

Constructive Computer Architecture:

Non-Pipelined Processors

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

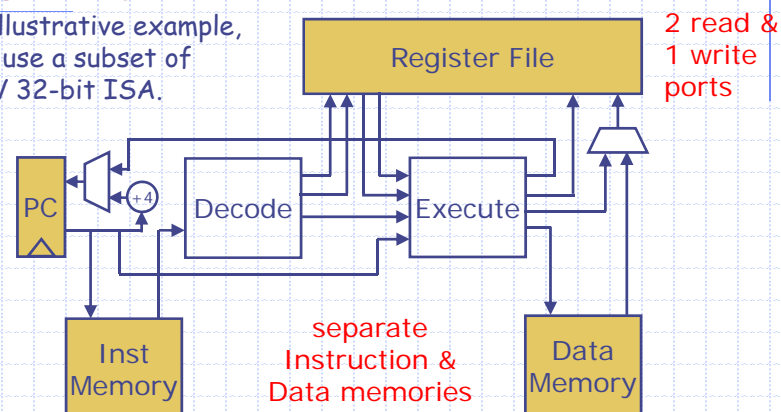
October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-1

Single-Cycle RISC Processor

As an illustrative example,
we will use a subset of
RISC-V 32-bit ISA.



Datapath and control are derived automatically
from a high-level rule-based description

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-2

Single-Cycle Implementation

code structure

```
module mkProc(Proc);
  Reg#(Addr) pc <- mkRegU;
  RFile      rf <- mkRFile;
  IMemory    iMem <- mkIMemory;
  DMemory    dMem <- mkDMemory;

  rule doProc;
    let inst = iMem.reg(pc);
    let dInst = decode(inst);
    let rVal1 = rf.rd1(dInst.rSrc1);
    let rVal2 = rf.rd2(dInst.rSrc2);
    let eInst = exec(dInst, rVal1, rVal2, pc);
    update rf, pc and dMem
```

to be explained later

instantiate the state

extracts fields needed for execution

produces values needed to update the processor state

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-3

RISC-V Register States

- ◆ 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)
 - cycle
 - instret
 - mhartid
 - mtohost
 - ...
- will be implemented in labs as needed

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-4

Computational Instructions

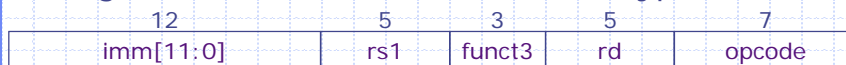
◆ Register-Register instructions (R-type)



- opcode=OP: $rd \leftarrow rs1$ (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)



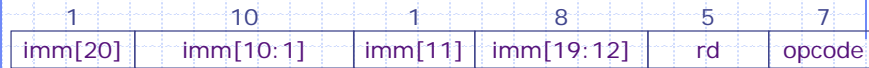
- opcode = OP-IMM: $rd \leftarrow rs1$ (funct3) I-imm
- I-imm = signExtend(inst[31:20])
- funct3 = ADDI/SLTI/SLTIU/ANDI/ORI/XORI

◆ A slight variant in coding for shift instructions - SLLI / SRLI / SRAI

- $rd \leftarrow rs1$ (funct3, inst[30]) I-imm[4:0]

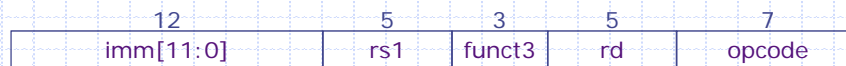
Control Instructions

◆ Unconditional jump and link (UJ-type)



- opcode = JAL: $rd \leftarrow pc + 4$; $pc \leftarrow pc + J-imm$
- J-imm = $signExtend(\{inst[31], inst[19:12], inst[20], inst[30:21], 1'b0\})$
- Jump $\pm 1MB$ range

◆ Unconditional jump via register and link (I-type)



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

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-7

Control Instructions *cont.*

◆ Conditional branches (SB-type)



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

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-8

Load & Store Instructions

◆ Load (I-type)



- opcode = LOAD: $rd \leftarrow mem[rs1 + I-imm]$
- I-imm = $signExtend(inst[31:20])$
- funct3 = LW/LB/LBU/LH/LHU

◆ Store (S-type)



