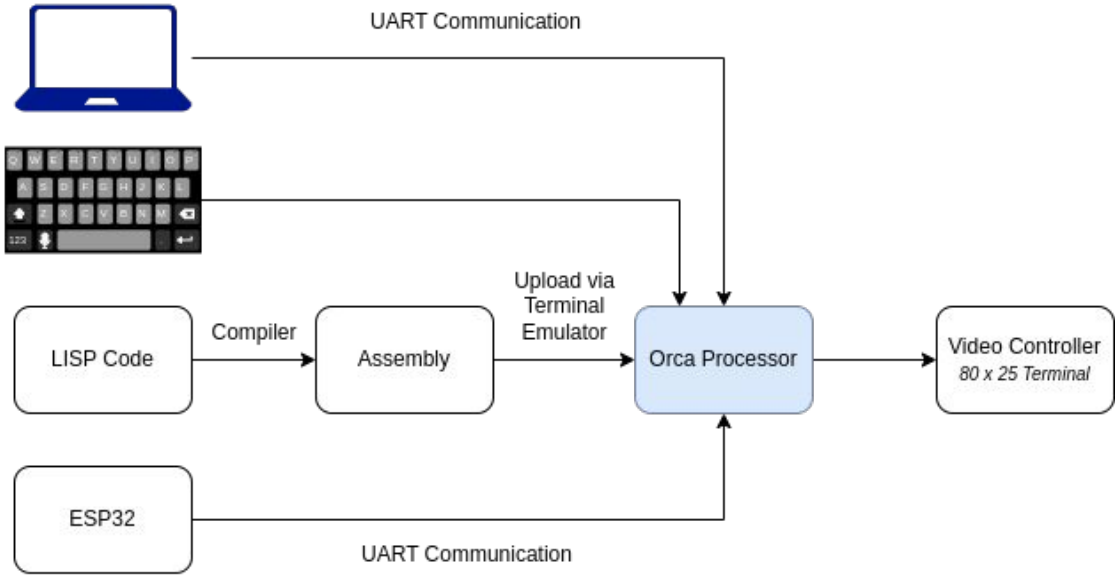
```
;; Building a list of squares from 0 to 9:
;; Note: loop is simply an arbitrary symbol use

(define (list-of-squares n)
  (let loop ((i n) (res '()))
    (if (< i 0)
        res
        (loop (- i 1) (cons (* i i) res)))))

(list-of-squares 9)
===> (0 1 4 9 16 25 36 49 64 81)
```



