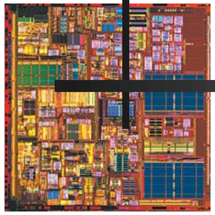
# ECE 411
Fall 2015

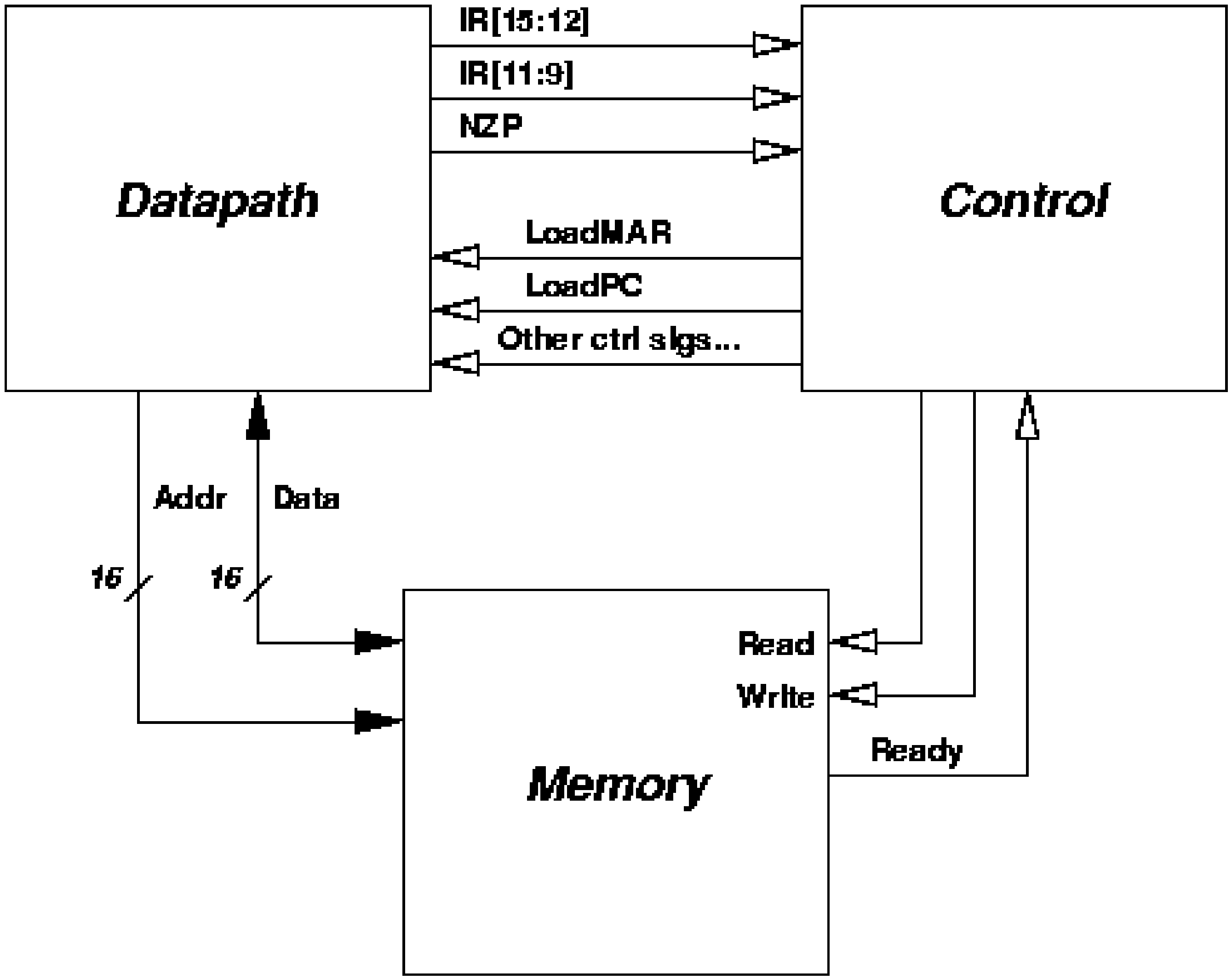
