Architecture (Sequential Implementation)

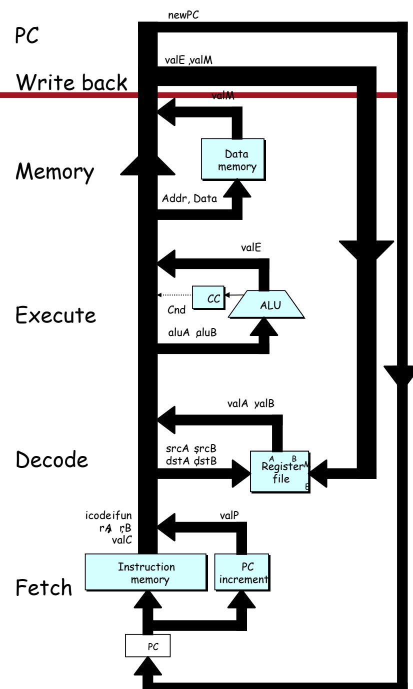
CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

Data Path

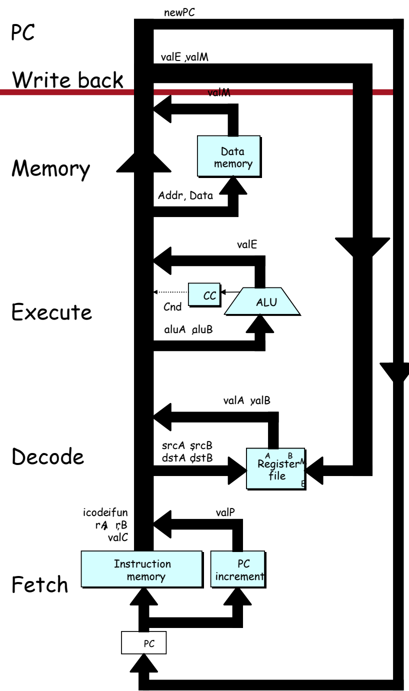
- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter



With Slides from Bryant and O'Hallaron

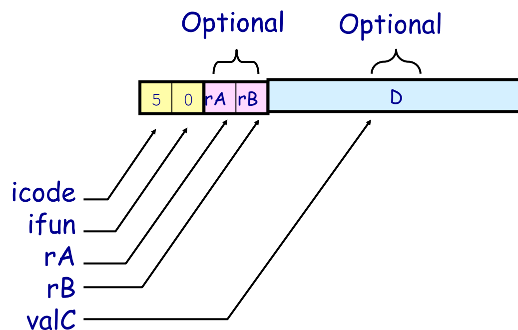
Stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



With Slides from Bryant and O'Hallaron

Instruction Decoding



- Instruction Format
  - Instruction byte — icode:ifun
  - Optional register byte — rA:rB
  - Optional constant word — valC

With Slides from Bryant and O'Hallaron

Executing Arith./Logical Operation

OPl rA, rB | 6 | fn | rA | rB

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

Stage Computation: Arith/Log. Ops

	OPl rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing `rmmovl`

- •Fetch
  - Read 6 bytes
- •Decode
  - Read operand registers
- •Execute
  - Compute effective address
- •Memory
  - Write to memory
- •Write back
  - Do nothing
- •PC Update
  - Increment PC by 6

Stage Computation: `rmmovl`

	rmmovl rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_4[PC+2]	Read displacement D
	valP ← PC+6	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_4[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- Use ALU for address computation

Executing `popl`

popl rA	b	0	rA	8

- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2

With Slides from Bryant and O'Hallaron

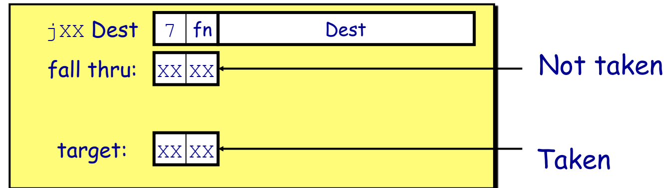
Stage Computation: `popl`

	popl rA	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valP ← PC+2	Compute next PC
Decode	valA ← R[%esp]	Read stack pointer
	valB ← R [%esp]	Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read from stack
Write back	R[%esp] ← valE	Update stack pointer
	R[rA] ← valM	Write back result
PC update	PC ← valP	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

