

CSC 252: Computer Organization

Spring 2026: Lecture 14

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

If mispredicted: kill mis-executed instructions, start from the correct target

Static Prediction

- Always Taken
- Always Not-taken

Dynamic Prediction

- Dynamically predict taken/not-taken for each specific jump instruction

Static Prediction

Observation (Assumption really): Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

Strategy:

- Forward jumps (i.e., `if-else`): always predict not-taken
- Backward jumps (i.e., `loop`): always predict taken

```
cmpq    %rsi, %rdi
jle     .corner_case    <before>
<do_A>
.corner_case:          .L1: <body>
<do_B>                cmpq B, A
ret                   jl  .L1
                        <after>
```

 **Mostly not taken**

 **Mostly taken**

Static Prediction

Knowing branch prediction strategy helps us write faster code

- Any difference between the following two code snippets?
- What if you know that hardware uses the always non-taken branch prediction?

```
if (cond) {  
    do_A()  
} else {  
    do_B()  
}
```

```
if (!cond) {  
    do_B()  
} else {  
    do_A()  
}
```

Dynamic Prediction

- Simplest idea:
 - If last time taken, predict taken; if last time not-taken, predict not-taken
 - Called 1-bit branch predictor
 - Works nicely for loops

```
for (i=0; i <5; i++) {...}
```

Iteration #1	0	1	2	3	4
Predicted Outcome	N	T	T	T	T
Actual Outcome	T	T	T	T	N



Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```
for (i=0; i <5; i++) {...}
```

Predict with 1-bit	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T
Actual Outcome	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N
Predict with 2-bit	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

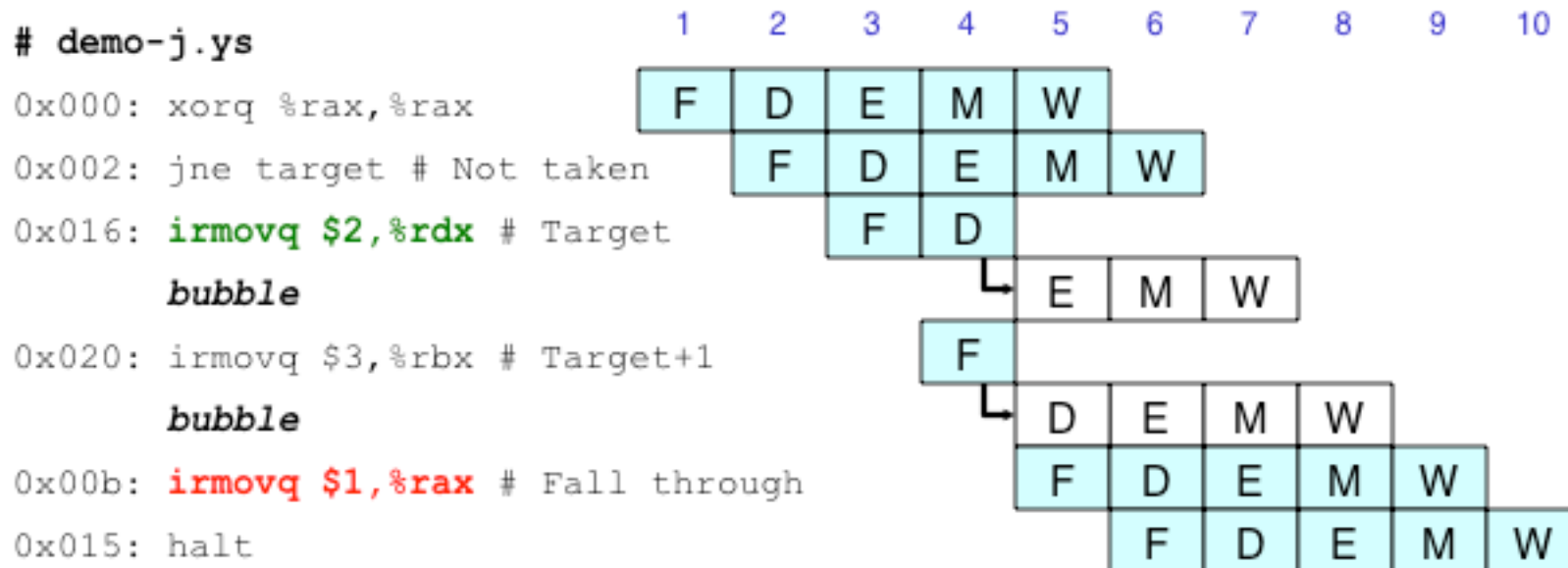
More Advanced Dynamic Prediction

- Look for past histories *across instructions*
- Branches are often correlated
 - Direction of one branch determines another

cond1 branch not-
taken means $(x \leq 0)$
branch taken