Lecture 2



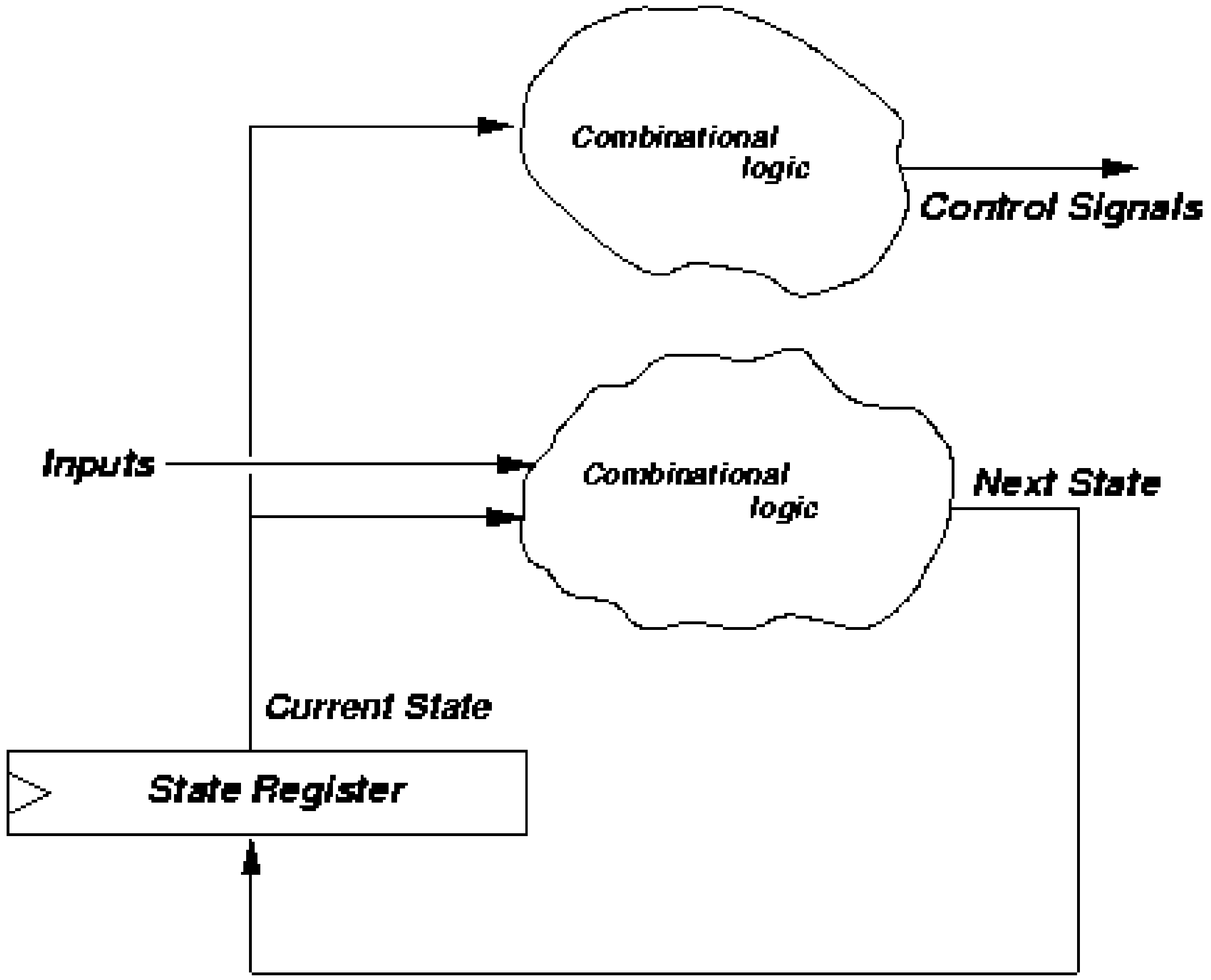
Control Unit Design; ISA Design; Performance Considerations