With Slides from Bryant and O'Hallaron

Executing Jumps



- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

With Slides from Bryant and O'Hallaron

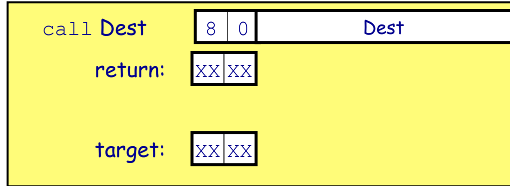
Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	Read instruction byte Read destination address Fall through address
Decode		
Execute	Cnd ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Cnd ? valC : valP	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

With Slides from Bryant and O'Hallaron

Executing `call`



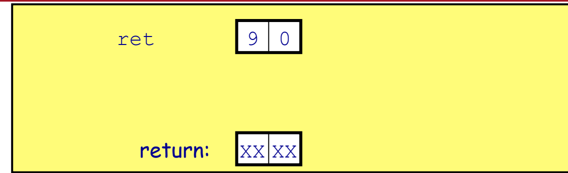
- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

Stage Computation: `call`

	`call Dest`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Compute return point
Decode	valB ← R[`%esp`]	Read stack pointer
Execute	valE ← valB + -4	Decrement stack pointer
Memory	M_4[valE] ← valP	Write return value on stack
Write back	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Executing `ret`



- Fetch
  - Read 1 byte
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read return address from old stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to return address

Stage Computation: `ret`

	ret	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[%esp] valB ← R[%esp]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read return address
Write back	R[%esp] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

Computation Steps

		OPl rA, rB	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	Read register byte
	valC		[Read constant word]
	valP	valP ← PC+2	Compute next PC
Decode	valA, srcA	valA ← R[rA]	Read operand A
	valB, srcB	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set condition code register
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

		call Dest	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_4[PC+1]	Read constant word
	valP	valP ← PC+5	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[%esp]	Read operand B
Execute	valE	valE ← valB + –4	Perform ALU operation
	Cond code		[Set condition code reg.]
Memory	valM	M_4[valE] ← valP	[Memory read/write]
Write back	dstE	R[%esp] ← valE	[Write back ALU result]
	dstM		Write back memory result
PC update	PC	PC ← valC	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

3 Simulators

- The most widely used methodology in computer architecture research
- Using a simulator to mimic the functionality and performance metrics of a target computer
- Why simulator?
  - The system that we are studying is not yet built
  - Observe internal state that is not observable
- Different levels of abstraction
  - Functional
  - Micro-architectural
  - Gate-level
  - Circuit level

--- We will use three Y86 simulators in this class

misc/yis: The ISA simulator

In misc/yis.c:

```
for (step = 0; step < max_steps && e == STAT_AOK; step++)
        e = step_state(s, stdout);
```

Output state change

In misc/isa.c

```
stat_t step_state(state_ptr s, FILE *error_file)
{
   ftpc++;
   hi0 = HI4(byte0);
   lo0 = LO4(byte0);
   …
}
```

misc/yis: The ISA simulator

In misc/isa.c

```
stat_t step_state(state_ptr s, FILE *error_file)
{
  …
  if (need_regids) {
      ok1 = get_byte_val(s->m, ftpc, &byte1);
      ftpc++;
      hi1 = HI4(byte1);
      lo1 = LO4(byte1);
  }
  if (need_imm) {
      okc = get_word_val(s->m, ftpc, &cval);
      ftpc += 4;
  }
    …
}
```