```
x = 0
if (cond1) x = 3
if (cond2) y = 19
if (x <= 0) z = 13
```

What Happens If We Mispredict?



Cancel instructions when mispredicted

- Assuming we detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by **bubbles**
- No side effects have occurred yet

Return Instruction

```
0x000:      irmovq Stack,%rsp      # Intialize stack pointer
0x00a:      call p                  # Procedure call
0x013:      irmovq $5,%rsi        # Return point
0x01d:      halt
0x020:      .pos 0x20
0x020: p:   irmovq $-1,%rdi        # procedure
0x02a:      ret
0x02b:      irmovq $1,%rax         # Should not be executed
0x035:      irmovq $2,%rcx         # Should not be executed
0x03f:      irmovq $3,%rdx         # Should not be executed
0x049:      irmovq $4,%rbx         # Should not be executed
0x100:      .pos 0x100
0x100:      Stack:                # Stack: Stack pointer
```

Stalling for Return

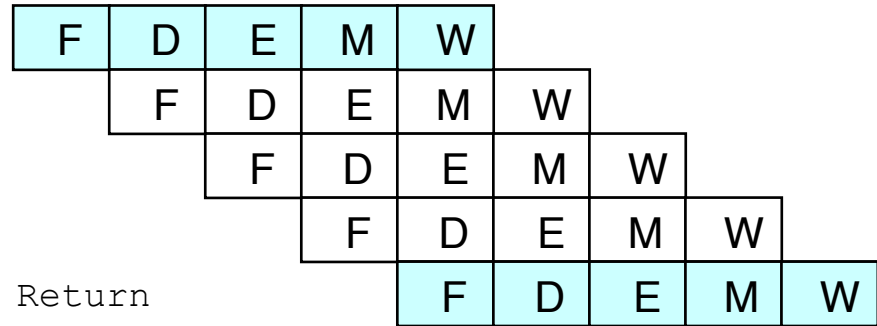
0x026: ret

nop

nop

nop

0x013: irmovq \$5,%rsi # Return

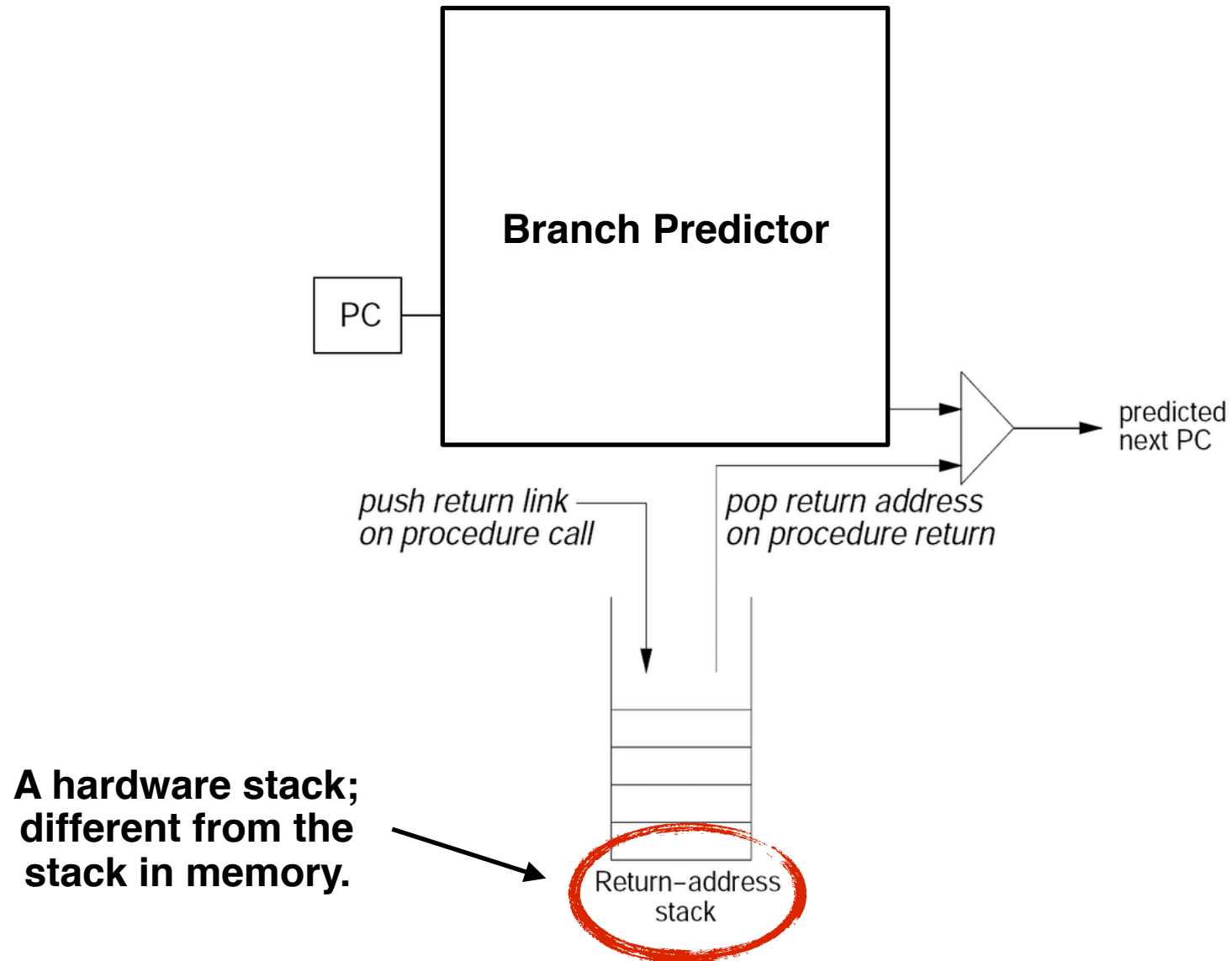


- **As `ret` passes through pipeline, stall at fetch stage**
 - While in decode, execute, and memory stage
- **Inject bubble into decode stage**
- **Release stall when reach write-back stage**

Return Address Stack (RAS)

- Stalling for return is silly since we know where exactly we need to jump to, except the jump target is retrieved later in the memory stage.
- Can we get that sooner? Where should we get it?

Return Address Stack (RAS)



Today: Making the Pipeline Really Work

- Control Dependencies

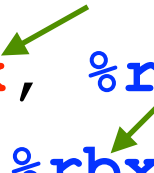
- Inserting Nops
- Stalling
- Delay Slots
- Branch Prediction

- Data Dependencies

- Inserting Nops
- Stalling
- Out-of-order execution

Data Dependencies

