Pranav Shekar
Will Lillis
CS 535

ISA Project Proposal

REVISED PROPOSAL:

Our architecture is *general purpose*. Its *distinguishing features* will be a configurable cache (Number of levels, size), memory-mapped input/output, and a branch predictor.

The *word size* is 32 bits. The *supported data types* will be integers (signed/unsigned), IEEE-754 floats, and raw bytes.

The *operations supported on integers* will be add, subtract, multiply, divide, modulo, bit shifts, XOR, AND, OR, and comparisons. The *operations supported on floats* will be add, subtract, multiply, divide, and comparisons.

Special features:
We will attempt to build a GUI debugger with the following features:

- Configurable size and depth of cache
    - 0-5 levels of cache, variable size on a per-level basis, increasing as level gets closer to main memory at a regular interval
- Hot loading/ storing of machine state
- Breakpoints
- Single step execution
- Run execution
- Branch prediction

For its *registers*, our architecture will support three different logical “banks”: general purpose, floating point, and one or more status/flag registers. There will be 16 general purpose registers (R0-R15), 16 float registers (F0-F15), 1 status/flag (S), and one program counter (P) register. All registers will be 32 bits each. The S register will hold the following status bits:

- EQ (Equal)
- LT (Less than)
- GT (Greater than)
- OF (Overflow)
- SG (Sign, 1 positive, 0 negative)
- ZO (Zero)

We will allow a maximum of 3 operands per instruction, with a *single instruction per word*. The *Princeton Architecture* (unified instruction and data memory) will be used, with an address unit of bytes. We will support *immediate*, *register indirect*, and *pc relative* addressing modes.The cache will be direct mapped, utilizing a write-through no allocate schema. We will utilize the usual 5-stage pipeline for our CPU.

Load/Store Instructions:

- LD8, LD16, and LD32
  - Load 8, 16, or 32 bits from memory to a general purpose register. The remainder of the register will be zeroed out in the case of LD8 and LD16

Instruction Type Field:

- Type 0 (No args)

Type Field (3 bits)	Opcode (1 bit)	Padding (28 bits)

- Type 1 (One immediate arg)

Type Field (3 bits)	Opcode (4 bits)	Immediate Arg (21 bits)	Padding (4 bits)

- Type 2 (Two general purpose registers)

Type Field (3 bits)	Opcode (3 bits)	Register 1 (4 bits)	Register 2 (4 bits)	Padding (18 bits)

- Type 3 (Two floating point registers)

Type Field (3 bits)	Opcode (1 bit)	Register 1 (4 bits)	Register 2 (4 bits)	Padding (20 bits)

- Type 4 (One general purpose register, one immediate)

Type Field (3 bits)	Opcode (4 bits)	Register (4 bits)	Immediate (21 bits)

- Type 5 (Three general purpose register args)

Type Field (3 bits)	Opcode (4 bits)	Register 1 (4 bits)	Register 2 (4 bits)	Register 3 (4 bits)	Padding (13 bits)

- Type 6 (Three floating point register args)

Type Field (3 bits)	Opcode (2 bits)	Register 1 (4 bits)	Register 2 (4 bits)	Register 3 (4 bits)	Padding (15 bits)

Instruction Encodings:

- 3 bits for the type field
- 1-4 bits for instruction
- Variable number of bits for operands
    - 4 bits to specify a register
    - 21 bit width for immediates
- Instructions will be of the form MNEMONIC <Arg1>, <Arg2>, <Arg3>

Instructions:
We offer the instructions split up by their type. Under each type, the mnemonic and a brief description are provided

******All immediate addresses will be unsigned values and all PC-relative addresses will be signed values******

- ***Type 0***
    - RET
        - Return to the address that is saved in the PC register (likely R15)
    - HALT
        - Halts execution
- ***Type 1***
    - CALL <Immediate Address>
        - Save the PC in a general purpose register (likely R15) and jump to the immediate address provided as an argument
    - JE <Immediate Address>
        - Sets the P register to the immediate address if the equal status bit in the S register is set
    - JNE <Immediate Address>
        - Sets the P register to the immediate address if the equal status bit in the S register is not set
    - *JGT* <Immediate Address>
        - Sets the P register to the immediate address if the greater than status bit in the S register is set
    - *JLT* <Immediate Address>
        - Sets the P register to the immediate address if the less than status bit in the S register is set
    - *JGTE* <Immediate Address>
        - Sets the P register to the immediate address if the equal or greater than status bits in the S register are set
    - *JLTE* <Immediate Address>
        - Sets the P register to the immediate address if the equal or less than status bits in the S register are set
    - IJE <PC-Relative Address>

        - Sets the P register to the immediate address if the equal status bit in the S register is set
    - IJNE <PC-Relative Address>
        - Sets the P register to the immediate address if the equal status bit in the S register is not set
    - *IJGT* <PC-Relative Address>
        - Sets the P register to the immediate address if the greater than status bit in the S register is set
    - *IJLT* <PC-Relative Address>
        - Sets the P register to the immediate address if the less than status bit in the S register is set
    - *IJGTE* <PC-Relative Address>
        - Sets the P register to the immediate address if the equal or greater than status bits in the S register are set
    - *IJLTE* <PC-Relative Address>
        - Sets the P register to the immediate address if the equal or less than status bits in the S register are set