# Design Overview

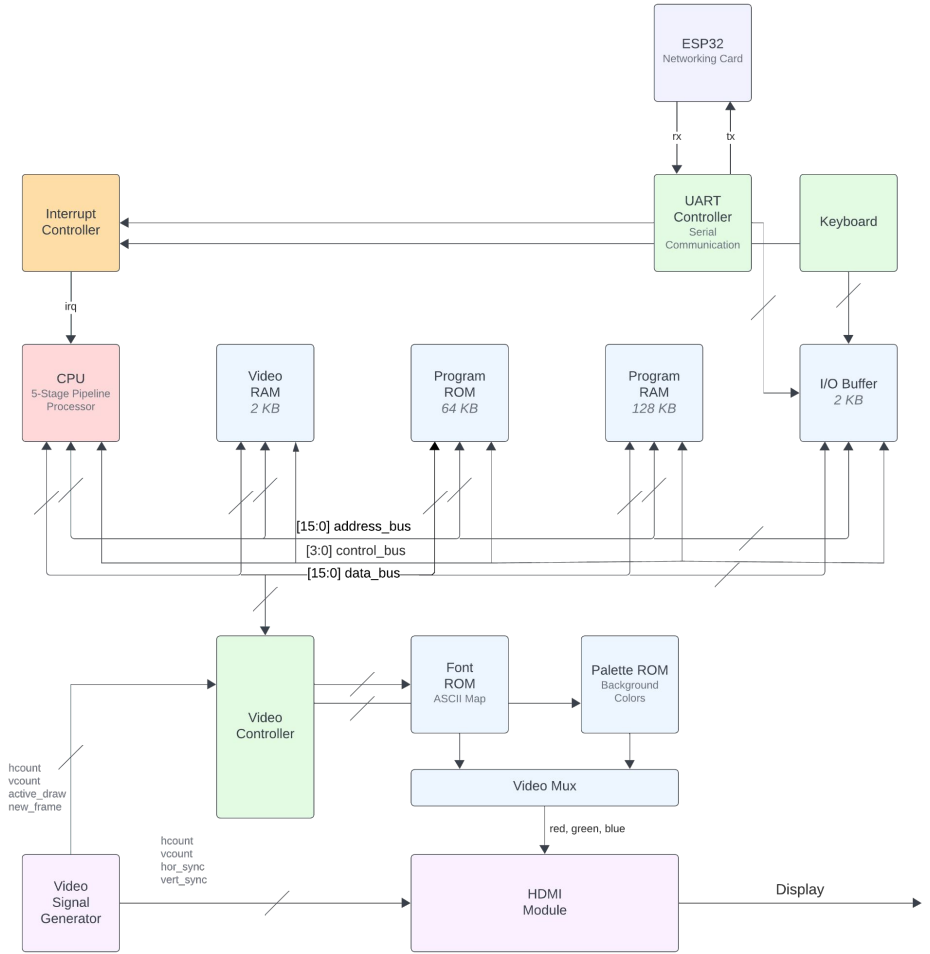


# Design Overview

- Design and implement peripheral devices (e.g. video controller / keyboard) that would be common in early personal microcomputers (such as the IBM PC)

- Design and implement a RISC-V based processor with optimizations on both hardware and software level to run LISP code efficiently

# Block Diagram



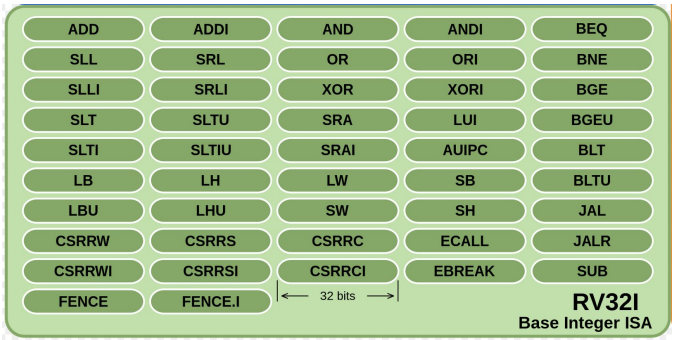
# Changes From Initial Proposal

- Switched compiler from targeting native RISC-V to virtual machine bytecode (virtual machine has hardware optimizations)
- Adding support for tagged architecture for the processor

# Processor

# RISC-V Instruction Set

- Based on RV32IMF with Zicsr extensions
  - 32-bit integer instructions
  - I = base instruction set architecture
  - M = multiply and divide instructions (extended ALU)
  - F = floating point instructions
  - Zicsr = control and status registers

# Typed Architectures: Architectural Support for Lightweight Scripting

Channoh Kim[1*] Jaehyeok Kim[1*] Sungmin Kim[1] Dooyoung Kim[1] Namho Kim[2]
Gitae Na[1] Young H. Oh[1] Hyeon Gyu Cho[1] Jae W. Lee[2]

