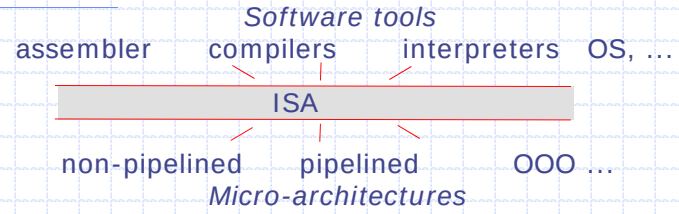
Constructive Computer Architecture:

RISC-V Instruction Set Architecture (ISA)

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

ISA: The software interface of programmable machine



- Instruction set architecture is a set of instructions
- Each instruction defines a way to transform the machine state
- ISA is a contract that an architect must follow while designing a machine a for a specific Performance, Power and Area (PPA) objective

RISC-V

- A new, open, free ISA from Berkeley
- Several variants
  - RV32, RV64, RV128 – Different data widths
  - 'I' – Base Integer instructions
  - 'M' – Multiply and Divide
  - 'A' – Atomic memory instructions
  - 'F' and 'D' – Single and Double precision floating point
  - 'V' – Vector extension
  - And many other modular extensions
- We will design an RV32I processor which is the base 32-bit variant

RV32I Register State

- 32 general purpose registers (GPR)
  - x0, x1, …, x31
  - 32-bit wide integer registers
  - x0 is hard-wired to zero
- Program counter (PC)
  - 32-bit wide
- CSR (Control and Status Registers)
  - User-mode
    - cycle (clock cycles) // read only
    - instret (instruction counts) // read only
  - Machine-mode
    - hartid (hardware thread ID) // read only
    - mepc, mcause etc. used for exception handling
  - Custom
    - mtohost (output to host) // write only – custom extension

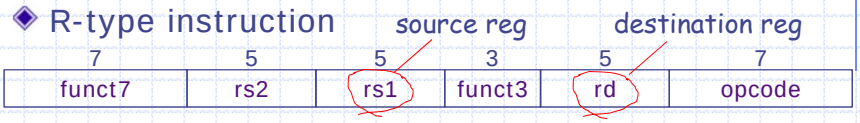
Memory

Instruction Types

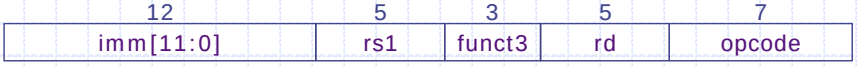
- Register-to-Register Arithmetic and Logical operations
- Control Instructions alter the sequential control flow
- Memory Instructions move data to and from memory
- CSR Instructions move data between CSRs and GPRs; the instructions often perform read-modify-write operations on CSRs
- Privileged Instructions are needed by the operating systems, and most cannot be executed by user programs

Instruction Formats

- R-type instruction



- I-type instruction & I-immediate (32 bits)



I-imm = signExtend(inst[31:20])

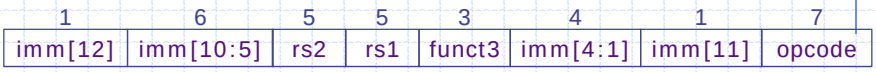
- S-type instruction & S-immediate (32 bits)

S-imm = signExtend({inst[31:25], inst[11:7]})

Immediate constants have strange encodings!

Instruction Formats *cont.*

- SB-type instruction & B-immediate (32 bits)

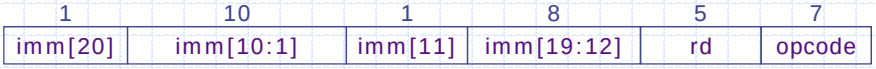


B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})

- U-type instruction & U-immediate (32 bits)

U-imm = signExtend({inst[31:12], 12'b0})

- UJ-type instruction & J-immediate (32 bits)



J-imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})

Computational Instructions

- Register-Register instructions (R-type)

  - opcode=OP: rd ← rs1 (funct3, funct7) rs2
  - funct3 = SLT/SLTU/AND/OR/XOR/SLL
  - funct3= ADD
    - funct7 = 0000000: rs1 + rs2
    - funct7 = 0100000: rs1 – rs2
  - funct3 = SRL
    - funct7 = 0000000: logical shift right
    - funct7 = 0100000: arithmetic shift right

Computational Instructions *cont*

- Register-immediate instructions (I-type)

  - opcode = OP-IMM: rd ← rs1 (funct3) I-imm
  - I-imm = signExtend(inst[31:20])
  - funct3 = ADDI/SLTI/SLTIU/ANDI/ORI/XORI

- A slight variant in coding for shift instructions - SLLI / SRLI / SRAI
  - rd ← rs1 (funct3, inst[30]) I-imm[4:0]