With Slides from Bryant and O'Hallaron

misc/yis: The ISA simulator

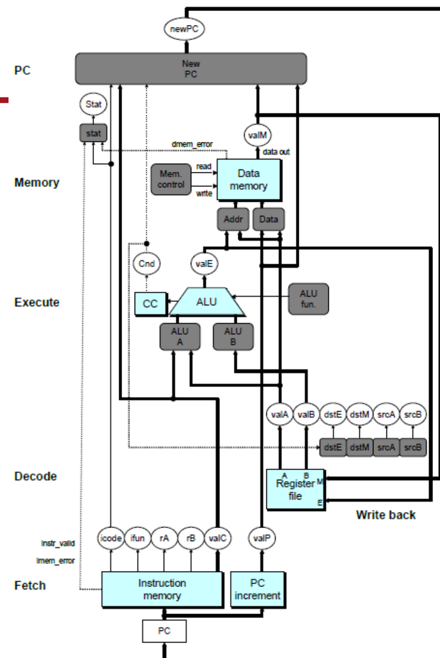
In misc/isa.c

```
stat_t step_state(state_ptr s, FILE *error_file)
{
    …
 switch (hi0) {
   case I_NOP:
       s->pc = ftpc;
       break;
   case I_HALT:
       …
   case I_RRMOVL:
      …
 }
```

With Slides from Bryant and O'Hallaron

SEQ Datapath

- Key
  - Blue boxes: predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes: control logic
    - Describe in HCL
  - White ovals: labels for signals
  - Thick lines: 32-bit word values
  - Thin lines: 4-8 bit values
  - Dotted lines: 1-bit values



Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify

Data Types

- `bool`: Boolean
  - `a, b, c, …`
- `int`: words
  - `A, B, C, …`
  - Does not specify word size---bytes, 32-bit words, …

Statements

- `bool a = bool-expr ;`
- `int A = int-expr ;`

HCL Operations

- Classify by type of value returned

Boolean Expressions

- Logic Operations
  - `a && b, a || b, !a`
- Word Comparisons
  - `A == B, A != B, A < B, A <= B, A >= B, A > B`
- Set Membership
  - `A in { B, C, D }`
    - Same as `A == B || A == C || A == D`

Word Expressions

- Case expressions
  - `[ a : A; b : B; c : C ]`
  - Evaluate test expressions `a`, `b`, `c`, … in sequence
  - Return word expression `A`, `B`, `C`, … for first successful test

seq/ssim: The Sequential Simulator

Compilation process:

- ../misc/hcl2c -n seq-std.hcl <seq-std.hcl >seq-std.c
- gcc -Wall -O2 -I../misc -o ssim seq-std.c ssim.c ../misc/isa.c -lm

In seq/ssim.c:

```
int sim_main(int argc, char **argv) {
        run_tty_sim();
}
Run_tty_sim() {
        icount = sim_run(instr_limit, &status, &result_cc);
}
```

seq/ssim: The Sequential Simulator

In seq/ssim.c

```
int sim_run(int max_instr, byte_t *statusp, cc_t *ccp)
{
   while (icount < max_instr) {
      run_status = sim_step();
      icount++;
      if (run_status != STAT_AOK)
         break;
   }
}
```

With Slides from Bryant and O'Hallaron

seq/ssim: The Sequential Simulator

In seq/ssim.c

```
static byte_t sim_step()  {
   update_state(); /* Update state from last cycle */
   valp = pc;
   instr = …
   imem_error = …;
   imem_icode = HI4(instr);
   imem_ifun = LO4(instr);
   icode = gen_icode();
   ifun  = gen_ifun();
   instr_valid = gen_instr_valid();
   …
```

With Slides from Bryant and O'Hallaron

seq/ssim: The Sequential Simulator

In seq/ssim.c

```
static byte_t sim_step() {
        ...
    if (gen_need_regids()) { ...
    if (gen_need_valC()) { ...
    srcB = gen_srcB();
   if (srcB != REG_NONE) {
      valb = get_reg_val(reg, srcB);
   ...
   cond = cond_holds(cc, ifun);
   destE = gen_dstE();
   destM = gen_dstM();
```


seq/ssim: The Sequential Simulator

In seq/seq-std.hcl

```
# Determine instruction code
int icode = [
      imem_error: INOP;
      1: imem_icode;        # Default: get from instruction memory
];

# Determine instruction function
int ifun = [
      imem_error: FNONE;
      1: imem_ifun;         # Default: get from instruction memory
];
```