# Extensions

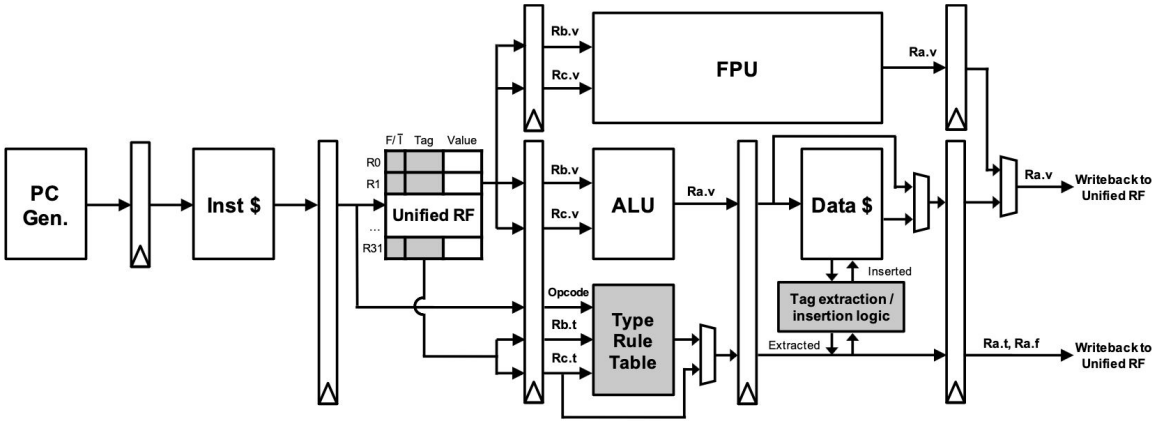



Figure 4: Pipeline structure augmented with Typed Architecture

# Extensions

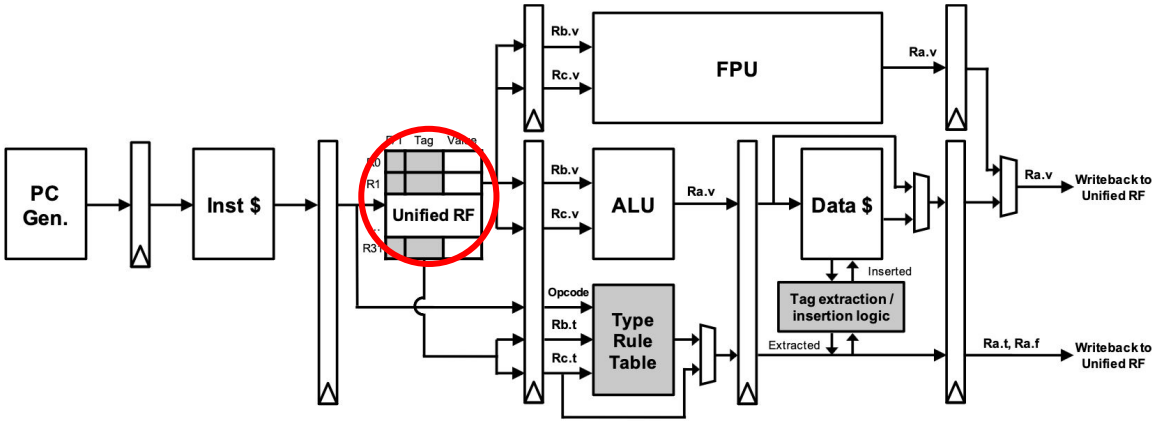



Figure 4: Pipeline structure augmented with Typed Architecture

# Extensions

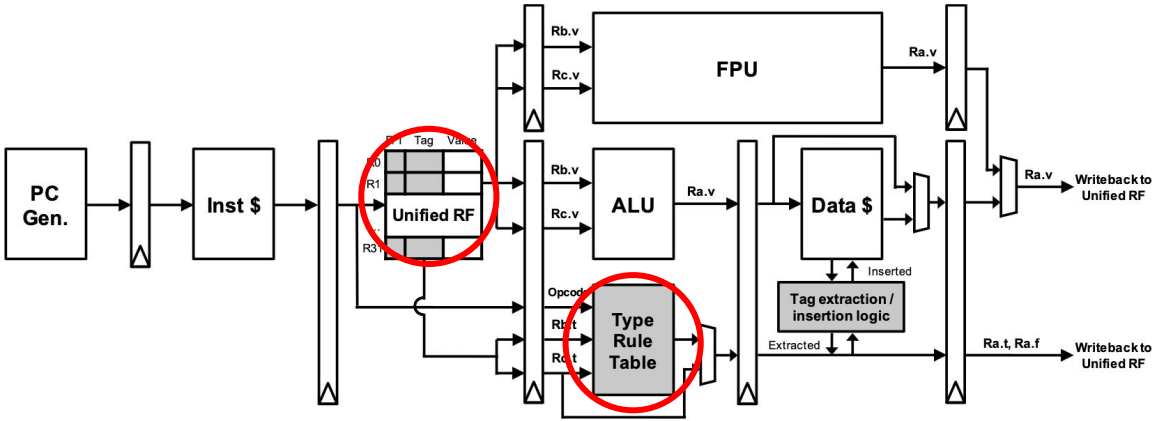



Figure 4: Pipeline structure augmented with Typed Architecture

# Extensions

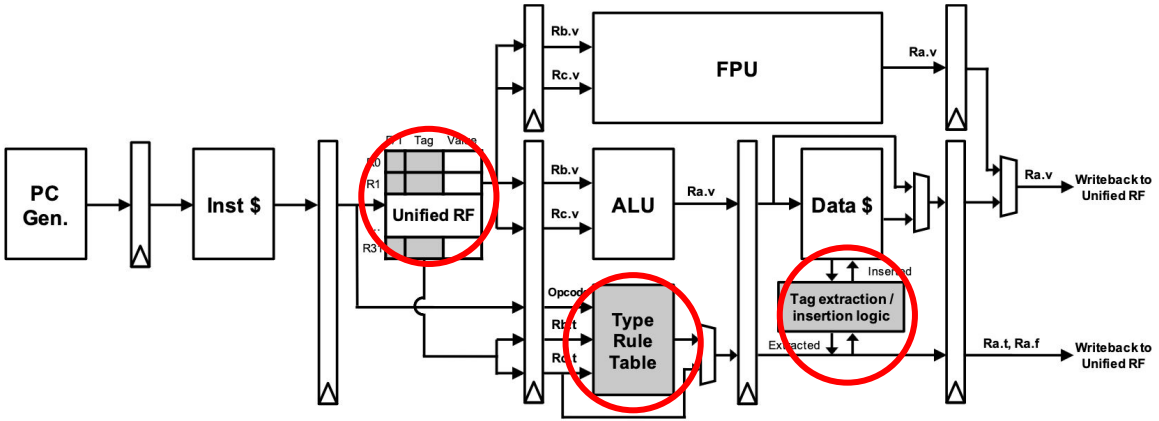



Figure 4: Pipeline structure augmented with Typed Architecture

# Extensions

| Instruction | Operation | Type Tag Handling | Description |
|---|---|---|---|
| **Memory Instructions** | | | |
| tld Rc, imm(Ra) | Rc.v ← Mem[Ra.v+imm] | Rc.t ← extract(Mem[Ra.v+imm+$R_{offset}$ ]) | Load dword with tag |
| tsd Rc, imm(Ra) | Mem[Ra.v+imm] ← Rc.v | Mem[Ra.v+imm+$R_{offset}$ ] ← insert(Rc.t) | Store dword with tag |
| **Arithmetic and Logical Instructions** | | | |
| xadd Ra,Rb,Rc | $Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} + Rc.v_{[63:0]}$ | **if** (type hits) | Add (dword) |