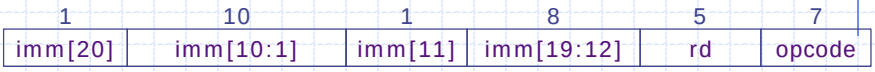
Computational Instructions *cont.*

- Register-immediate instructions (U-type)

  - opcode = LUI : rd ← U-imm
  - opcode = AUIPC : rd ← pc + U-imm
  - U-imm = {inst[31:12], 12'b0}

Control Instructions

- Unconditional jump and link (UJ-type)

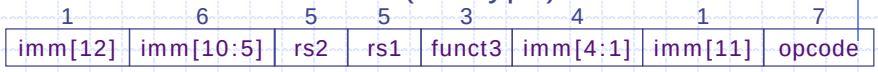


  - opcode = JAL: rd ← pc + 4; pc ← pc + J-imm
  - J-imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})
  - Jump ±1MB range
- Unconditional jump via register and link (I-type)

  - opcode = JALR: rd ← pc + 4; pc ← (rs1 + I-imm) & ~0x01
  - I-imm = signExtend(inst[31:20])

Control Instructions *cont.*

- Conditional branches (SB-type)



  - opcode = BRANCH: pc ← compare(funct3, rs1, rs2) ? pc + B-imm : pc + 4
  - B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})
  - Jump ±4KB range
  - funct3 = BEQ/BNE/BLT/BLTU/BGE/BGEU

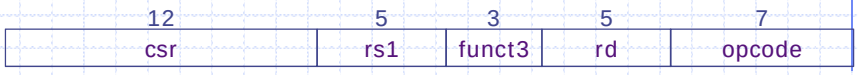
Load & Store Instructions

- Load (I-type)

  - opcode = LOAD: rd ← mem[rs1 + I-imm]
  - I-imm = signExtend(inst[31:20])
  - funct3 = LW/LB/LBU/LH/LHU
- Store (S-type)

  - opcode = STORE: mem[rs1 + S-imm] ← rs2
  - S-imm = signExtend({inst[31:25], inst[11:7]})
  - funct3 = SW/SB/SH

Instructions to Read and Write CSR



- opcode = SYSTEM
- CSRW rs1, csr (funct3 = CSRRW, rd = x0): csr ← rs1
- CSRR csr, rd (funct3 = CSRRS, rs1 = x0): rd ← csr

Assembly Code VS Binary

- Its too tedious to write programs in binary
- To simplify writing programs, assemblers provide:
  - mnemonics for instructions
    - add x1, x2, x3
  - pseudo instructions
    - mov x1, x2 // short for add x1, x2, x0
    - li x1, 6175 // short for lui x1, 2 ; addi x1, x1, -2017 (exact sequence depends on immediate value)
  - symbols for program locations and data
    - bnz x1, loop_begin
    - lw x1, flag
- Assemblers translate programs into machine code for the processor to execute

GCD in C

```
// require: x >= 0 && y > 0
int gcd(int a, int b) {
  int t;
  while(a != 0) {
    if(a >= b) {
      a = a - b;
    }
    else {
      t = a; a = b; b = t;
    }
  }
  return b;
}
```

GCD in RISC-V Assembler

```
// a: x1, b: x2, t: x3
begin:
  beqz x1, done // if(x1 == 0) goto done
  blt x1, x2, b_bigger // if(x1 < x2) goto b_bigger
  sub x1, x1, x2 // x1 := x1 - x2
  j begin // goto begin
b_bigger:
  mv x3, x1 // x3 := x1
  mv x1, x2 // x1 := x2
  mv x2, x3 // x2 := x3
  j begin // goto begin
done:  // now x2 contains the gcd
```


Application Binary Interface (ABI)

- Specifies rules for register usage in passing arguments and results for function calls
  - Callee-saved registers vs Caller-saved registers
- Assigns aliases for registers x1-x31
  - a0 to a7 – function argument registers (caller-saved)
  - a0 and a1 – function return value registers
  - s0 to s11 – Saved registers (callee-saved)
  - t0 to t6 – temporary registers (caller-saved)
  - ra – return address (caller-saved)
  - sp – stack pointer (callee-saved)
  - gp (global pointer), and tp (thread pointer) point to specific locations in memory used by the program for global and thread-local variables respectively

Calling GCD

Using the ABI

```
// assume we want to do the gcd of s0 and s1
// and put the result in s2
mov a0, s0          preparing arguments in a0, a1, etc.
mov a1, s1
addi sp, sp, -8 // saving registers in the stack
                // as needed by the caller
sw t0, 4(sp)        temporary
sw t1, 0(sp)
jal gcd // call gcd function
lw t0, 4(sp) // restoring caller-saved registers
lw t1, 0(sp)
addi sp, sp, 8
mov s2, a0
```


