# Intro to Computer Arch

CARP Research Presentation 1

# First: Languages

# What about Hardware?

# Hardware Description Languages (HDL)

Common HDLs

- VHDL
- Verilog / **SystemVerilog**

Used to describe hardware - like a **CPU**

- Compiles to a “netlist” rather than assembly

Used in testing or for deployment on FPGAs

# What does a computer (CPU) do?

# CPU

Executes instructions

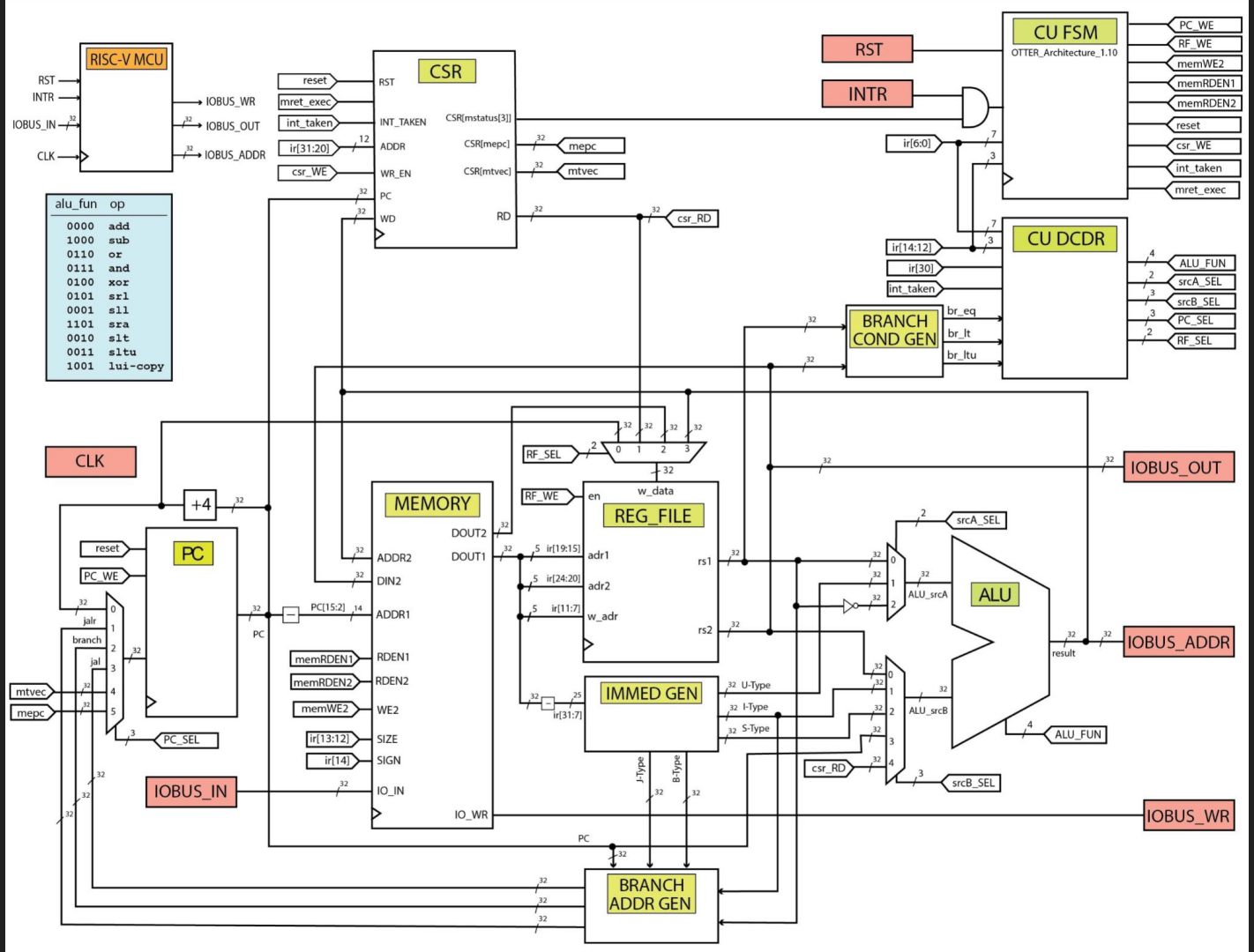
Needs a common language to work on

- Compilers
- Assembly

Different modules, execution blocks, for different purposes

Forms of speeding up instruction execution (see next presentation 👀)