| xsub Ra,Rb,Rc | $Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} - Rc.v_{[63:0]}$ | Rc.t ← OutputType(op, Ra.t, Rb.t) | Subtract (dword) |
| xmul Ra,Rb,Rc | $Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} \times Rc.v_{[63:0]}$ | **else** NextPC ← $R_{hdl}$ | Multiply (dword) |
| **Configuration Instructions** | | | |
| setoffset Ra | $R_{offset}$ ← Ra.v | - | Set Ra to $R_{offset}$ |
| setmask Ra | $R_{mask}$ ← Ra.v | - | Set Ra to $R_{mask}$ |
| setshift Ra | $R_{shift}$ ← Ra.v | - | Set Ra to $R_{shift}$ |
| set_trt Ra | TypeRuleTable.push.data(Ra.v) | - | Push Ra to Type Rule Table (TRT) |
| flush_trt | TypeRuleTable.flush() | - | Flush TRT |
| **Miscellaneous** | | | |
| thdl label | $R_{hdl}$ ← NextPC + (disp << 2) | - | Store `label` addr to $R_{hdl}$ |
| tchk Rb,Rc | - | NextPC ← (type hits) ? PC + 4 : $R_{hdl}$ | Look up TRT with two source type tags (Ra.t and Rb.t) |
| tget Ra,Rb | Ra.v ← ZeroExt64(Rb.t) | - | Copy the type field of Rb to Ra.v (64-bit zero extended) |
| tset Ra,Rb | $Rb.t \leftarrow Ra.v_{[7:0]}$ | - | Copy the least significant byte of Ra.v to Rb.t |