GCD in RISC-V Assembler

Using the ABI

```
// a: a0, b: a1, t: t0
gcd:      argument
  beqz a0, done
  blt a0, a1, b_bigger
  sub a0, a0, a1
  j gcd
b_bigger:  temporary
  mv t0, a0
  mv a0, a1
  mv a1, t0
  j gcd
done:  // now a1 contains the gcd
  mv a0, a1 // move to a0 for returning
  ret // jr ra
```

```
// a: x1, b: x2, t: x3
begin:
  beqz x1, done
  blt x1, x2, b_bigger
  sub x1, x1, x2
  j begin
b_bigger:
  mv x3, x1
  mv x1, x2
  mv x2, x3
  j begin
done:
```

ABI dictates that the result must come back in a0

Multiply

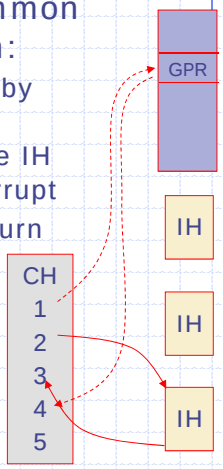
- mul x1, x2, x3 is an instruction in the 'M' extension (x1 := x2 * x3)
  - If 'M' is not implemented, this is an illegal instruction
- What happens if we run code from an RV32IM machine on an RV32I machine?
  - mul causes an illegal instruction exception
- An exception handler can take over and abort the program or emulate the instruction

Exception handling

- When an exception is caused
  - Hardware saves the information about the exception in CSRs:
    - mepc – exception PC
    - mcause – cause of the exception
    - mstatus.mpp – privilege mode of exception
  - Processor jumps to the address of the trap handler (stored in the mtvec CSR) and increases the privilege level
- An exception handler, a software program, takes over and performs the necessary action

Software for interrupt handling

- Hardware transfers control to the common software interrupt handler (CH) which:
  1. Saves all GPRs into the memory pointed by `mscratch`
  2. Passes `mcause`, `mepc`, stack pointer to the IH (a C function) to handle the specific interrupt
  3. On the return from the IH, writes the return value to `mepc`
  4. Loads all GPRs from the memory
  5. Execute ERET, which does:
     - set pc to `mepc`
     - pop `mstatus` (mode, enable) stack



Common Interrupt Handler- SW

```
common_handler: # entry point for exception handler
  # get the pointer to HW-thread local stack
  csrrw sp, mscratch, sp # swap sp and mscratch
  # save x1, x3 ~ x31 to stack (x2 is sp, save later)
  addi sp, sp, -128
  sw x1, 4(sp)
  sw x3, 12(sp)
  ...
  sw x31, 124(sp)
  # save original sp (now in mscratch) to stack
  csrr s0, mscratch # store mscratch to s0
  sw s0, 8(sp)
```

Common handler- SW *cont.*

Setting up and calling IH_Dispacher

```
common_handler:
  ... # we have saved all GPRs to stack
  # call C function to handle interrupt
  csrr a0, mcause # arg 0: cause
  csrr a1, mepc # arg 1: epc
  mv a2, sp # arg 2: sp – pointer to all saved GPRs
  jal ih_dispatcher # calls ih_dispatcher which may
                    # have been written in C
  # return value is the PC to resume
  csrw mepc, a0
  # restore mscratch and all GPRs
  addi s0, sp, 128; csrw mscratch, s0
  lw x1, 4(sp); lw x3, 12(sp); ...; lw x31, 124(sp)
  lw x2, 8(sp) # restore sp at last
  mret # finish handling interrupt
```


IH Dispatcher (in C)

```
long ih_dispatcher(long cause, long epc, long *regs) {
  // regs[i] refers to GPR xi stored in stack
  if(cause == 0x02)
    // illegal instruction
    return illegal_ih(cause, epc, regs);
  else if(cause == 0x08)
    // ecall (environment-call) instruction
    return syscall_ih(cause, epc, regs);
  else ... // other causes
}
```

SW emulation of MULT instruction

```
mul rd, rs1, rs2
```

- With proper exception handlers we can implement unsupported instructions in SW
- MUL returns the low 32-bit result of rs1*rs2 into rd
- MUL is decoded as an unsupported instruction and will throw an Illegal Instruction exception
- SW handles the exception in illegal_inst_ih() function
  - illegal_inst_ih() checks the opcode and function code of MUL to call the emulated multiply function
- Control is resumed to epc + 4 after emulation is done (ERET)

Illegal Instruction IH (in C)

```
long illegal_inst_ih(long cause, long epc, long *regs)
{   uint32_t inst = *((uint32_t*)epc); // fetch inst
    // check opcode & function codes
    if((inst & MASK_MUL) == MATCH_MUL) {
      // is MUL, extract rd, rs1, rs2 fields
      int rd = (inst >> 7) & 0x01F;
      int rs1 = ...; int rs2 = ...;
      // emulate regs[rd] = regs[rs1] * regs[rs2]
      emulate_multiply(rd, rs1, rs2, regs);
      return epc + 4; // done, resume at epc+4
    } else abort();
}
```