# How do the instructions execute?


RISC-V MCU
RST
INTR
IOBUS_IN
CLK
IOBUS_WR
IOBUS_OUT
IOBUS_ADDR
CSR
reset
mret_exec
int_taken
ir[31:20]
csr_WE
INT_TAKEN
ADDR
WR_EN
PC
WD
CSR[mstatus[3]]
CSR[mepc]
CSR[mtvec]
RD
mepc
mtvec
csr_RD
RST
INTR
CU FSM
OTTER_Architecture_1.10
PC_WE
RF_WE
memWE2
memRDEN1
memRDEN2
reset
csr_WE
int_taken
mret_exec
ir[6:0]
CU DCDR
ir[14:12]
ir[30]
int_taken
ALU_FUN
srcA_SEL
srcB_SEL
PC_SEL
RF_SEL
BRANCH COND GEN
br_eq
br_lt
br_ltu
alu_fun op
0000 add
1000 sub
0110 or
0111 and
0100 xor
0101 srl
0001 sll
1101 sra
0010 slt
0011 sltu
1001 lui-copy
CLK
+4
PC
reset
PC_WE
jalr
branch
jal
mtvec
mepc
PC_SEL
MEMORY
ADDR2
DIN2
ADDR1
DOUT2
DOUT1
RDEN1
RDEN2
WE2
SIZE
SIGN
IO_IN
IO_WR
memRDEN1
memRDEN2
memWE2
ir[13:12]
ir[14]
IOBUS_IN
RF_SEL
RF_WE
REG_FILE
w_data
en
adr1
adr2
w_adr
rs1
rs2
ir[19:15]
ir[24:20]
ir[11:7]
IMMED GEN
ir[31:7]
U-Type
I-Type
S-Type
J-Type
B-Type
ALU
ALU_srcA
ALU_srcB
result
srcA_SEL
srcB_SEL
ALU_FUN
csr_RD
IOBUS_OUT
IOBUS_ADDR
IOBUS_WR
BRANCH ADDR GEN
PC

## RISC-V OTTER Assembly Instruction Overview

Table 12 lists RISC-V OTTER instructions; shaded instructions are pseudoinstructions