ILLINOIS

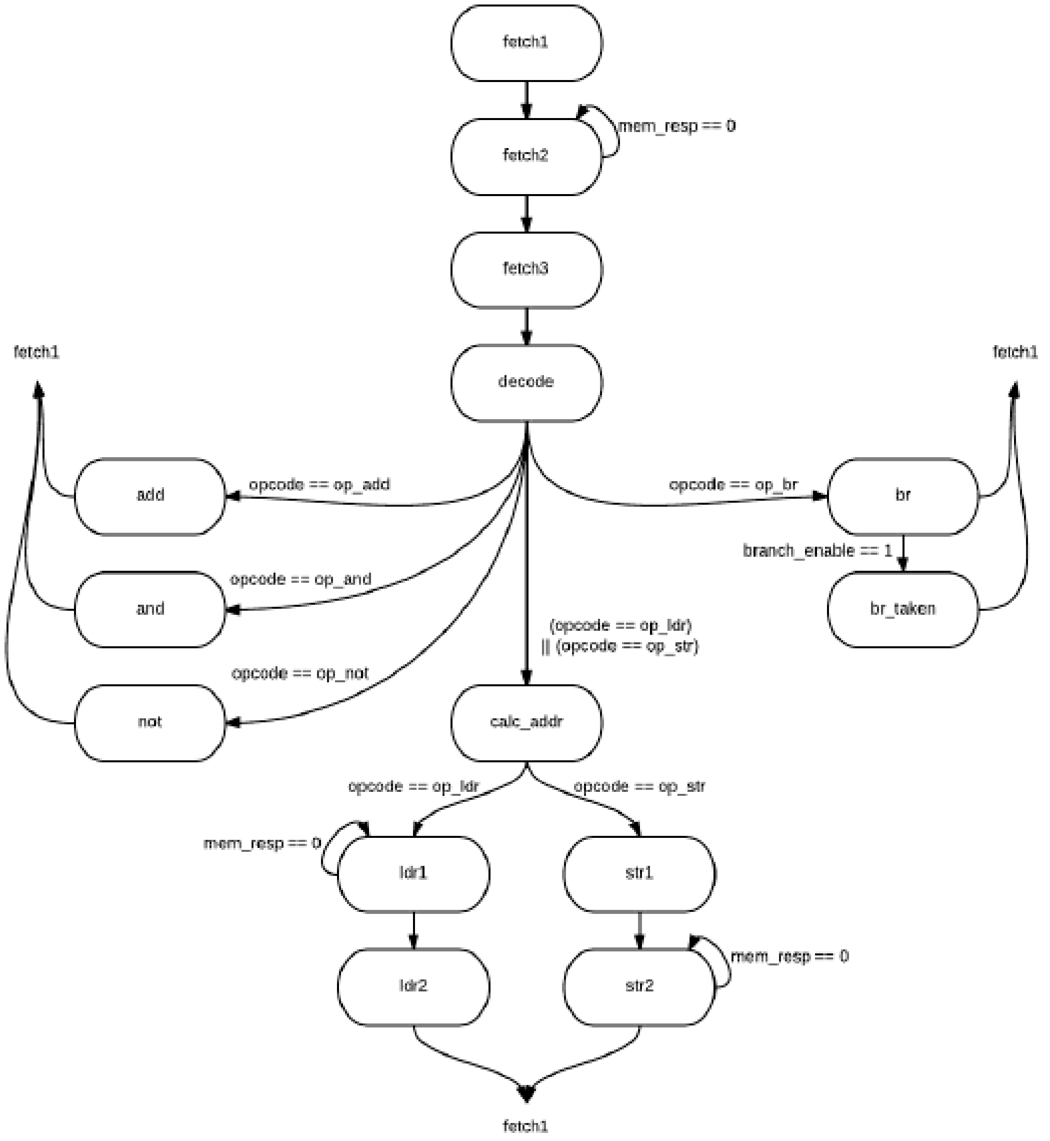
# Control Unit – Control Signals



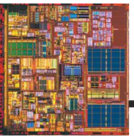
# Control Signals and Next State



## State Diagram



```
always_comb
begin : state_actions
/* Default assignments */
  load_pc = 1'b0;
  load_ir = 1'b0;
  load_regfile = 1'b0;
  aluop = alu_add;
  /* … */
  case(state)
    fetch1: begin
      /* MAR <= PC */
      marmux_sel = 1;
      load_mar = 1;
      /* PC <= PC + 2 */
      pcmux_sel = 0;
      load_pc = 1;
    end
    fetch2: begin
      /* Read memory */
      mem_read = 1;
      mdrmux_sel = 1;
      load_mdr = 1;
     end
    fetch3: begin
      /* Load IR */
      load_ir = 1;
     end
    decode: /* Do nothing */;
    s_add: begin
      /* DR <= SRA + SRB */
      aluop = alu_add;
      load_regfile = 1;
      regfilemux_sel = 0;
      load_cc = 1;
    end
  /*…*/
```



## Next State Logic in RTL Code

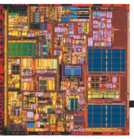
```
always_comb
begin : next_state_logic
    next_state = state;

    case(state)
       fetch1: next_state = fetch2;
       fetch2: if (mem_resp) next_state = fetch3;
       fetch3: next_state = decode;

       decode: begin
           case (dp.opcode)
               op_add:   next_state = s_add;
               op_and:   next_state = s_and;
               ...
            endcase
        end
       s_br: begin
            if (dp.nzp_match)
                next_state = s_br_taken;
            else
                next_state = fetch1;
       end

        ...
       default: next_state = fetch1;

    endcase
end
```



## Key ISA decisions

Instruction format – Length? Variable? Fixed? Fields?

How many registers?