- ***Type 2***
    - CMP8/CMP16/CMP32 Rx, Ry
        - Sets appropriate status bits in the S register by comparing the appropriate number of bits between Rx and Ry
    - LDIN8, LDIN16, and LDIN32 <Rx>, <Ry>
        - Load an 8, 16, or 32 bit value at the address pointed to by Ry, into a general purpose register Rx. The remainder of the register will be zeroed out in the case of LD8 and LD16

- ***Type 3***
    - CMPF <Fx>, <Fy>
        - Sets appropriate status bits in the S register by comparing the appropriate number of bits between Fx and Fy
- ***Type 4***
    - LD8, LD16, and LD32 <Rx>, <Immediate Address>
        - Load 8, 16, or 32 bits from memory to a general purpose register. The remainder of the register will be zeroed out in the case of LD8 and LD16
    - LDI8, LDI16, and LDI32 <Rx>, <Immediate Address>
        - Load an 8, 16, or 32 bit immediate value to a general purpose register. The remainder of the register will be zeroed out in the case of LD8 and LD16
    - ST8, ST16, and ST32 <Rx>, <Immediate Address>
        - Stores 8, 16, or 32 bits from a general purpose register to an immediate address.
    - ADDIM <Rx>, <Immediate Address>

        - Adds the immediate value to the contents of Rx, placing the result into Rx. The appropriate status bits in the S register are set.
- ***Type 5***
    - ADDI <Rx>, <Ry>, <Rz>
        - Adds the contents of Ry and Rz, placing the result in Rx. The appropriate status bits in the S register are set.
    - SUBI <Rx>, <Ry>, <Rz>
        - Subtracts the contents of Rz from Ry. The appropriate status bits in the S register are set.
    - MULI <Rx>, <Ry>, <Rz>
        - Multiplies the contents of Ry and Rz, placing the result in Rx. The appropriate status bits in the S register are set.
    - DIVI <Rx>, <Ry>, <Rz>
        - Divides the contents of Rz from Ry, placing the result in Rx.The appropriate status bits in the S register are set.
    - MODI <Rx>, <Ry>, <Rz>
        - Gets the result of Ry modulo Rz, placing the result in Rx.The appropriate status bits in the S register are set.
    - RBSI <Rx>, <Ry>, <Rz>
        - Performs a right bit shift, where Ry is what we want to start with, Rz is the number of bits to shift, and Rx is the result. The appropriate status bits in the S register are set.
    - XORI <Rx>, <Ry>, <Rz>
        - Performs an XOR operation on Ry with Rz, placing the result in Rx. The appropriate status bits in the S register are set.
    - ANDI <Rx>, <Ry>, <Rz>
        - Performs an AND operation on Ry with Rz, placing the result in Rx.The appropriate status bits in the S register are set.
    - ORI <Rx>, <Ry>, <Rz>
        - Performs an OR operation on Ry with Rz, placing the result in Rx.The appropriate status bits in the S register are set.
    - ADDU <Rx>, <Ry>, <Rz>
        - Adds the contents of Ry and Rz, placing the result in Rx. The appropriate status bits in the S register are set.
    - SUBU <Rx>, <Ry>, <Rz>
        - Subtracts the contents of Rz from Ry. The appropriate status bits in the S register are set.
    - MULU <Rx>, <Ry>, <Rz>
        - Multiplies the contents of Ry and Rz, placing the result in Rx. The appropriate status bits in the S register are set.
    - DIVU <Rx>, <Ry>, <Rz>
        - Divides the contents of Rz from Ry, placing the result in Rx.The appropriate status bits in the S register are set.
    - MODU <Rx>, <Ry>, <Rz>

        - Gets the result of Ry modulo Rz, placing the result in Rx.The appropriate status bits in the S register are set.
- ***Type 6***
    - ADDF <Fx>, <Fy>, <Fz>
        - Adds the contents of Fy and Fz, placing the result in Fx. The appropriate status bits in the S register are set.
    - SUBF <Fx>, <Fy>, <Fz>
        - Subtracts the contents of Fz from Fy. The appropriate status bits in the S register are set.
    - MULF <Fx>, <Fy>, <Fz>
        - Multiplies the contents of Fy and Fz, placing the result in Fx. The appropriate status bits in the F register are set.
    - DIVF <Fx>, <Fy>, <Fz>
        - Divides the contents of Fz from Fy, placing the result in Fx.The appropriate status bits in the S register are set.

Memory Subsystem:
We want to explore the effects of different cache sizes and depths on our benchmarking programs. Thus, our architecture will support a configurable number of caches with configurable sizes. We plan to support a maximum number of 21 bits in address range for our main memory.

Project Plan:
The project will be developed using Rust, leveraging the egui GUI crate. Pranav will develop via VSCode on an M1 Mac, while Will will develop via Neovim on Ubuntu/ Windows (barring any unforeseen platform-specific issues). The entire project will be managed in a Github repository. Outstanding tasks, bugs, etc. will be managed via issues on the repo. Initial development of the project will focus on implementing the memory subsystem, CPU pipeline, and the GUI. Following this, various advanced features will be added to the GUI (as time permits), such as hot loading of program state.

As we are housemates, we will be meeting in person regularly to discuss the project. We plan to use the built-in unit testing that is offered by Rust's cargo utility to thoroughly test the functionality of our project.