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

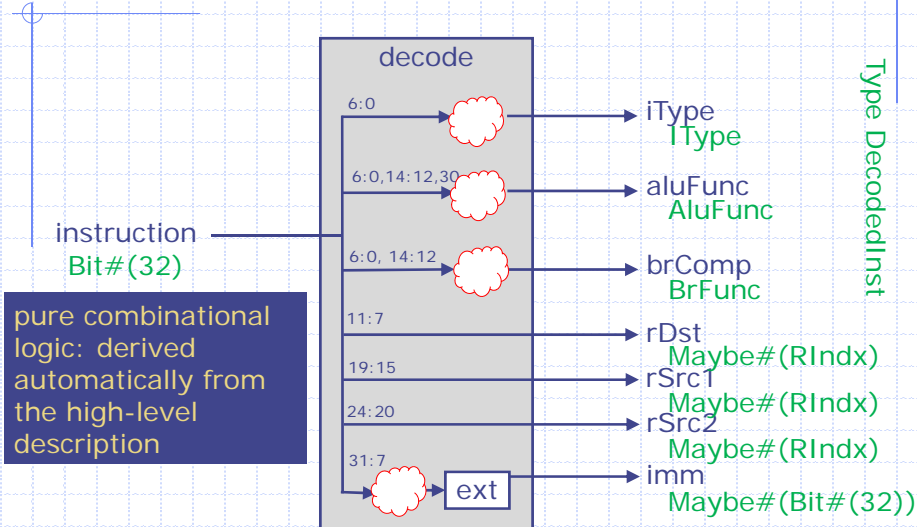
October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-9

Decoding Instructions:

extract fields needed for execution



October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-10

Decoded Instruction Type

```

typedef struct {
    IType      iType;
    AluFunc    aluFunc;
    BrFunc     brFunc;
    Maybe#(RIndx) dst;
    Maybe#(RIndx) src1;
    Maybe#(RIndx) src2;
    Maybe#(Data) imm;
} DecodedInst deriving(Bits, Eq);

typedef enum {Unsupported, Alu, Ld, St, J, Jr, Br, Auipc}
IType deriving(Bits, Eq);
typedef enum {Add, Sub, And, Or, Xor, Slt, Sltu, Sll,
Sra, Srl} AluFunc deriving(Bits, Eq);
typedef enum {Eq, Neq, Lt, Ltu, Ge, Geu, AT, NT} BrFunc
deriving(Bits, Eq);

```

Destination register 0 behaves like an Invalid destination

Instruction groups with similar executions paths

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-11

Internal names for various opcode and funct3 patterns

```

// opcode
Bit#(7) opOpImm = 7'b0010011; // OP-IMM
Bit#(7) opOp    = 7'b0110011; // OP
Bit#(7) opLui   = 7'b0110111; // LUI
Bit#(7) opAuipc = 7'b0010111; // AUIPC
Bit#(7) opJal   = 7'b1101111; // JAL
Bit#(7) opJalr  = 7'b1100111; // JALR
Bit#(7) opBranch = 7'b1100011; // BRANCH
Bit#(7) opLoad  = 7'b0000011; // LOAD
Bit#(7) opStore = 7'b0100011; // STORE
// funct3
Bit#(3) fnADD    = 3'b000; // ADD
Bit#(3) fnSLL    = 3'b001; // SLL
Bit#(3) fnSLT    = 3'b010; // SLT .....

```

Values are specified in the RISC-V ISA

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-12

Decode Function

```
function DecodedInst decode(Bit#(32) inst);
  DecodedInst dInst = ?;
  let opcode = inst[ 6 : 0 ];
  let rd      = inst[ 11 : 7 ];
  let funct3  = inst[ 14 : 12 ];
  let rs1     = inst[ 19 : 15 ];
  let rs2     = inst[ 24 : 20 ];
  let aluSel  = inst[ 30 ]; // Add/Sub, Srl/Sra
  Bit#(32)immI=...; Bit#(32)immS=...; Bit#(32)immB=...;
  Bit#(32)immU=...; Bit#(32)immJ=...; // I/S/B/U/J-imm
  case (opcode)
    opOp
    ...
  endcase
  return dInst;
endfunction
```

initially undefined

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-13

Decoding Instructions: Computational Instructions

```
opOp: begin
  dInst.iType = Alu;
  dInst.aluFunc = case (funct3)
    fnAND: And;
    fnSLTU: Sltu;
    ...
    fnADD: Add;
    fnSR: aluSel == 0 ? Srl : Sra;
  endcase;
  dInst.brFunc = NT;
  dInst.dst = Valid rd;
  dInst.src1 = Valid rs1;
  dInst.src2 = Invalid;
  dInst.imm = Valid ImmI;
end
```