```
1      irmovq $50, %rax
2      addq   %rax, %rbx
3      mrmovq 100(%rbx), %rdx
```



- Result from one instruction used as operand for another
 - **Read-after-write (RAW)** dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
 - Get correct results
 - Minimize performance impact

A Subtle Data Dependency

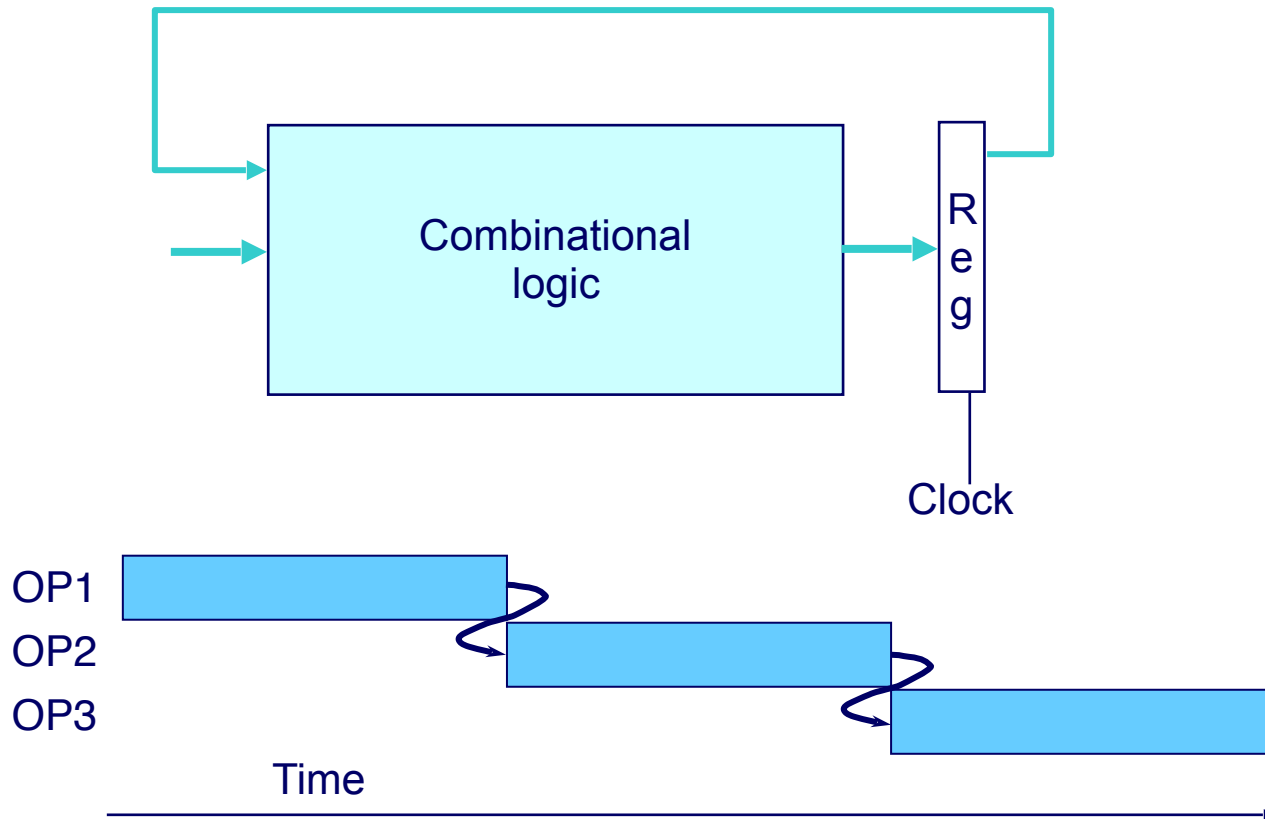
- Jump instruction example below:
 - `jne L1` determines whether `irmovq $1, %rax` should be executed
 - But `jne` doesn't know its outcome **until after its Execute stage**.

Why?

- There is a data dependency between `xorg` and `jne`. The “data” is the status flags.

		1	2	3	4	5	6	7	8	9
xorg %rax, %rax		F	D	E	M	W				
jne L1	# Not taken		F	D	E	M	W			
nop				F	D	E	M	W		
nop					F	D	E	M	W	
irmovq \$1, %rax	# Fall Through					F	D	E	M	W
L1: irmovq \$4, %rcx	# Target						F	D	E	M
L1: irmovq \$3, %rax	# Target + 1							F	D	E

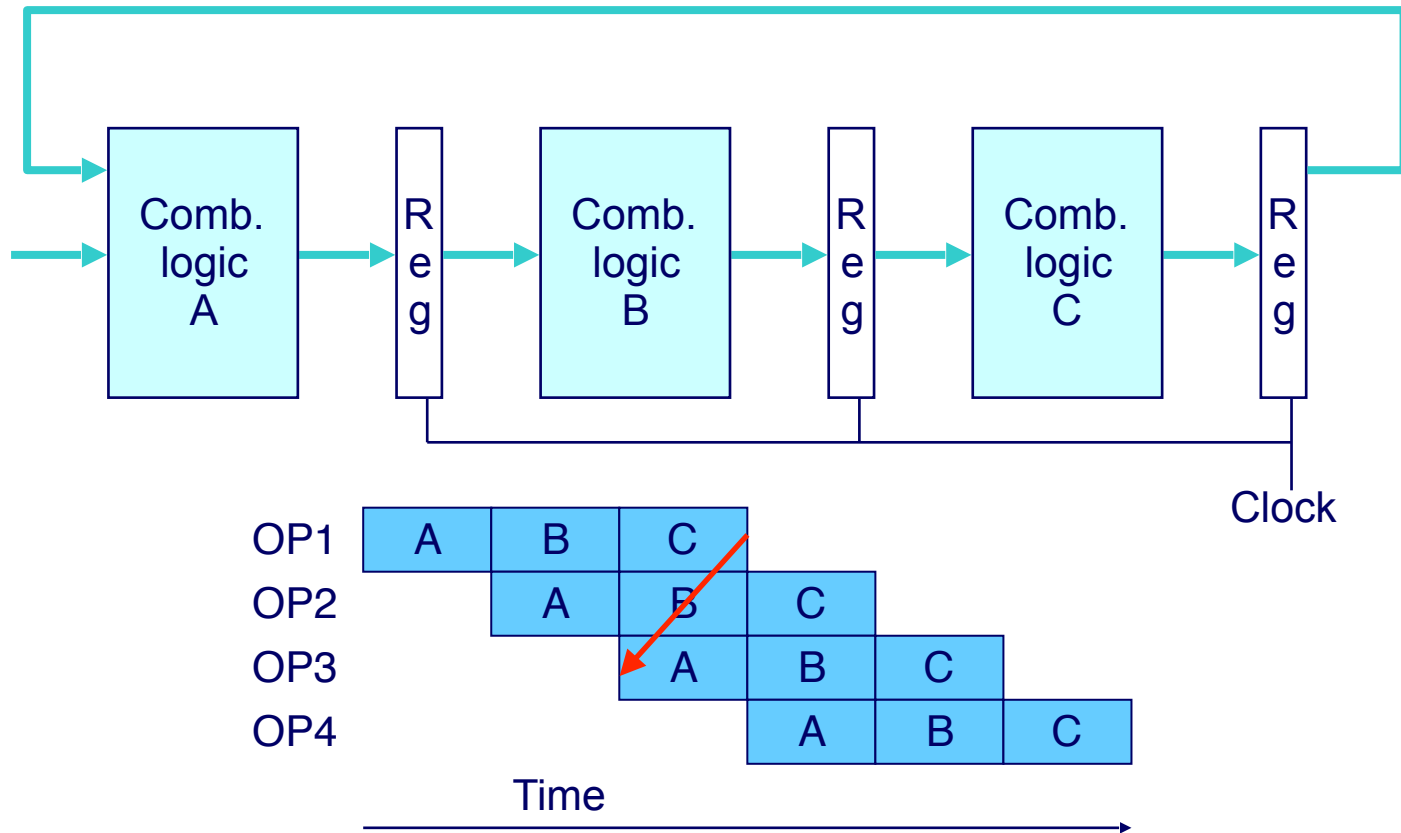
Data Dependencies in Single-Cycle Machines



In Single-Cycle Implementation:

- Each operation starts only after the previous operation finishes. Dependency always satisfied.

Data Dependencies in Pipeline Machines



Data Hazards happen when:

- Result does not feed back around in time for next operation

Data Dependencies: No Nop

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

0x014: addq %rdx,%rax

0x016: halt

1	2	3	4	5	6	7	8
F	D	E	M	W			
	F	D	E	M	W		
		F	D	E	M	W	
			F	D	E	M	W

Remember registers get
updated in the Write-back stage

addq reads wrong %rdx and %rax

Data Dependencies: 1 Nop

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

0x014: nop

0x015: addq %rdx,%rax

0x017: halt

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	E	M	W			
		F	D	E	M	W		
			F	D	E	M	W	
				F	D	E	M	W

addq still reads wrong %rdx and %rax

Data Dependencies: 2 Nop's

	1	2	3	4	5	6	7	8	9	10
0x000: irmovq \$10,%rdx	F	D	E	M	W					
0x00a: irmovq \$3,%rax		F	D	E	M	W				
0x014: nop			F	D	E	M	W			
0x015: nop				F	D	E	M	W		
0x016: addq %rdx,%rax					F	D	E	M	W	
0x018: halt						F	D	E	M	W

**addq reads the correct %rdx,
but %rax still wrong**

Data Dependencies: 3 Nop's

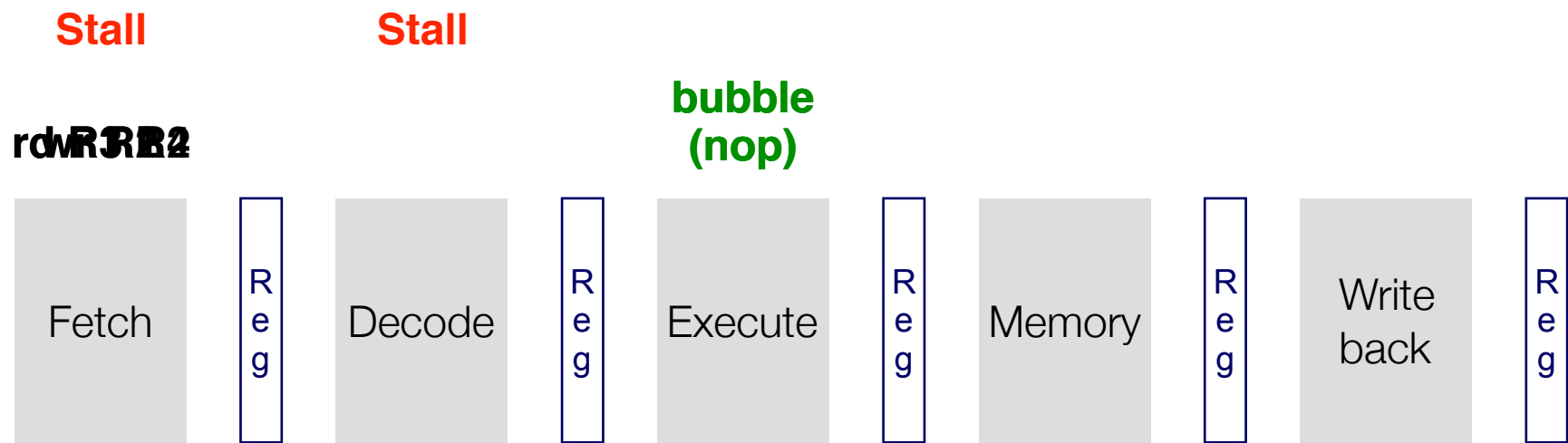
	1	2	3	4	5	6	7	8	9	10	11
0x000: irmovq \$10,%rdx	F	D	E	M	W						
0x00a: irmovq \$3,%rax		F	D	E	M	W					
0x014: nop			F	D	E	M	W				
0x015: nop				F	D	E	M	W			
0x016: nop					F	D	E	M	W		
0x017: addq %rdx,%rax						F	D	E	M	W	
0x019: halt							F	D	E	M	W

**addq reads the correct %rdx
and %rax**

Resolving Data Dependencies

- Software Mechanisms
 - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
- Hardware mechanisms
 - Stalling
 - Forwarding
 - Out-of-order execution

Hardware Generated Nops (Bubble and Stalling)



Detecting Stall Condition

- Using a “**scoreboard**”. Each register has a bit.
- Every instruction that writes to a register sets the bit.
- Every instruction that reads a register would have to check the bit first.
 - If the bit is set, then generate a bubble
 - Otherwise, free to go!!

Detecting Stall Condition