seq/ssim: The Sequential Simulator

In seq/seq-std.hcl

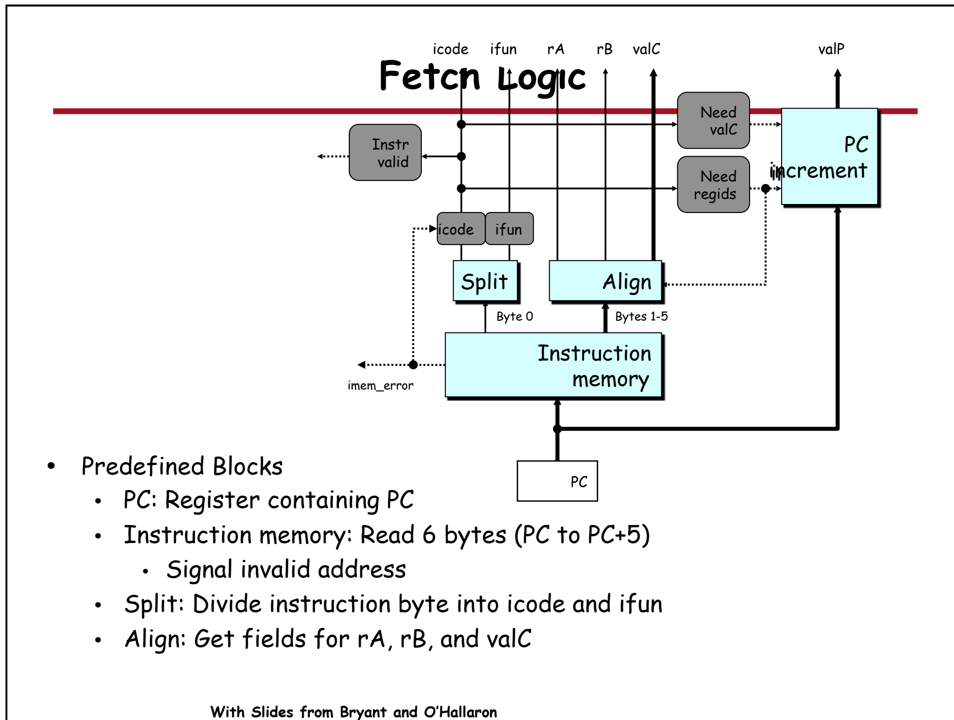
```
# Does fetched instruction require a regid byte?
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
        icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
```