Decoding instructions with immediate operand (i.e. opcode = OP-IMM) is similar

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-14

Decoding Instructions: Conditional Branch

```
opBranch: begin
  Maybe#(BrFunc) brF = case(func3)
    fnBEQ: Valid Eq;
    ...
    fnBGEU: Valid Geu;
  default: Invalid;
endcase;
dInst.iType = isValid(brF) ? Br : Unsupported;
dInst.aluFunc = ?;
dInst.brFunc = fromMaybe(?, brF);
dInst.dst = Invalid;
dInst.src1 = Valid rs1;
dInst.src2 = Valid rs2;
dInst.imm = Valid immB;
end
```

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-15

Decoding Instructions: Load & Store

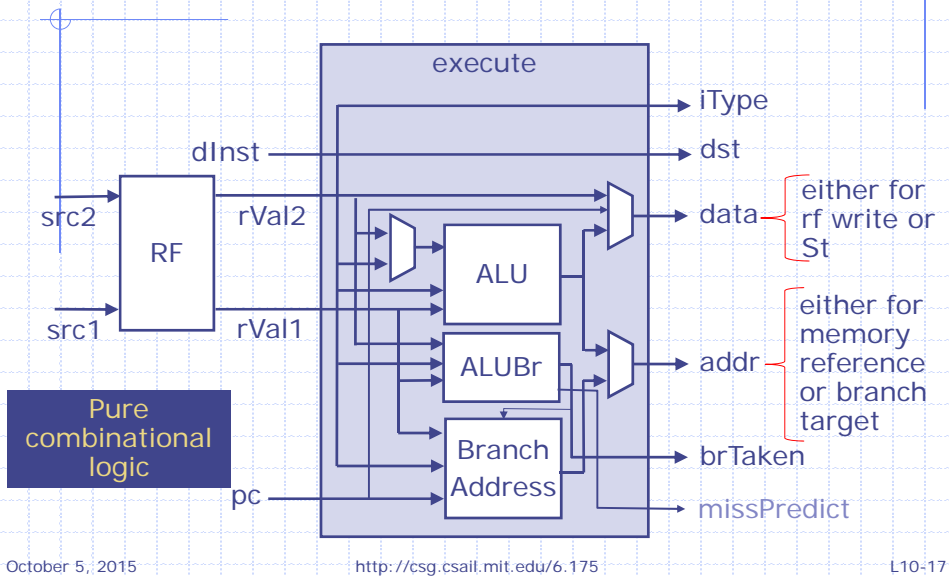
```
opLoad: begin // only support LW
  dInst.iType = funct3 == fnLW ? Ld : Unsupported;
  dInst.aluFunc = Add; // calc effective addr
  dInst.brFunc = NT;
  dInst.dst = Valid rd;
  dInst.src1 = Valid rs1;
  dInst.src2 = Invalid;
  dInst.imm = Valid immI;
end
opStore: begin // only support SW
  dInst.iType = funct3 == fnSW ? St : Unsupported;
  dInst.aluFunc = Add; // calc effective addr
  dInst.brFunc = NT;
  dInst.dst = Invalid;
  dInst.src1 = Valid rs1;
  dInst.src2 = Valid rs2;
  dInst.imm = Valid immS;
end
```

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-16

Reading Registers and Executing Instructions



Output of exec function

```
typedef struct {
    IType      iType;
    Maybe#(RIndx) dst;
    Data      data;
    Addr      addr;
    Bool      mispredict;
    Bool      brTaken;
} ExecInst deriving (Bits, Eq);
```