```
0x000: irmovq $10,%rdx
```

```
0x00a: irmovq $3,%rax
```

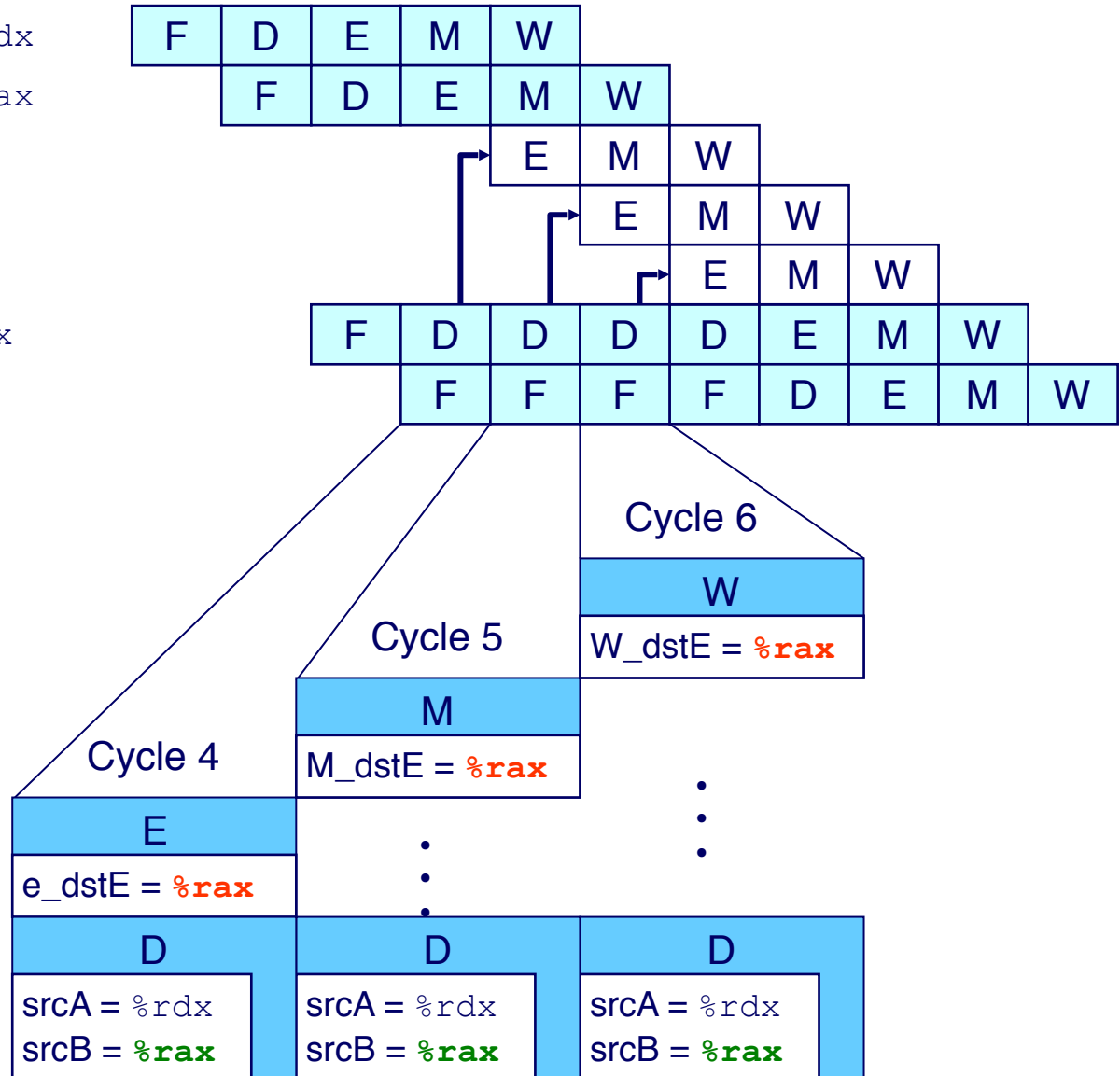
bubble

bubble

bubble

