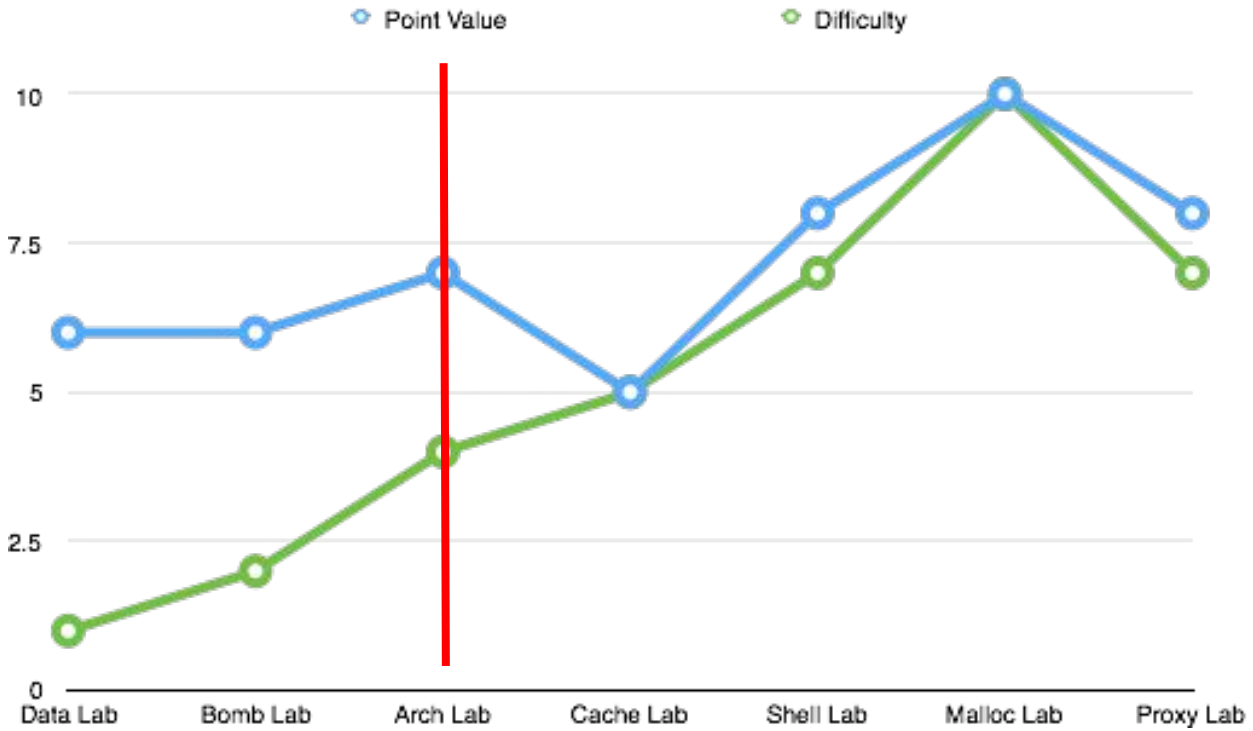
# 18-600 Recitation #6

Architecture Lab

# Overview

- Where we are
- Archlab Intro
- Optimizing a Y86 program
- Intro to HCL
- Optimizing Processor Design

# Where we Are


Point Value
Difficulty
10
7.5
5
2.5
0
Data Lab
Bomb Lab
Arch Lab
Cache Lab
Shell Lab
Malloc Lab
Proxy Lab

# Archlab Intro

- Part A: write and simulate Y86-64 programs
  - Covered in last week's recitation
- Part B: optimize a Y86 program and processor design
  - This week!