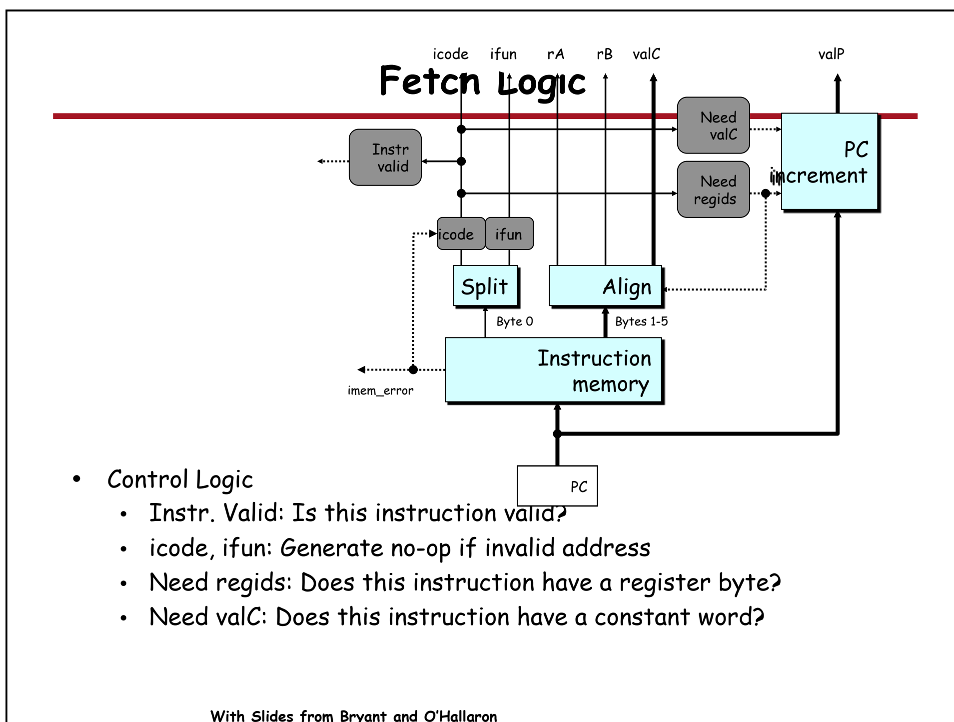

seq/ssim: The Sequential Simulator

In seq/seq-std.c

```
int gen_icode()  { … }
int gen_ifun()  { … }
int gen_need_regids() {
  return ((icode) == (I_RRMOVL) || (icode) == (I_ALU) || (icode) ==
     (I_PUSHL) || (icode) == (I_POPL) || (icode) == (I_IRMOVL) || (icode)
      == (I_RMMOVL) || (icode) == (I_MRMOVL));
}
int gen_need_valC() {… }
```

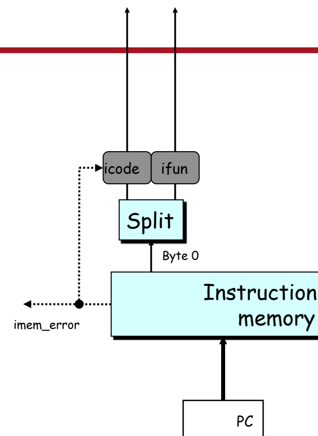


- Predefined Blocks
  - PC: Register containing PC
  - Instruction memory: Read 6 bytes (PC to PC+5)
    - Signal invalid address
  - Split: Divide instruction byte into icode and ifun
  - Align: Get fields for rA, rB, and valC



- Control Logic
  - Instr. Valid: Is this instruction valid?
  - icode, ifun: Generate no-op if invalid address
  - Need regids: Does this instruction have a register byte?
  - Need valC: Does this instruction have a constant word?

Fetch Control Logic in HCL




```
# Determine instruction code
int icode = [
        imem_error: INOP;
        1: imem_icode;
];

# Determine instruction function
int ifun = [
        imem_error: FNONE;
        1: imem_ifun;
];
```

With Slides from Bryant and O'Hallaron

Fetch Control Logic in HCL

Instruction	Encoding
halt	0 0
nop	1 0
cmovXX rA, rB	2 fn rA rB
irmovl V, rB	3 0 8 rB V
rmmovl rA, D(rB)	4 0 rA rB D
mrmovl D(rB), rA	5 0 rA rB D
OPl rA, rB	6 fn rA rB
jXX Dest	7 fn Dest
call Dest	8 0 Dest
ret	9 0
pushl rA	A 0 rA 8
popl rA	B 0 rA 8

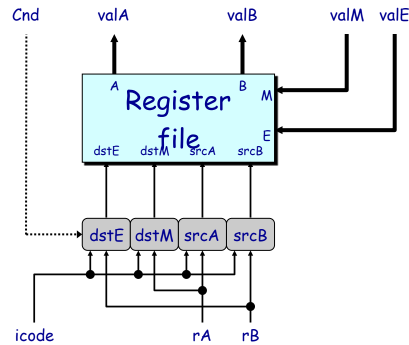
```
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                   IIRMOVL, IRMMOVL, IMRMOVL };

bool instr_valid = icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
              IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
```

With Slides from Bryant and O'Hallaron

Decode Logic

- Register File
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 15 (0xF) (no access)



Control Logic

- srcA, srcB: read port addresses
- dstE, dstM: write port addresses

Signals

- Cnd: Indicate whether or not to perform conditional move
  - Computed in Execute stage