Where do instruction operands reside?

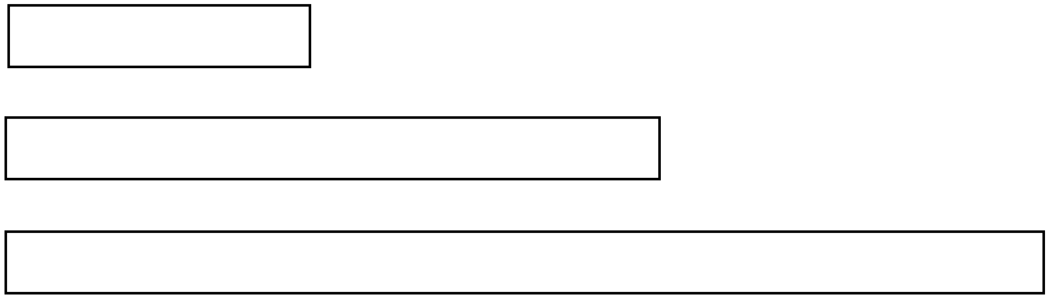
- e.g., can you add contents of memory to a register?

Operands

- how many? how big?
- how are memory addresses computed?

# Instruction Length

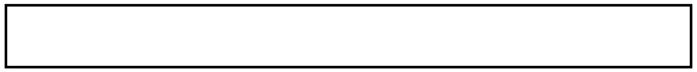
**Variable:**



**x86 – Instructions vary from 1 to 17 Bytes long**

**VAX – from 1 to 54 Bytes**

**Fixed:**

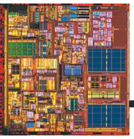


**MIPS, PowerPC, and most other RISC's:**

**all instruction are 4 Bytes long**

## Instruction Length

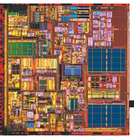
- Variable-length instructions (x86, VAX):
  - - require multi-step, complex fetch and decode.
  - + allow smaller binary programs that require less disk storage, less DRAM at runtime, less memory, bandwidth and better cache efficiency
- Fixed-length instructions (RISC's)
  - + allow easy fetch and decode.
  - + simplify pipelining and parallelism.
  - - result in larger binary programs that require more disk storage, more DAM at runtime, more memory bandwidth and lower cache efficiency



## ARM Case Study

- ARM (Advanced RISC Machine)
  - Started with fixed, 32-bit instruction length
  - Added Thumb instructions
    - A subset of the 32-bit instructions
    - All encoded in 16 bits
    - All translated into equivalent 32-bit instructions within the processor pipeline at runtime
    - Can access only 8 general purpose registers
- Motivated by many resource constrained embedded applications that require less disk storage, less DRAM at runtime, less memory, bandwidth and better cache efficiency

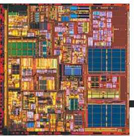
## How many registers?



- Most computers have a small set of <u>registers</u>
  - Memory to hold values that will be used soon
  - A typical instruction use 2 or 3 register values
- Advantages of a small number of registers:
  - It requires fewer instruction bits to specify which one.
  - Less hardware
  - Faster access (shorter wires, fewer gates)
  - Faster context switch (when all registers need saving)
- Advantages of a larger number:
  - Fewer <u>loads</u> and <u>stores</u> needed
  - Easier to express several operations in parallel

In 411, "load" means moving data from memory to register, "store" is reverse

## Where do operands reside (*when the ALU needs them*)?



Stack machine:

- "Push" loads memory into 1st register ("top of stack"), moves other regs down
- "Pop" does the reverse.
- "Add" combines contents of first two registers, moves rest up.

Accumulator machine:

- Only 1 register (called the "accumulator")
- Instruction include "store" and "acc $\leftarrow$ acc + mem"

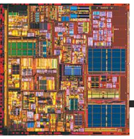
Register-Memory machine :

- Arithmetic instructions can use data in registers and/or memory

Load-Store Machine (aka Register-Register Machine):

- Arithmetic instructions can only use data in registers.