Execute Function

```
function ExecInst exec(DecodedInst dInst, Data rVal1,
                      Data rVal2, Addr pc);
  ExecInst eInst = ?;
  Data aluVal2 = fromMaybe(rVal2, dInst.imm);

  let aluRes = alu(rVal1, aluVal2, dInst.aluFunc);
  eInst.iType = dInst.iType;
  eInst.data = dInst.iType==St ? rVal2 :
              (dInst.iType==J || dInst.iType==Jr) ?
              (pc+4) : dInst.iType==Auipc ?
              (pc+fromMaybe(?, dInst.imm)) : aluRes;

  let brTaken = aluBr(rVal1, rVal2, dInst.brFunc);
  let brAddr = brAddrCalc(pc, rVal1, dInst.iType,
                          fromMaybe(?, dInst.imm), brTaken);

  eInst.brTaken = brTaken;
  eInst.addr = (dInst.iType==Ld || dInst.iType==St)?
              aluRes : brAddr;
  eInst.dst = dInst.dst;
  return eInst;
endfunction
```

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-19

Single-Cycle SMIPS *atomic state updates*

```
if(eInst.iType == Ld)
  eInst.data <- dMem.req(MemReq{op: Ld,
                          addr: eInst.addr, data: ?});
else if (eInst.iType == St)
  let dummy <- dMem.req(MemReq{op: St,
                          addr: eInst.addr, data: data});

if(isValid(eInst.dst))
  rf.wr(fromMaybe(?, eInst.dst), eInst.data);

pc <= eInst.brTaken ? eInst.addr : pc + 4;
```

```
endrule
endmodule
```

state updates

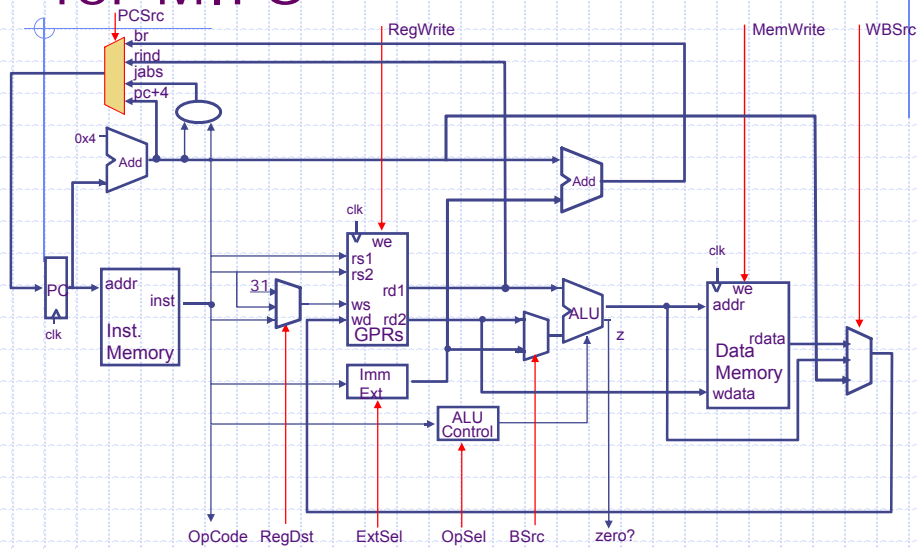
The whole processor is described using one rule;
lots of big combinational functions

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-20

Harvard-Style Datapath for MIPS old way



October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-21

Hardwired Control Table old way

Opcode	ExtSel	BSrc	OpSel	MemW	RegW	WBSrc	RegDst	PCSrc
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	sExt ₁₆	Imm	Op	no	yes	ALU	rt	pc+4
ALUi _u	uExt ₁₆	Imm	Op	no	yes	ALU	rt	pc+4
LW	sExt ₁₆	Imm	+	no	yes	Mem	rt	pc+4
SW	sExt ₁₆	Imm	+	yes	no	*	*	pc+4
BEQZ _{z=0}	sExt ₁₆	*	0?	no	no	*	*	br
BEQZ _{z=1}	sExt ₁₆	*	0?	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	R31	jabs
JR	*	*	*	no	no	*	*	rind
JALR	*	*	*	no	yes	PC	R31	rind