With Slides from Bryant and O'Hallaron

A Source

Stage	Instruction	Comment
	OPl rA, rB	
Decode	valA ← R[rA]	Read operand A
	cmovXX rA, rB	
Decode	valA ← R[rA]	Read operand A
	`rmmovl` rA, D(rB)	
Decode	valA ← R[rA]	Read operand A
	`popl` rA	
Decode	valA ← R[`%esp`]	Read stack pointer
	jXX Dest	
Decode		No operand
	`call` Dest	
Decode		No operand
	`ret`	
Decode	valA ← R[`%esp`]	Read stack pointer

```
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

With Slides from Bryant and O'Hallaron

E Destination

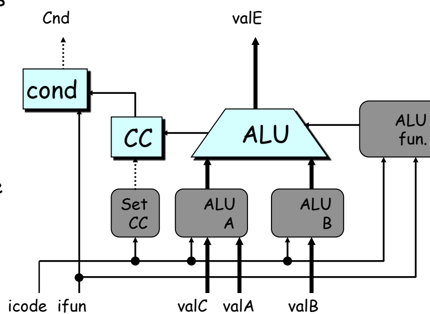
	OPl rA, rB	
Write-back	R[rB] ← valE	Write back result
	cmovXX rA, rB	
Write-back	R[rB] ← valE	Conditionally write back result
	rmmovl rA, D(rB)	
Write-back		None
	popl rA	
Write-back	R[%esp] ← valE	Update stack pointer
	jXX Dest	
Write-back		None
	call Dest	
Write-back	R[%esp] ← valE	Update stack pointer
	ret	
Write-back	R[%esp] ← valE	Update stack pointer

```
int dstE = [
        icode in { IRRMOVL } && Cnd : rB;
        icode in { IIRMOVL, IOPL} : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't write any register
];
```

With Slides from Bryant and O'Hallaron

Execute Logic

- Units
  - ALU
    - Implements 4 required functions
    - Generates condition code values
  - CC
    - Register with 3 condition code bits
  - cond
    - Computes conditional jump/move flag
- Control Logic
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?



With Slides from Bryant and O'Hallaron

ALU A Input

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	cmovXX rA, rB	
Execute	valE ← 0 + valA	Pass valA through ALU
	rmmovl rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	popl rA	
Execute	valE ← valB + 4	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	valE ← valB + -4	Decrement stack pointer
	ret	
Execute	valE ← valB + 4	Increment stack pointer

```
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];
```

With Slides from Bryant and O'Hallaron

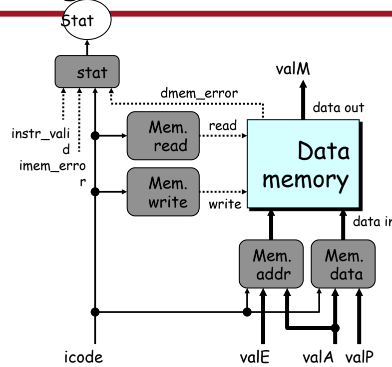
ALU Oper-ation

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	cmovXX rA, rB	
Execute	valE ← 0 + valA	Pass valA through ALU
	rmmovl rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	popl rA	
Execute	valE ← valB + 4	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	valE ← valB + -4	Decrement stack pointer
	ret	
Execute	valE ← valB + 4	Increment stack pointer

```
int alufun = [
        icode == IOPL : ifun;
        1 : ALUADD;
];
```

With Slides from Bryant and O'Hallaron

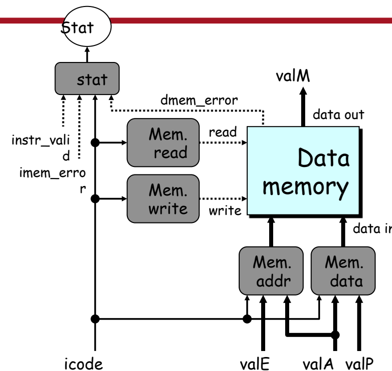
Memory Logic



- Memory
  - Reads or writes memory word
- Control Logic
  - stat: What is instruction status?
  - Mem. read: should word be read?
  - Mem. write: should word be written?
  - Mem. addr.: Select address
  - Mem. data.: Select data

With Slides from Bryant and O'Hallaron

Instruction Status



- Control Logic
  - stat: What is instruction status?