# Part B

- Covers the optimization of Y86 programs and pipelined processor design
- Will be modifying two files: ncopy.ys and pipe-full.hcl
    - Goal for **ncopy.ys**: improve the performance by reducing the CPE (cycles per element, the number of cycles used / the number of elements copied)
    - Goal for **pipe-full.hcl**: implement the IADDQ instruction in the processor design, add any additional optimizations

# Optimizing a Y86 Program

- Reduce the CPE of ncopy by reducing the number of operations from inefficient code, data dependencies and conditional control flow
- Methods:
  - Streamline Code
  - Reduce Data Dependencies
  - Loop Unrolling
  - Jump Tables (and other control flow changes)

# Streamline Code

- Convert code to use IADDQ instruction

iaddq V, rB


```
irmovq $8, %r8
addq %r8, %r10
```

→

```
iaddq $8, %r10
```

- Reduce memory accesses

```
.L1
mrmovq (%rbx), %rax
...
jne .L1
```

→

```
mrmovq (%rbx), %rax
.L1
...
jne .L1
```

- Reducing number of register operations

```
.L1
irmovq $8, %rax
...
jne .L1
```

→

```
irmovq $8, %rax
.L1
...
jne .L1
```

And more!

# Reduce Data Dependencies

- Results from one operation depend on the results of the other, stalling the pipeline execution
- Avoid clumping dependent operations together

```
irmovq $8, %r8
addq %r8, %r10
irmovq $10, %r9
addq %r9, %rbx
```



```
irmovq $8, %r8
irmovq $10, %r9
addq %r8, %r10
addq %r9, %rbx
```

# Loop Unrolling

- Program transformation which **reduces the number of iterations** for a loop by increasing the number of elements computed on each iteration
- **Reduces the number of operations** that do not contribute directly to loop result, including loop indexing and conditional branching
- Exposes additional ways to optimize computation (multiple computations may be combined)

# Loop Unrolling

```
# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum: irmovq $8,%r8        # Constant 8
     irmovq $1,%r9       # Constant 1
     xorq %rax,%rax      # sum = 0
     andq %rsi,%rsi      # Set CC
     jmp     test         # Goto test
loop:     mrmovq (%rdi),%r10   # Get *start
     addq %r10,%rax       # Add to sum
     addq %r8,%rdi        # start++
     subq %r9,%rsi        # count--.  Set CC
test:     jne    loop     # Stop when 0
     ret                 # Return
```

```
# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum: irmovq $8,%r8        # Constant 8
     irmovq $2,%r9       # Constant 2
     xorq %rax,%rax      # sum = 0
     subq %r9,%rsi       # Set CC
     jmp     test         # Goto test
loop:     mrmovq (%rdi),%r10   # Get *start
     addq %r10,%rax       # Add to sum
     addq %r8,%rdi        # start++
     mrmovq (%rdi),%r10
     addq %r10,%rax       # Add to sum
     addq %r8,%rdi        # start++
     subq %r9,%rsi        # count-=2.  Set CC
test:     jge     loop    # Stop when < 0
     addq %r9, %rsi
     je done       # Goto done if 0
     mrmovq (%rdi),%r10   # Get *start
     addq %r10,%rax       # Add to sum
done: ret                  # Return
```

# Jump Tables

- Jump table is an abstract data structure to transfer control
- It is an array where each entry is the address of a code segment
- Each code segment implements set of actions which should be taken when a particular condition is satisfied
- Time taken is invariant of the number of switch cases

# Jump Tables: Example

```
void switch_eg(long x, long n, long *dest)
{
        long val = x;

        switch (n) {

                case 100:
                    val *= 13;
                    break;

                case 102:
                    val += 10;

                case 103:
                    val += 11;
                    break;

                case 104:
                case 106:
                    val *= val;
                    break;

                default:
                    val = 0;
                }
        *dest = val;
}
```

Example switch statement

```
1       void switch_eg_impl(long x, long n, long *dest)
2       {
3           static void *jt[7] = {
4               &&loc_A, &&loc_def, &&loc_B,
5               &&loc_C, &&loc_D, &&loc_def,
6               &&loc_D
7           };
8           unsigned long index = n - 100;
9           long val;
10          if (index > 6)
11                goto loc_deg;
12          goto *jt[index];
13      loc_A:    /* Case 100 */
14          val = x * 13;
15          goto done;
16      loc_B:    /* Case 102 */
17          x = x + 10;
18      loc_C:    /* Case 103 */
19          val  = x + 1;;
20          goto done;
21      loc_D:    /* Case 104, 106 */
22          val = x * x;
23          goto done;
24      loc_def: /* Default case */
25          val = 0;
26      done:
27          *dest = val;
28      }
```