ILLINOIS



## Comparing the ISA classes

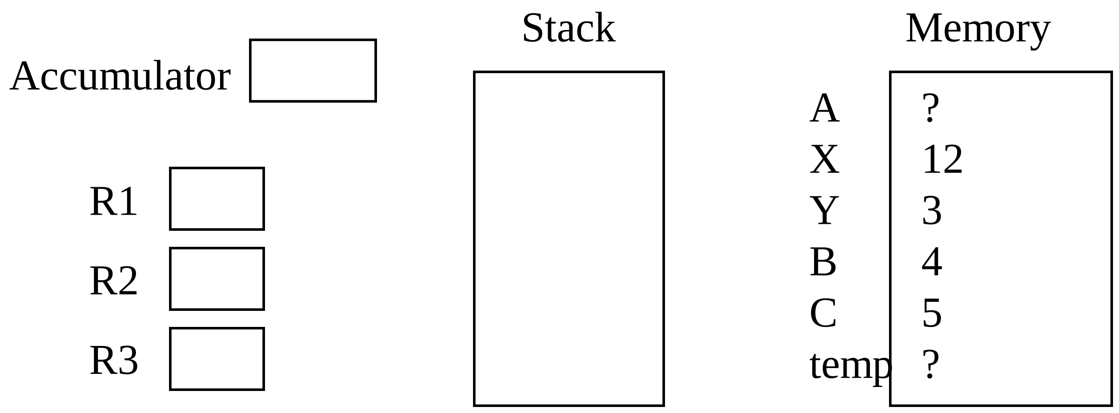
Code sequence for `C = A + B`

| Stack | Accumulator | Register-Memory | Load-Store |
|---|---|---|---|
| `Push A` | `Load A` | `Add C, A, B` | `Load R1,A` |
| `Push B` | `Add B` | | `Load R2,B` |
| `Add` | `Store C` | | `Add R3,R1,R2` |
| `Pop C` | | | `Store C,R3` |
| Java VMs | DSPs | VAX, x86 partially | |

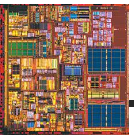
# Comparing the ISA classes

Stack Accumulator Reg-Mem Load-store

$$\boxed{A = X*Y + X*Z}$$



# Load-store architectures



can do:

add r1=r2+r3

load r3, M(address)

store r1, M(address)

⇒ forces heavy dependence on registers, which works for today's CPUs

can't do:

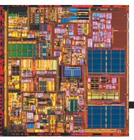
```
add r1=r2+M(address)
```

- more instructions

+ fast implementation (e.g., easy pipelining)

+easier to keep instruction lengths fixed

## Instruction formats *-what does each bit mean?*



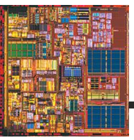
Machine needs to determine quickly,

- "This is a 6-byte instruction"
- "Bits 7-11 specify a register"
- ...
- Serial decoding bad

Having many different instruction formats...

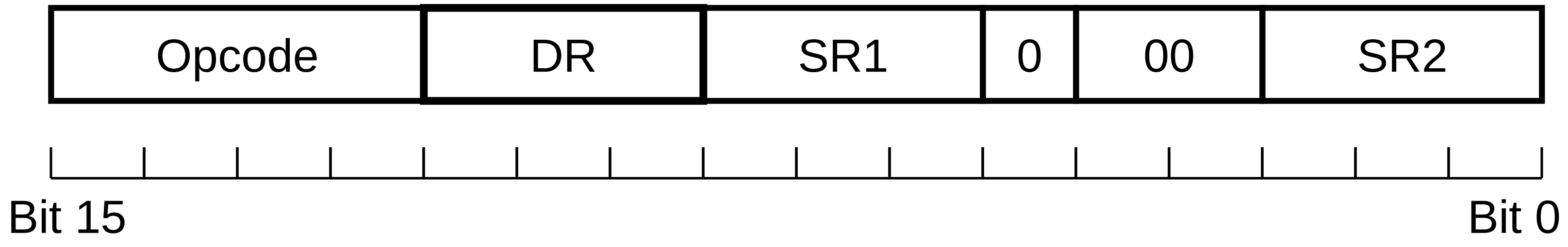
- complicates decoding
- uses instruction bits (to specify the format)

**What would be a good thing about having many different instruction formats?**

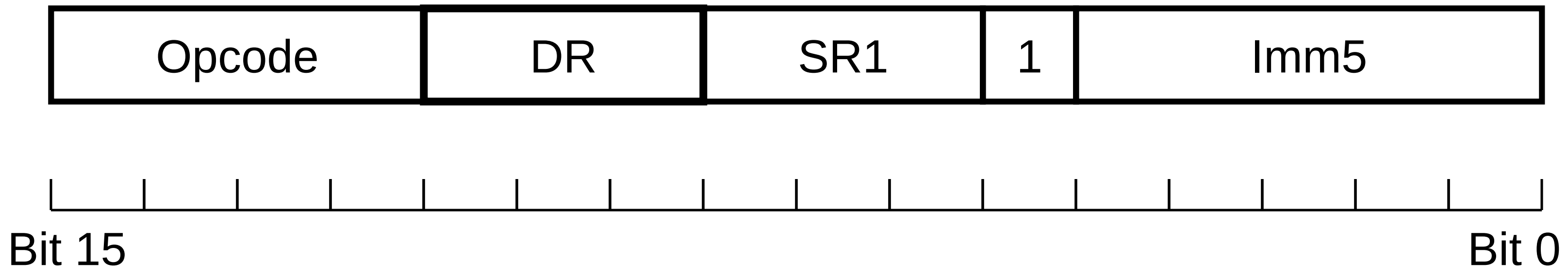


## LC-3bInstruction Formats

- ADD, AND (without Immediate)