```
## Determine instruction status
int Stat = [
        imem_error || dmem_error : SADR;
        !instr_valid: SINS;
        icode == IHALT : SHLT;
        1 : SAOK;
];
```

With Slides from Bryant and O'Hallaron

Memory Address

	OPl rA, rB	
Memory		No operation

	rmmovl rA, D(rB)	
Memory	M_4[valE] ← valA	Write value to memory

	popl rA	
Memory	valM ← M_4[valA]	Read from stack

	jXX Dest	
Memory		No operation

	call Dest	
Memory	M_4[valE] ← valP	Write return value on stack

	ret	
Memory	valM ← M_4[valA]	Read return address

```
int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
        icode in { IPOPL, IRET } : valA;
        # Other instructions don't need address
];
```

With Slides from Bryant and O'Hallaron

Memory Read

	OPl rA, rB	
Memory		No operation

	rmmovl rA, D(rB)	
Memory	M_4[valE] ← valA	Write value to memory

	popl rA	
Memory	valM ← M_4[valA]	Read from stack

	jXX Dest	
Memory		No operation

	call Dest	
Memory	M_4[valE] ← valP	Write return value on stack

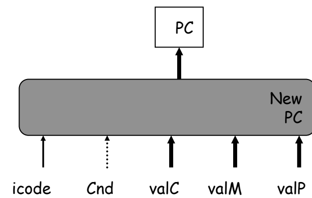
	ret	
Memory	valM ← M_4[valA]	Read return address

```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

With Slides from Bryant and O'Hallaron

PC Update Logic

- New PC
  - Select next value of PC



With Slides from Bryant and O'Hallaron

PC Update

	OPl rA, rB	
PC update	PC ← valP	Update PC

	rmmovl rA, D(rB)	
PC update	PC ← valP	Update PC

	popl rA	
PC update	PC ← valP	Update PC

	jXX Dest	
PC update	PC ← Cnd ? valC : valP	Update PC

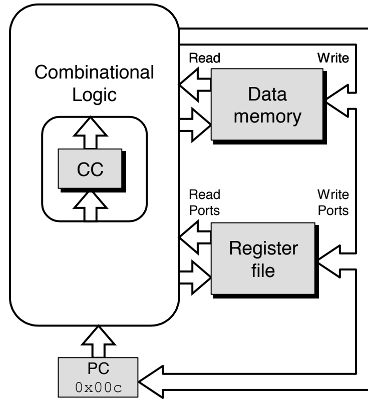
	call Dest	
PC update	PC ← valC	Set PC to destination

	ret	
PC update	PC ← valM	Set PC to return address

```
int new_pc = [
        icode == ICALL : valC;
        icode == IJXX && Cnd : valC;
        icode == IRET : valM;
        1 : valP;
];
```

With Slides from Bryant and O'Hallaron

SEQ Operation

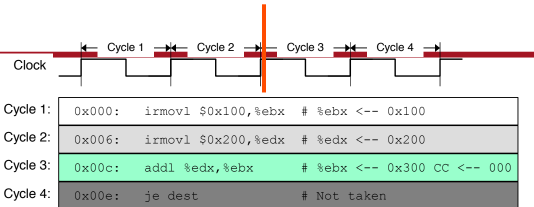


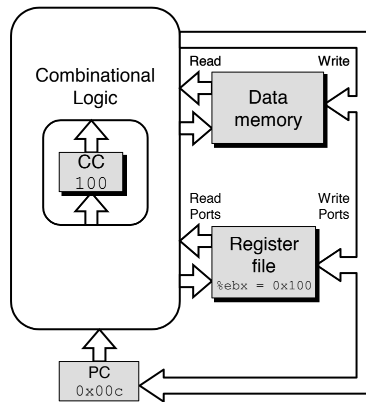
- State
  - PC register
  - Cond. Code register
  - Data memory
  - Register file