| Instruction | Description | RTL | Comment |
|---|---|---|---|
| `add rd,rs1,rs2` | addition | X[rd] ← X[rs1] + X[rs2] | |
| `addi rd,rs1,imm` | addition with immediate | X[rd] ← X[rs1] + sext(imm) | |
| `and rd,rs1,rs2` | bitwise AND | X[rd] ← X[rs1] · X[rs1] | |
| `andi rd,rs1,imm` | Bitwise AND immediate | X[rd] ← X[rs1] · sext(imm) | |
| `auipc rd,imm` | add upper immediate to PC | X[rd] ← PC + (sext(imm)<<12) | |
| `beq rs1,rs2,imm` | branch if equal | PC ← PC + sext(imm) if (X[rs1] == X[rs2]) | imm ≠ value |
| `beqz rs1,imm` | branch if equal to zero | PC ← PC + sext(imm) if (X[rs1] == 0) | imm ≠ value |
| `bge rs1,rs2,imm` | branch if greater than or equal | PC ← PC + sext(imm) if (X[rs1] $\geq_s$ X[rs2]) | imm ≠ value |
| `bgeu rs1,rs2,imm` | branch if greater than or equal unsigned | PC ← PC + sext(imm) if (X[rs1] $\geq_u$ X[rs2]) | imm ≠ value |
| `bgez rs1,imm` | branch if greater than or equal to zero | PC ← PC + sext(imm) if (X[rs1] $\geq_s$ 0) | imm ≠ value |
| `bgt rs1,rs2,imm` | branch if greater than | PC ← PC + sext(imm) if (X[rs1] $>_s$ X[rs2]) | imm ≠ value |
| `bgtu rs1,rs2,imm` | branch if greater than unsigned | PC ← PC + sext(imm) if (X[rs1] $>_u$ X[rs2]) | imm ≠ value |
| `bgtz rs1,rs2,imm` | branch if greater than zero | PC ← PC + sext(imm) if (X[rs1] $>_s$ 0) | imm ≠ value |
| `ble rs1,rs2,imm` | branch if less than or equal | PC ← PC + sext(imm) if (X[rs1] $\leq_s$ X[rs2]) | imm ≠ value |
| `bleu rs1,rs2,imm` | branch if less than or equal (unsigned) | PC ← PC + sext(imm) if (X[rs1] $\leq_u$ X[rs2]) | imm ≠ value |
| `blez rs1,rs2,imm` | branch if less than or equal zero | PC ← PC + sext(imm) if (X[rs1] $\leq_s$ 0) | imm ≠ value |
| `blt rs1,rs2,imm` | branch if less than | PC ← PC + sext(imm) if (X[rs1] $<_s$ X[rs2]) | imm ≠ value |
| `bltz rs1,imm` | branch if less than zero | PC ← PC + sext(imm) if (X[rs1] $<_s$ 0) | imm ≠ value |
| `bltu rs1,rs2,imm` | branch if less than (unsigned) | PC ← PC + sext(imm) if (X[rs1] $<_u$ X[rs2]) | imm ≠ value |
| `bne rs1,rs2,imm` | branch if not equal | PC ← PC + sext(imm) if (X[rs1] ≠ X[rs2]) | imm ≠ value |
| `bnez rs1,imm` | branch if not equal to zero | PC ← PC + sext(imm) if (X[rs1] ≠ 0) | imm ≠ value |
| `call label` | branch to subroutine<br>WARNING: overwrites X6 | X[rd] ← PC + 8; PC ← &symbol<br>(rd=X1 if rd omitted) | imm ≠ value<br>overwrites X6 |
| `csrrc rd,csr,rs1` | control & status reg read and bit clear | X[rd] ← CSR[csr]; CSR[csr] ← CSR[csr] & ~rs1 | clears part of reg |
| `csrrs rd,csr,rs1` | control & status reg read and bit set | X[rd] ← CSR[csr]; CSR[csr] ← CSR[csr] \| rs1 | sets part of reg |
| `csrrw rd,csr,rs1` | control & status register read & write | X[rd] ← CSR[csr]; CSR[csr] ← rs1 | writes entire reg |
| `csrw rs1,csr` | control & status register write | CSR[csr] ← rs1 | |
| `j imm` | unconditional branch | PC ← PC + sext(imm) | imm ≠ value |
| `jal rd,imm`<br>`jal imm` | unconditional branch with offset | X[rd] ← PC + 4; PC ← PC + sext(imm) | imm ≠ value<br>rd=X1 if rd omitd |
| `jalr rd,rs1,imm`<br>`jalr rs1`<br>`jalr rs1,imm` | unconditional branch with offset & link | X[rd] ← PC+4; PC ← (X[rs1] + sext(imm)) & ~1 | imm ≠ value<br>rd=X1 if rd omitd |