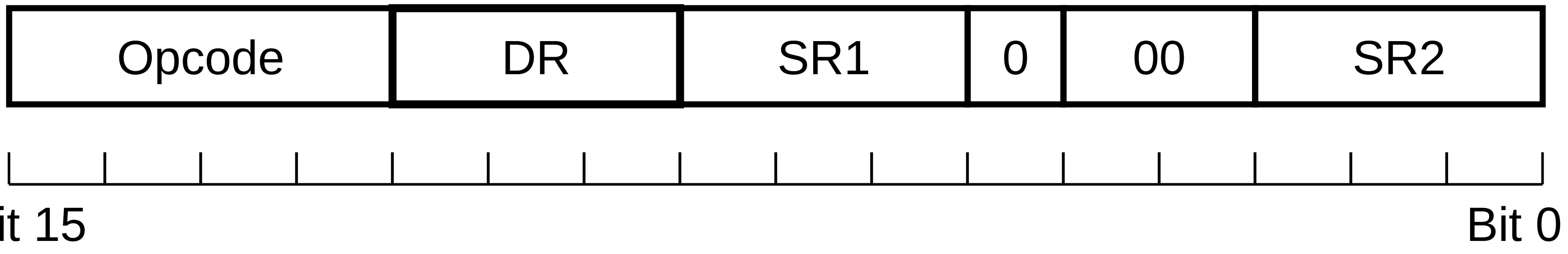


- ADD, AND (with Immediate), NOT

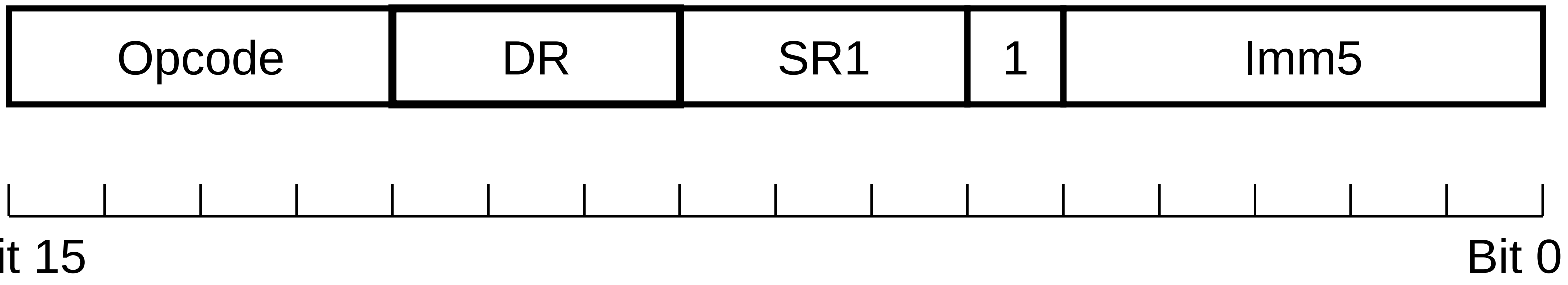


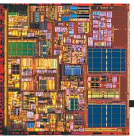
## LC-3b Instruction Formats

- ADD, AND (without Immediate)