BSrc = Reg / Imm
RegDst = rt / rd / R31

WBSrc = ALU / Mem / PC
PCSrc = pc+4 / br / rind / jabs

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-22

Extras

Instruction Formats

◆ R-type instruction



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



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

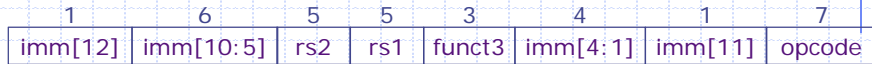
◆ S-type instruction & S-immediate (32 bits)



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

Instruction Formats *cont.*

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



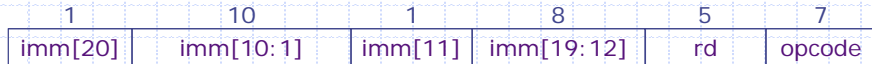
B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})

◆ 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})

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-25

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}

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-26

Decoding Instructions

```
case (opcode)
  opOpImm: ...
  opOp: ...
  opLui: ...
  opAuipc: ...
  opJal: ...
  opJalr: ...
  opBranch: ...
  opLoad: ...
  opStore: ...
  default: ... // Unsupported
endcase;
```

// opcode	
Bit#(7) opOpImm	= 7'b0010011;
Bit#(7) opOp	= 7'b0110011;
Bit#(7) opLui	= 7'b0110111;
Bit#(7) opAuipc	= 7'b0010111;
Bit#(7) opJal	= 7'b1101111;
Bit#(7) opJalr	= 7'b1100111;
Bit#(7) opBranch	= 7'b1100011;
Bit#(7) opLoad	= 7'b0000011;
Bit#(7) opStore	= 7'b0100011;

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-27

Decoding Instructions: Computational Instructions - Imm

```
opOpImm: begin
  dInst.iType = Alu;
  dInst.aluFunc = case (funct3)
    fnADD: Add;
    fnSLTU: Sltu;
    ...
    fnSLL: Sll;
    fnSR: aluSel == 0 ? Srl : Sra;
  endcase;
  dInst.brFunc = NT;
  dInst.dst = Valid rd;
  dInst.src1 = Valid rs1;
  dInst.src2 = Invalid;
  dInst.imm = Valid immI;
end
```

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-28

Decoding Instructions: Computational Instructions *cont.*

```
opLui: begin // rd = immU + r0
    dInst.iType = Alu;
    dInst.aluFunc = Add;
    dInst.brFunc = NT;
    dInst.dst = tagged Valid rd;
    dInst.src1 = tagged Valid 0;
    dInst.src2 = tagged Invalid;
    dInst.imm = tagged Valid immU;
end
opAuipc: begin
    dInst.iType = Auipc;
    dInst.aluFunc = ?;
    dInst.brFunc = NT;
    dInst.dst = tagged Valid rd;
    dInst.src1 = tagged Invalid;
    dInst.src2 = tagged Invalid;
    dInst.imm = tagged Valid immU;
end
```

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-29

Decoding Instructions: Unconditional Jumps

```
opJal: begin
    dInst.iType = J;
    dInst.aluFunc = ?;
    dInst.brFunc = AT;
    dInst.dst = Valid rd;
    dInst.src1 = Invalid;
    dInst.src2 = Invalid;
    dInst.imm = Valid immJ;
end
opJalr: begin
    dInst.iType = Jr;
    dInst.aluFunc = ?;
    dInst.brFunc = AT;
    dInst.dst = Valid rd;
    dInst.src1 = Valid rs1;
    dInst.src2 = Invalid;
    dInst.imm = Valid immI;
end
```

October 5, 2015

<http://csg.csail.mit.edu/6.175>

L10-30

Decoding instructions: Unsupported

```
default: begin
    dInst.iType = Unsupported;
    dInst.aluFunc = ?;
    dInst.brFunc = NT;
    dInst.dst = Invalid;
    dInst.src1 = Invalid;
    dInst.src2 = Invalid;
    dInst.imm = Invalid;
end
```

Branch Address Calculation

```
function Addr brAddrCalc(Addr pc, Data val,
                        IType iType, Data imm, Bool taken);
    Addr pcPlus4 = pc + 4;
    Addr targetAddr = case (iType)
        J : {pc + imm};
        Jr : {truncateLSB(val + imm), 1'b0};
        Br : (taken ? pc + imm : pcPlus4);
    default: pcPlus4;
    endcase;
    return targetAddr;
endfunction
```