Table 2: Description of Extended ISA (64-bit)

# Extensions

| Instruction | Operation | Type Tag Handling | Description |
|---|---|---|---|
| **Memory Instructions** | | | |
| tld Rc, imm(Ra) | Rc.v ← Mem[Ra.v+imm] | Rc.t ← extract(Mem[Ra.v+imm+$R_{offset}$ ]) | Load dword with tag |
| tsd Rc, imm(Ra) | Mem[Ra.v+imm] ← Rc.v | Mem[Ra.v+imm+$R_{offset}$ ] ← insert(Rc.t) | Store dword with tag |
| **Arithmetic and Logical Instructions** | | | |
| xadd Ra,Rb,Rc | $Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} + Rc.v_{[63:0]}$ | **if** (type hits) | Add (dword) |
| xsub Ra,Rb,Rc | $Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} - Rc.v_{[63:0]}$ | Rc.t ← OutputType(op, Ra.t, Rb.t) | Subtract (dword) |
| xmul Ra,Rb,Rc | $Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} \times Rc.v_{[63:0]}$ | **else** NextPC ← $R_{hdl}$ | Multiply (dword) |
| **Configuration Instructions** | | | |
| setoffset Ra | $R_{offset}$ ← Ra.v | - | Set Ra to $R_{offset}$ |
| setmask Ra | $R_{mask}$ ← Ra.v | - | Set Ra to $R_{mask}$ |
| setshift Ra | $R_{shift}$ ← Ra.v | - | Set Ra to $R_{shift}$ |
| set_trt Ra | TypeRuleTable.push.data(Ra.v) | - | Push Ra to Type Rule Table (TRT) |
| flush_trt | TypeRuleTable.flush() | - | Flush TRT |
| **Miscellaneous** | | | |
| thdl label | $R_{hdl}$ ← NextPC + (disp << 2) | - | Store `label` addr to $R_{hdl}$ |
| tchk Rb,Rc | - | NextPC ← (type hits) ? PC + 4 : $R_{hdl}$ | Look up TRT with two source type tags (Ra.t and Rb.t) |
| tget Ra,Rb | Ra.v ← ZeroExt64(Rb.t) | - | Copy the type field of Rb to Ra.v (64-bit zero extended) |
| tset Ra,Rb | $Rb.t \leftarrow Ra.v_{[7:0]}$ | - | Copy the least significant byte of Ra.v to Rb.t |

Table 2: Description of Extended ISA (64-bit)

# Stages

- ***Stage 1***: 6.004-style Single-Cycle Design
- ***Stage 2:*** Pipeline Support
- ***Stage 3:*** Typed architecture