- ADD, AND (with Immediate), NOT



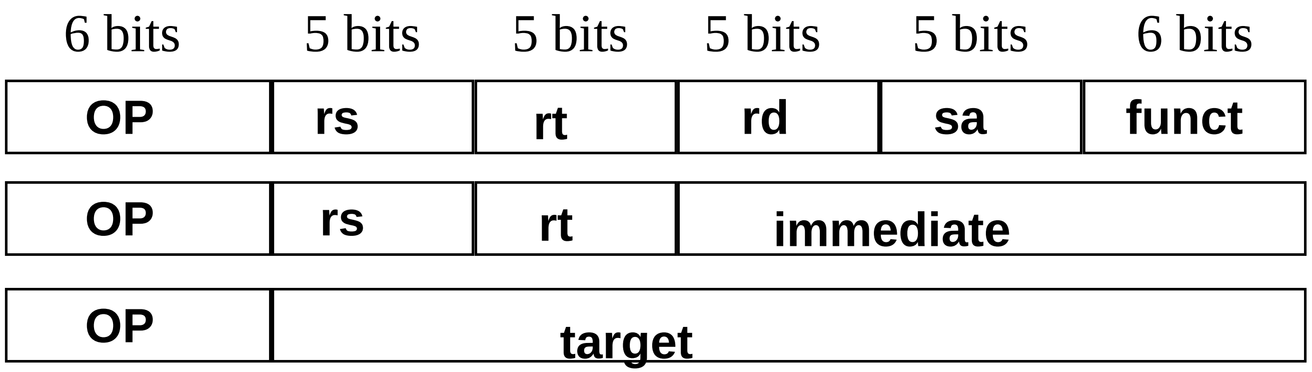


# MIPS Instruction Formats

| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|---|
| r format | **OP** | **rs** | **rt** | **rd** | **sa** | **funct** |
| i format | **OP** | **rs** | **rt** | **immediate** | | |
| j format | **OP** | **target** | | | | |

- for instance, “`add r1, r2, r3`” has
  - OP=0, rs=2, rt=3, rd=1, sa=0 (shift amount), funct=32
  - 000000 00010 00011 00001 00000 100000
- opcode (OP) tells the machine which format

# MIPS ISA Tradeoffs



What if?

- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. Y = AX + B)

**Think about how sparsely the bits are used**

## Conditional branch

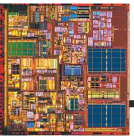
- How do you specify the destination of a branch/jump?
  - Theoretically, the destination is a full address
    - 16 bits for LC3b
    - 32 bits for MIPS

## Conditional branch

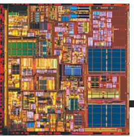
- How do you specify the destination of a branch/jump?
- studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else).

## Conditional branch

- How do you specify the destination of a branch/jump?
- studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else).
    - we can specify a relative address in much fewer bits than an absolute address
    - e.g., beq $1, $2, 100    => if ($1 == $2) PC = PC + 100 * 4



## Conditional branch



- How do you specify the destination of a branch/jump?
- studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else).
  - we can specify a relative address in much fewer bits than an absolute address
  - e.g., beq $1, $2, 100 => if ($1 == $2) PC = PC + 100 * 4
- How do we specify the condition of the branch?

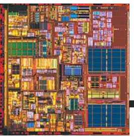
# MIPS conditional branches

- beq, bne *beq r1, r2, addr => if (r1 == r2) goto addr*
- slt \$1, \$2, \$3 => if (\$2 < \$3) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions

Always, never, !=, = =, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
    w = w+1;
else
    w = 5;
```

```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```



## Jumps

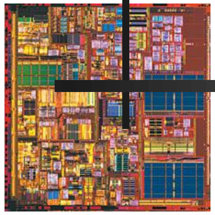
- need to be able to jump to an absolute address sometime
- need to be able to do procedure calls and returns

- jump -- j 10000  => PC = 10000
- jump and link -- jal 100000 => $31 = PC + 4; PC = 10000
  - used for procedure calls

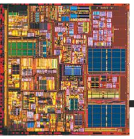
| OP | target |
|---|---|

- jump register -- jr $31 => PC = $31
  - used for returns, but can be useful for lots of other things.

## Computer Performance



## Computer Performance: TIME, TIME, TIME



- Response Time (latency)
  - — How long does it take for my job to run?
  - — How long does it take to execute a job?
  - — How long must I wait for the database query?
- Throughput
  - — How many jobs can the machine run at once?
  - — What is the average execution rate?
  - — How much work is getting done?

- *If we upgrade a machine with a new processor what do we increase?*

*If we add a new machine to the lab what do we increase?*

ILLINOIS

# Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

| | Inst Count | CPI | Clock Rate |
|---|---|---|---|
| Program | X | | |
| Compiler | X | (X) | |
| Inst. Set. | X | X | |
| Organization | X | X | X |
| Technology | | | X |

## P&H Definition of Performance

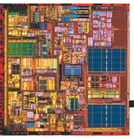
- For some program running on machine X,

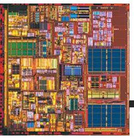
  $$\text{Performance}_X = 1 \,/\, \text{Execution time}_X$$

- "X is n times faster than Y"

  $$\text{Performance}_X \,/\, \text{Performance}_Y = n$$

- Problem:
  - machine A runs a program in 20 seconds
  - machine B runs the same program in 25 seconds





## How to Improve Performance

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

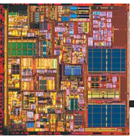
So, to improve performance (everything else being equal) you can either

_________ the # of required cycles for a program, or

_________ the clock cycle time or,  said another way,

_________ the clock rate.

ILLINOIS



## Now that we understand cycles

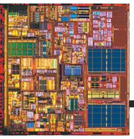
- A given program will require
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds
- We have a vocubulary that relates these quantities:
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (cycles per instruction)

    *a floating point intensive application might have a higher CPI*
  - MIPS (millions of instructions per second)

    *this would be higher for a program using simple instructions*

ILLINOIS

## CPI Example



- Suppose we have two implementations of the same instruction set architecture (ISA).