Translation into extended C
(labels as values)

# Jump Tables: Example

```
void switch_eg(long x, long n, long *dest)
{
        long val = x;

        switch (n) {

                case 100:
                    val *= 13;
                    break;

                case 102:
                    val += 10;

                case 103:
                    val += 11;
                    break;

                case 104:
                case 106:
                    val *= val;
                    break;

                default:
                    val = 0;
                }
        *dest = val;
}
```

Example switch statement

```
1          .section        .rodata
2          .align 8
3        .L4
4          quad .L3 Case 100
5          quad .L8 Case 101: default
6          quad .L5 Case 102
7          quad .L6 Case 103
8          quad .L7 Case 104
9          quad .L8 Case 105: default
10         quad .L7 Case 106
```

Jump Table

```
1       switch_eg:
2         subq $100, %rsi            Compute index = n - 100
3         cmpq $6, %rsi              Compare index:6
4         ja   .L8                   If >, goto loc_def
5         jmp  *.L4(,%rsi,8)         Goto *jg[index]
6       .L3:                         loc_A:
7         leaq (%rdi, %rdi, 2), %rax   3*x
8         leaq (%rdi, %rax, 4), %rdi   val = 13*x
9         jmp  .L2                     goto done
10      .L5:                         loc_B:
11        addq $10, %rdi               x = x + 10
12      .L6:                         loc_C:
13        addq $11, %rdi               val = x + 11
14        jmp .L2                      goto done
15      .L7                          loc_D:
16        imulq %rdi, %rdi             val = x * x
17        jmp .L2                      goto done
18      .L8:                         loc_def:
19        movl $0, %edi                val = 0
20      .L2:                         done:
21        movl $rdi, (%rdx)            *dest = val
22        ret                          Return
```

Assembly Code

# Intro to HCL

HCL is used to describe the control logic of the different processor designs

HCL has some of the features of a hardware description language (HDL), allowing users to describe Boolean functions and word-level selection operations

It differs from HDL in terms of:

- declaring registers and other storage elements
- looping and conditional constructs
- module definition

HCL is really just a language for generating a very stylized form of C code

Advantage: Clearly separate the functionality of the hardware

# HCL expressions

Define the behavior of a block of combinational logic

- bool name = bool-expr;
- int name = int-expr;

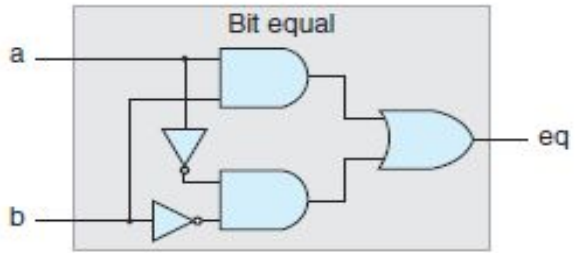
Logical Operations:

- a && b
- A || b
- !a

# HCL expressions contd...

Word Comparisons:

- a == b
- a != b
- a <= b



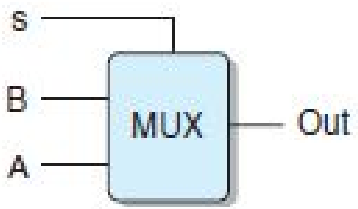

Set Membership:

- a in {b, c, d}

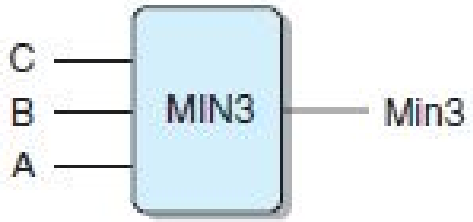
Boolean expression:

bool eq = (a && b) || (!a && !b);

# Word Expressions




```
int Out = [
        s : A;
        1 : B;
];
```




```
int Min3 = [
        A <= B && A <= C : A;
        B <= A && B <= C : B;
        1 :C;
];
```