| Instruction | Syntax | Description | Execution |
|---|---|---|---|
| LUI | `lui rd, luiConstant` | Load Upper Immediate | `reg[rd] <= luiConstant « 12` |
| JAL | `jal rd, label` | Jump and Link | `reg[rd] <= pc + 4`<br>`pc <= label` |
| JALR | `jalr rd, offset(rs1)` | Jump and Link Register | `reg[rd] <= pc + 4`<br>`pc <= {(reg[rs1] + offset)[31:1], 1'b0}` |
| BEQ | `beq rs1, rs2, label` | Branch if = | `pc <= (reg[rs1] == reg[rs2]) ? label: pc + 4` |
| BNE | `bne rs1, rs2, label` | Branch if ≠ | `pc <= (reg[rs1] != reg[rs2]) ? label: pc + 4` |
| BLT | `blt rs1, rs2, label` | Branch if < (Signed) | `pc <= (reg[rs1] <$_s$ reg[rs2]) ? label: pc + 4` |
| BGE | `bge rs1, rs2, label` | Branch if ≥ (Signed) | `pc <= (reg[rs1] >=$_s$ reg[rs2]) ? label: pc + 4` |
| BLTU | `bltu rs1, rs2, label` | Branch if < (Unsigned) | `pc <= (reg[rs1] <$_u$ reg[rs2]) ? label: pc + 4` |
| BGEU | `bgeu rs1, rs2, label` | Branch if ≥ (Unsigned) | `pc <= (reg[rs1] >=$_u$ reg[rs2]) ? label: pc + 4` |
| LB | `lb rd, offset(rs1)` | Load Byte | `reg[rd] <= signExtend(mem[addr])` |
| LH | `lh rd, offset(rs1)` | Load Half Word | `reg[rd] <= signExtend(mem[addr + 1: addr])` |
| LW | `lw rd, offset(rs1)` | Load Word | `reg[rd] <= mem[addr + 3: addr]` |
| LBU | `lbu rd, offset(rs1)` | Load Byte (Unsigned) | `reg[rd] <= zeroExtend(mem[addr])` |
| LHU | `lhu rd, offset(rs1)` | Load Half Word (Unsigned) | `reg[rd] <= zeroExtend(mem[addr + 1: addr])` |
| SB | `sb rs2, offset(rs1)` | Store Byte | `mem[addr] <= reg[rs2][7:0]` |
| SH | `sh rs2, offset(rs1)` | Store Half Word | `mem[addr + 1: addr] <= reg[rs2][15:0]` |
| SW | `sw rs2, offset(rs1)` | Store Word | `mem[addr + 3: addr] <= reg[rs2]` |
| ADDI | `addi rd, rs1, constant` | Add Immediate | `reg[rd] <= reg[rs1] + constant` |
| SLTI | `slti rd, rs1, constant` | Compare < Immediate (Signed) | `reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0` |
| SLTIU | `sltiu rd, rs1, constant` | Compare < Immediate (Unsigned) | `reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0` |
| XORI | `xori rd, rs1, constant` | Xor Immediate | `reg[rd] <= reg[rs1] ^ constant` |
| ORI | `ori rd, rs1, constant` | Or Immediate | `reg[rd] <= reg[rs1] \| constant` |
| ANDI | `andi rd, rs1, constant` | And Immediate | `reg[rd] <= reg[rs1] & constant` |
| SLLI | `slli rd, rs1, shamt` | Shift Left Logical Immediate | `reg[rd] <= reg[rs1] « shamt` |
| SRLI | `srli rd, rs1, shamt` | Shift Right Logical Immediate | `reg[rd] <= reg[rs1] »$_u$ shamt` |
| SRAI | `srai rd, rs1, shamt` | Shift Right Arithmetic Immediate | `reg[rd] <= reg[rs1] »$_s$ shamt` |
| ADD | `add rd, rs1, rs2` | Add | `reg[rd] <= reg[rs1] + reg[rs2]` |
| SUB | `sub rd, rs1, rs2` | Subtract | `reg[rd] <= reg[rs1] - reg[rs2]` |
| SLL | `sll rd, rs1, rs2` | Shift Left Logical | `reg[rd] <= reg[rs1] « reg[rs2][4:0]` |
| SLT | `slt rd, rs1, rs2` | Compare < (Signed) | `reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0` |
| SLTU | `sltu rd, rs1, rs2` | Compare < (Unsigned) | `reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0` |
| XOR | `xor rd, rs1, rs2` | Xor | `reg[rd] <= reg[rs1] ^ reg[rs2]` |
| SRL | `srl rd, rs1, rs2` | Shift Right Logical | `reg[rd] <= reg[rs1] »$_u$ reg[rs2][4:0]` |
| SRA | `sra rd, rs1, rs2` | Shift Right Arithmetic | `reg[rd] <= reg[rs1] »$_s$ reg[rs2][4:0]` |
| OR | `or rd, rs1, rs2` | Or | `reg[rd] <= reg[rs1] \| reg[rs2]` |
| AND | `and rd, rs1, rs2` | And | `reg[rd] <= reg[rs1] & reg[rs2]` |