  All updated as clock rises
- Combinational Logic
  - ALU
  - Control logic
  - Memory reads
    - Instruction memory
    - Register file
    - Data memory

With Slides from Bryant and O'Hallaron

SEQ Operation #2

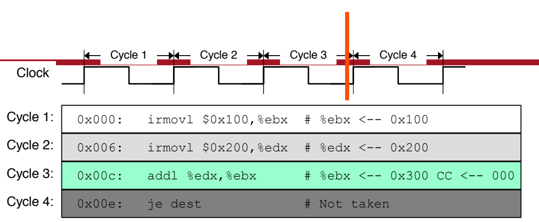


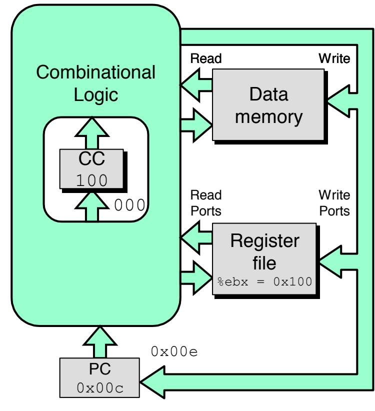


- state set according to second `irmovl` instruction
- combinational logic starting to react to state changes

With Slides from Bryant and O'Hallaron

SEQ Operation #3

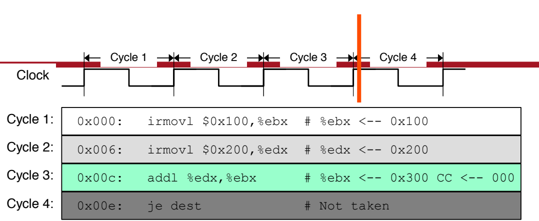


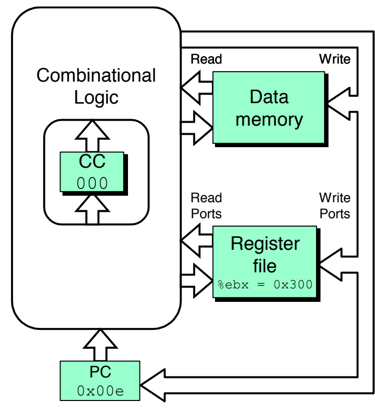


- state set according to second irmovl instruction
- combinational logic generates results for addl instruction

With Slides from Bryant and O'Hallaron

SEQ Operation #4

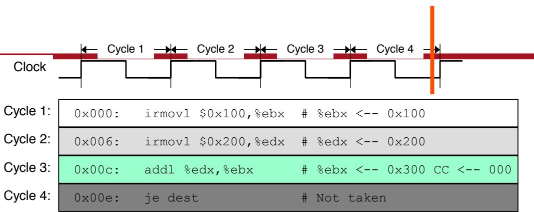


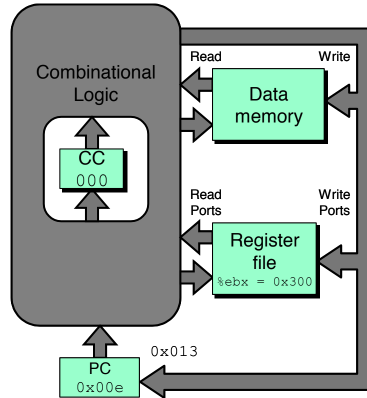


- state set according to addl instruction
- combinational logic starting to react to state changes

With Slides from Bryant and O'Hallaron

SEQ Operation #5





- state set according to `addl` instruction
- combinational logic generates results for `je` instruction

SEQ Summary

- Implementation
  - Express every instruction as series of simple steps
  - Follow same general flow for each instruction type
  - Assemble registers, memories, predesigned combinational blocks
  - Connect with control logic
- Limitations
  - Too slow to be practical
  - In one cycle, must propagate through instruction memory, register file, ALU, and data memory
  - Would need to run clock very slowly
  - Hardware units only active for fraction of clock cycle