| `jr rs1` | unconditional branch to register address | PC ← X[rs1] | |
| `la rd,symbol` | load absolute address of symbol | X[rd] ← &symbol | |
| `lb rd,imm(rs1)` | load byte | X[rd] ← sext( M[X[rs1] + sext(imm) ] [7:0] ) | |
| `lbu rd,imm(rs1)` | load byte unsigned | X[rd] ← zext( M[X[rs1] + sext(imm)] [7:0] ) | |
| `lh rd,imm(rs1)` | load halfword | X[rd] ← sext( M[X[rs1] + sext(imm) ] [15:0] ) | |
| `lhu rd,imm(rs1)` | load halfword unsigned | X[rd] ← zext( M[ X[rs1] + sext(imm) ] [15:0] ) | |
| `li rd,imm` | load immediate | X[rd] ← imm | |
| `lw rd,imm(rs1)` | load word into register | X[rd] ← M[X[rs1] + sext(imm) ] [31:0] | |
| `lui rd,imm` | load upper immediate | X[rd] ← imm << 12 | |
| `mret` | machine mode exception return | PC ← CSR[mepc]; CSR[mstatus(mie) ← mstatus(mpie) | |
| `mv rd,rs1` | move | X[rd] ← X[rs1] | |
| `neg rd,rs2` | negate | X[rd] ← -X[rs2] | |
| `nop` | no operation | nada (PC ← PC + 4) | |
| `not rd,rs2` | ones complement | X[rd] ← ~X[rs2] | |
| `or rd,rs1,rs2` | bitwise inclusive OR | X[rd] ← X[rs1] \| X[rs2] | |
| `ori rd,rs1,imm` | bitwise inclusive OR immediate | X[rd] ← X[rs1] \| sext(imm) | |
| `ret` | return from subroutine | PC ← X1 | |
| `sb rs2,imm(rs1)` | store byte in memory | M[ X[rs1] + sext(imm) ] ← X[rs2][7:0] | |
| `seqz rd,rs1` | set if equal to zero | X[rd] ← ( X[rs1] == 0 ) ? 1 : 0 | |
| `sgtz rd,rs2` | set if greater than zero | X[rd] ← ( X[rs2] $>_s$ 0 ) ? 1 : 0 | |
| `sh rs2,imm(rs1)` | store halfword in memory | M[ X[rs1] + sext(imm) ] ← X[rs2][15:0] | |
| `sw rs2,imm(rs1)` | store word | M[ X[rs1] + sext(imm) ] ← X[rs2] | |
| `sll rd,rs1,rs2` | logical shift left | X[rd] ← X[rs1] << X[rs2][4:0] | |
| `slli rd,rs1,imm` | logical shift left immediate | X[rd] ← X[rs1] << imm[4:0] | |
| `slt rd,rs1,rs2` | set if less than | X[rd] ← ( X[rs1] $<_s$ X[rs2] ) ? 1 : 0 | |
| `slti rd,rs1,imm` | set if less than immediate | X[rd] ← ( X[rs1] $<_s$ sext(imm) ) ? 1 : 0 | |
| `sltiu rd,rs1,imm` | set if less than immediate unsigned | X[rd] ← ( X[rs1] $<_u$ sext(imm) ) ? 1 : 0 | |
| `sltu rd,rs1,rs2` | set if less than unsigned | X[rd] ← ( X[rs1] $<_u$ X[rs2] ) ? 1 : 0 | |
| `sltz rd,rs1` | set if less than zero | X[rd] ← ( X[rs1] $<_s$ 0 ) ? 1 : 0 | |
| `snez rd,rs2` | set if not equal to zero | X[rd] ← ( X[rs2] ≠ 0 ) ? 1 : 0 | |
| `sra rd,rs1,rs2` | arithmetic shift right | X[rd] ← X[rs1] $>>_s$ X[rs2] [4:0] | |
| `srai rd,rs1,imm` | arithmetic shift right immediate | X[rd] ← X[rs1] $>>_s$ imm[4:0] | |
| `srl rd,rs1,rs2` | logical shift right | X[rd] ← X[rs1] >> X[rs2][4:0] | |
| `srli rd,rs1,imm` | logical shift right immediate | X[rd] ← X[rs1] >> imm[4:0] | |
| `sub rd,rs1,rs2` | subtract | X[rd] ← X[rs1] – X[rs2] | |
| `xor rd,rs1,rs2` | exclusive OR | X[rd] ← X[rs1] ^ X[rs2] | |
| `xori rd,rs1,imm` | exclusive OR immediate | X[rd] ← X[rs1] ^ sext(imm) | |

**Table 13: RISC-V OTTER Instructions with RTL description.**

# Thanks

Stick around CARP for more presentations in the future:

- Advanced CPU Architecture (CPE333 and Beyond)
- CPU vs. GPU Architecture
- Introduction to Research