# Peripherals

# Video

- 80 x 25 text-mode video interface
  - Utilizes a video BRAM containing the text buffer to be printed on the screen
  - That then is connected to a BROM 256 entries of 8 x 16 pixels representing the dot matrix of how the ascii character is printed on screen

| Bit(s) | Value |
|---|---|
| 0-7 | ASCII code point |
| 8-11 | Foreground color |
| 12-14 | Background color |
| 15 | Blink |

# Serial / UART

- We will implement a UART protocol to interface and communicate with external connections
  - Computer terminal emulator to send characters over to the processor to print to terminal
  - Potential communication with the reach goal ESP32 implementation (for internet networking)

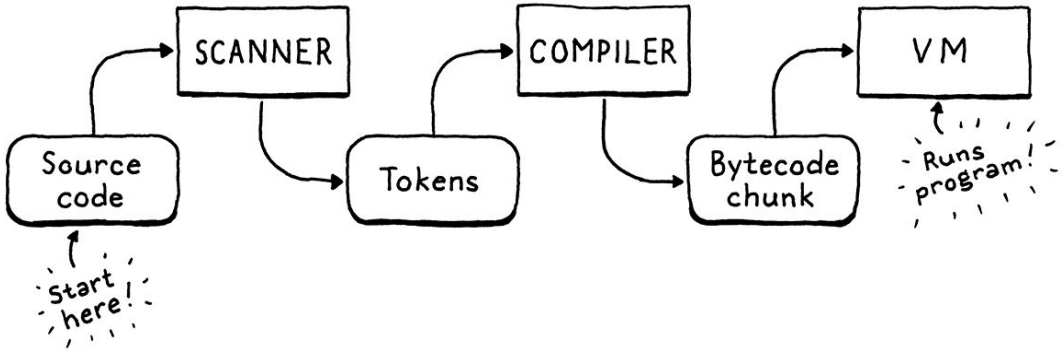
# ESP32 [Stretch Goal]

- As stretch goal for peripheral connections, we will implement computer networking by interfacing with a WiFi-enabled ESP32
    - Connection with the processor will be done via UART protocol
    - Software can use MMIO to interface with data from ESP32

# Software

# Compiler

- ***Source Language***: Scheme (R5RS-subset)
- Bytecode compiler (written in Python) running on a VM (written in C)
- Using *specialized* xadd, xsub, xmul instructions for polymorphic arithmetic



# Logistics

| Week | Kosi | Hao |
| --- | --- | --- |
| Oct 26 - Nov 2 (Wk 1) | ISA Design + Refining Project Scope | ISA Design + Refining Project Scope |
| Nov 2 - Nov 9 (Wk 2) | Video Controller | Processor |
| Nov 9 - Nov 16 (Wk 3) | Video Controller | Processor |
| Nov 16 - Nov 23 (Wk 4) | UART | Processor |
| Nov 23 - Nov 30 (Wk 5) | Compiler | PS/2 Keyboard |
| Nov 30 - Dec 7 (Wk 6) | Compiler | ESP32 Network Connection |
| Dec 7 - Dec 14 (Wk 7) | Final Report | Final Report |

# Evaluation

*Is Orca any good?*

***Performance:*** Compare cycle count for programs using specialized instructions vs programs not using specialized instructions for multiple benchmark programs (n-queens, n-body, matrix multiplication, etc.)

***Correctness***: Running standard RISC-V unit tests to make sure core is ISA compliant

***Manual Testing***: Processor can communicate with video pipeline and peripheral devices

# Checkoff Checklist

# Commitment

**CPU:** Implementing a single cycle RISC-V architecture with RV32I base instruction set.

**Video Controller:** A video controller that supports 80x25 VGA text mode. Video uses HDMI. Output can be displayed

**UART Controller:** Successfully implementing a simple UART serial controller communicating to the processor through memory-mapped I/O. Communication with an external laptop.

**should be completed by end of week 4*

# Goal

**Compiler**: Implementing a bytecode compiler for LISP executing on a virtual machine running on the CPU.

**CPU (Tagged Architecture):** Tagged architecture to support efficient handling of typed arithmetics on a hardware level.