  For some program,
  Machine A has a clock cycle time of 10 ns. and a CPI of 2.0
  Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

  What machine is faster for this program, and by how much?

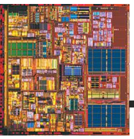
- *If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will likely be identical during a comparison?*

# # of Instructions Example

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles (respectively).

  The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C
  The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.

  Which sequence will be faster? How much?
  What is the CPI for each sequence?



## MIPS example

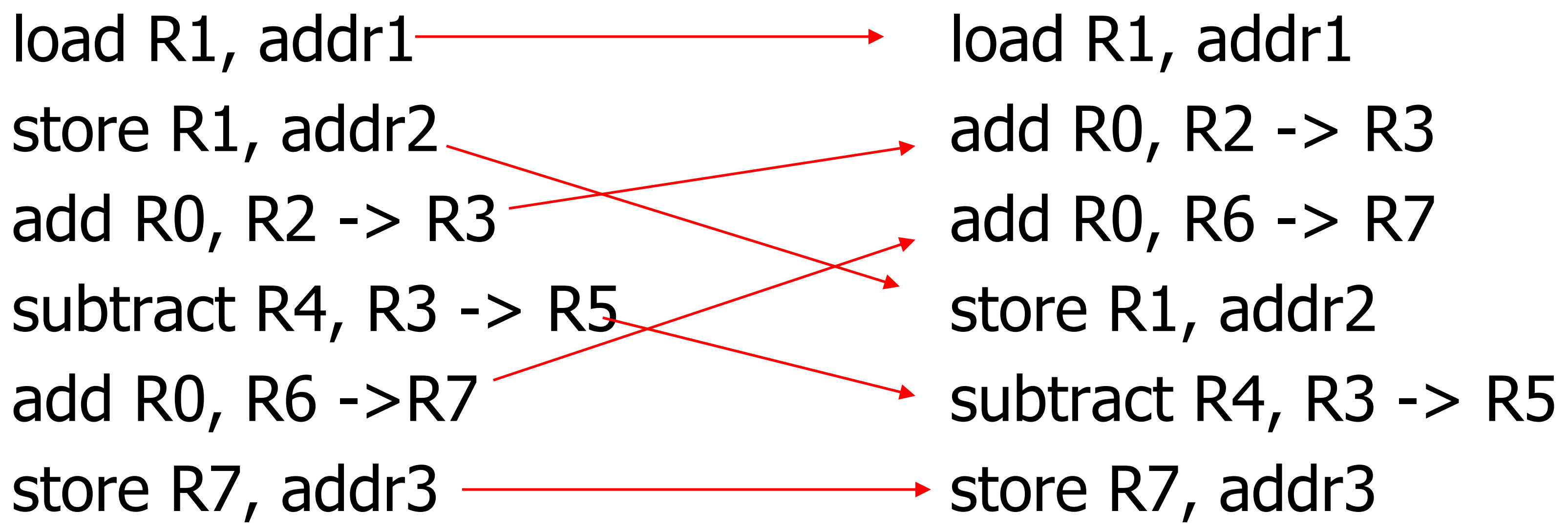
- Two different compilers are being tested for a 100 MHz. machine with
  three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles (respectively). Both compilers are used to produce code for a large piece of software.

  The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

  The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

- Which sequence will be faster according to MIPS?
- Which sequence will be faster according to execution time?

## Which is faster in the same CPU?

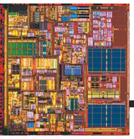


```
load R1, addr1
store R1, addr2
add R0, R2 -> R3
subtract R4, R3 -> R5
add R0, R6 ->R7
store R7, addr3
```

```
load R1, addr1
add R0, R2 -> R3
add R0, R6 -> R7
store R1, addr2
subtract R4, R3 -> R5
store R7, addr3
```

**Twice as fast on some machines and same on others**

**CPI can vary due to interactions among instructions!**



## READ CHAPTER 2
## COMPLETE MP0