# Adding a new Instruction: Example

- Why add new instructions ?
  - Reduce number of CPI
  - Comes close to C syntax and eases programming
- Eg: iaddq
  - Eg: iaddq: Add an immediate value to a register
  - Useful when used within loop body
  - Useful when there are insufficient registers to hold immediate values
  - **May help in arch lab, points allocated for it**
- Eg: Array indexing
  - Example from lecture
  - Supporting array indexing in the instruction set can simplify programming
  - Not required to be done for the lab

```
iaddq $8, %r9      vs      irmovq $8, %r8
                           addq %r8, %r9
```

```
/* Find number of elements in Null terminated list */
long len (long a[])
{
    long len;
    for (len = 0; a[len]; len++)
      ;
    Return len;
}
```

**Sample C Program**

```
L3:
     addq $1,%rax
     cmpq $0, (%rdi,%rax,8)
     jne L3
```

**X86 assembly**

```
Loop:
    addq %r8, %rax          # len++
    addq %r9, %rdi          # a++
    mrmovq(%rdi), %rdx      # val = *a
    andq %rdx, %rdx         # Test val
    jne Loop                # If !0,
```

**Y86 assembly**

# Adding a new instruction to HCL: Tips

- All instructions have to be checked for valid opcode, arguments
- Define the source and destination values at each stage in the pipeline
  - Refer to slides 48-65 in lecture 8
  - Notice conventions such as
    - valC: constant value
    - valP: value loaded from instruction pointer
    - valM: value loaded from memory
    - M[] : value in memory
    - valA/valB: value loaded from registers rA/rB
    - Reg[rA]: value in register rA
    - valE: value computed in execute stage
- Source and destination values can be read from registers or memory
- Both source and destination cannot be memory locations
- Execution of instruction may need checking/setting condition codes

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

Example illustrating HCL naming conventions

# Optimizing Processor Design

- Pipeline stalls and bubbles increase the CPI
  - Inserted to avoid data and control hazards
- They can be avoided by **forwarding**
- Forwarding:
  - Recall that every stage of the pipeline has source and destination register files
  - The source register values of each stage carries forward the destination register values from the previous stage
  - But why wait till updated values reach previous stage ?
  - **Carry forward values from future stages**

# Forwarding

- Forwarding:
  - When do this? Couple of scenarios
    - Destination register of an ALU operation may be source register for the next ALU operation
    - Destination register of a memory operation may be source register for the next ALU operation
    - Destination register of a memory operation (Load) may be source register for the next memory operation (Store)

# Example program without forwarding

```
add r1, r2
sub r2, r3
and r2, r4
```

| | Src, Dst | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD | R1, R2 | IF | ID | EX | MEM | **WB** | | | | | |
| SUB | R2, R3 | | IF | Stall | Stall | Stall | ID | EX | WB | | |
| AND | R2, R4 | | | Stall | Stall | Stall | IF | ID | EX | MEM | WB |

- Decode stage of the second and third instructions receive the results in time from writeback

# Example program contd with forwarding

| | Src, Dst | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| ADD | R1, R2 | IF | ID | EX | MEM | WB | | | | |
| SUB | R2, R3 | | IF | ID | EX | MEM | WB | | | |
| AND | R2, R4 | | | IF | ID | EX | MEM | WB | | |

- The first forwarding is for value of R2 from $EX_{add}$ to $EX_{sub}$
- The second forwarding is also for value of R2 from $MEM_{add}$ to $EX_{and}$
- This code now can be executed without stalls

# Another example program with forwarding

```
add r1, r2
mrmovq d(r2), r3
rmmovq r3, d(r2)
```

| | Src, Dst | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| ADD | R1, R2 | IF | ID | EX | MEM | WB | | | | |
| MRMOVQ | d(R2), R3 | | IF | ID | EX | MEM | WB | | | |
| RMMOVQ | R3, d(R2) | | | IF | ID | EX | MEM | WB | | |

- The first forwarding is for value of R2 from $EX_{add}$ to $EX_{mrmovq}$
- The second forwarding is also for value of R2 from $MEM_{add}$ to $EX_{rmmovq}$
- The third forwarding is for value of R3 from $MEM_{mrmovq}$ to $MEM_{rmmovq}$

# Additional Notes

- Review chapter 5 for more information on how to optimize ncopy
- Review chapter 4 for more information on the processor architecture & HCL language
- Good luck!

# Questions?