**CPU (Pipelined):** Pipelining post tagged architecture implementation to improve processor instruction throughput

# Stretch

**Keyboard:** Add support for keyboard.

**Network Card via ESP32:** Allow UART communication with a ESP32 with a written script to fetch some data over the internet and displaying it via the video controller.

**SD Card Memory** *Stretchiest of the Stretch. Stretched beyond our comprehension*: Add support for additional memory via a SD card. Support uploading and reading file to the SD card through software implementation.

# Q&A

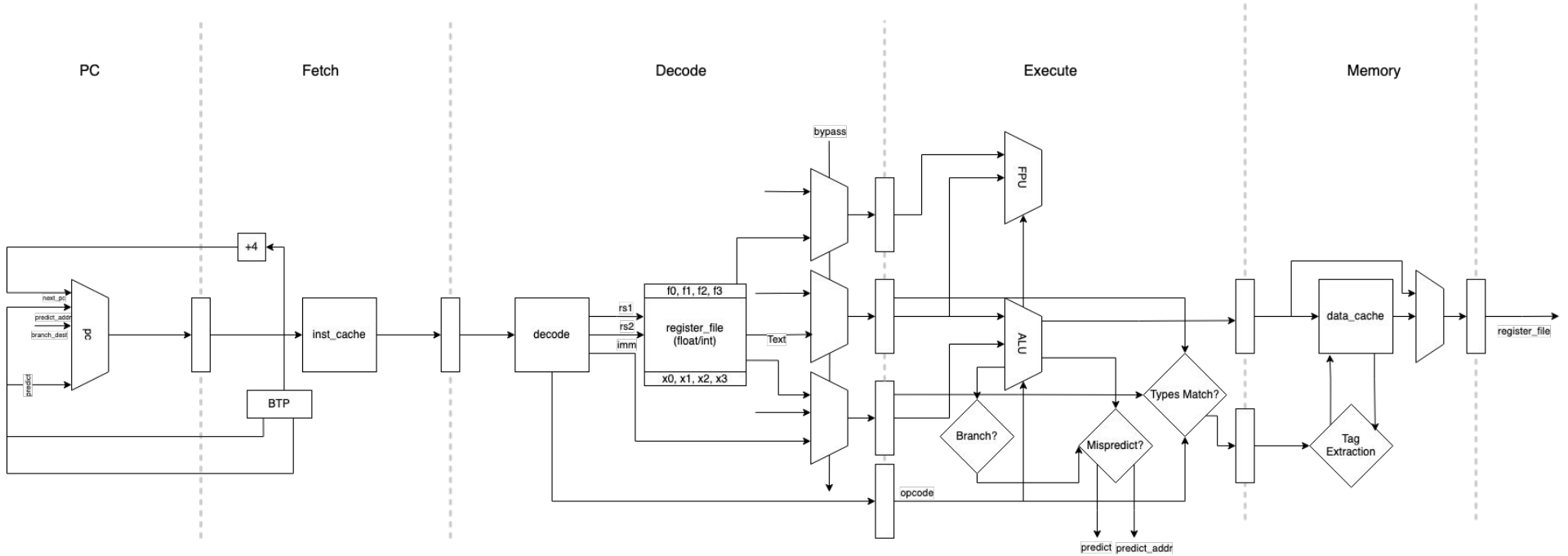
Got any questions?

# References

[1] *Hafer, Plankl, and Schmitt.* COLIBIRI: A Coprocessor for LISP based on RISC.
https://link.springer.com/chapter/10.1007/978-1-4615-3752-6_5

[2] *Kim and et al.* Typed Architectures: Architectural Support for Lightweight Scripting.
https://channoh.github.io/pubs/asplos17-typed.pdf

[3] *Cui, Li, and Wei.* RISC-V Instruction Set Architecture Extensions: A Survey.
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10049118


PC
Fetch
Decode
Execute
Memory
+4
next_pc
predict_addr
branch_dest
predict
pc
BTP
inst_cache
decode
rs1
rs2
imm
f0, f1, f2, f3
register_file
(float/int)
x0, x1, x2, x3
Text
bypass
FPU
ALU
Branch?
Mispredict?
Types Match?
opcode
predict
predict_addr
data_cache
Tag
Extraction
register_file