```
0x014: addq %rdx,%rax
```

```
0x016: halt
```



Data Forwarding

Naïve Pipeline

- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
- The decode stage can't start until the write-back stage finishes

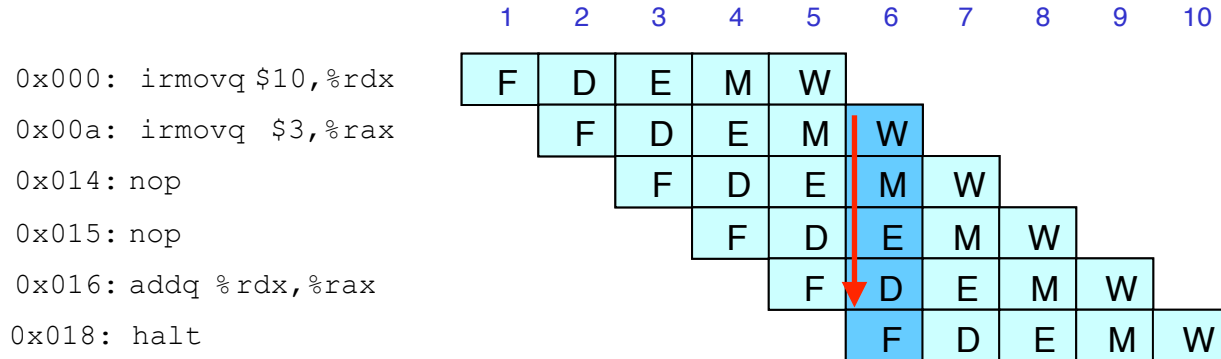
Observation

- Value generated in execute or memory stage

Trick

- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage

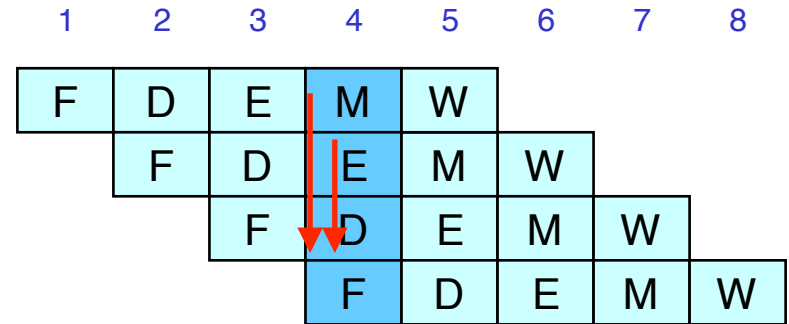
Data Forwarding Example



- `irmovq` writes `%rax` to the register file at the end of the write-back stage
- But the value of `%rax` is already available at the beginning of the write-back stage
- Forward `%rax` to the decode stage of `addq`.

Data Forwarding Example #2

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq  %rdx,%rax
0x016: halt
```



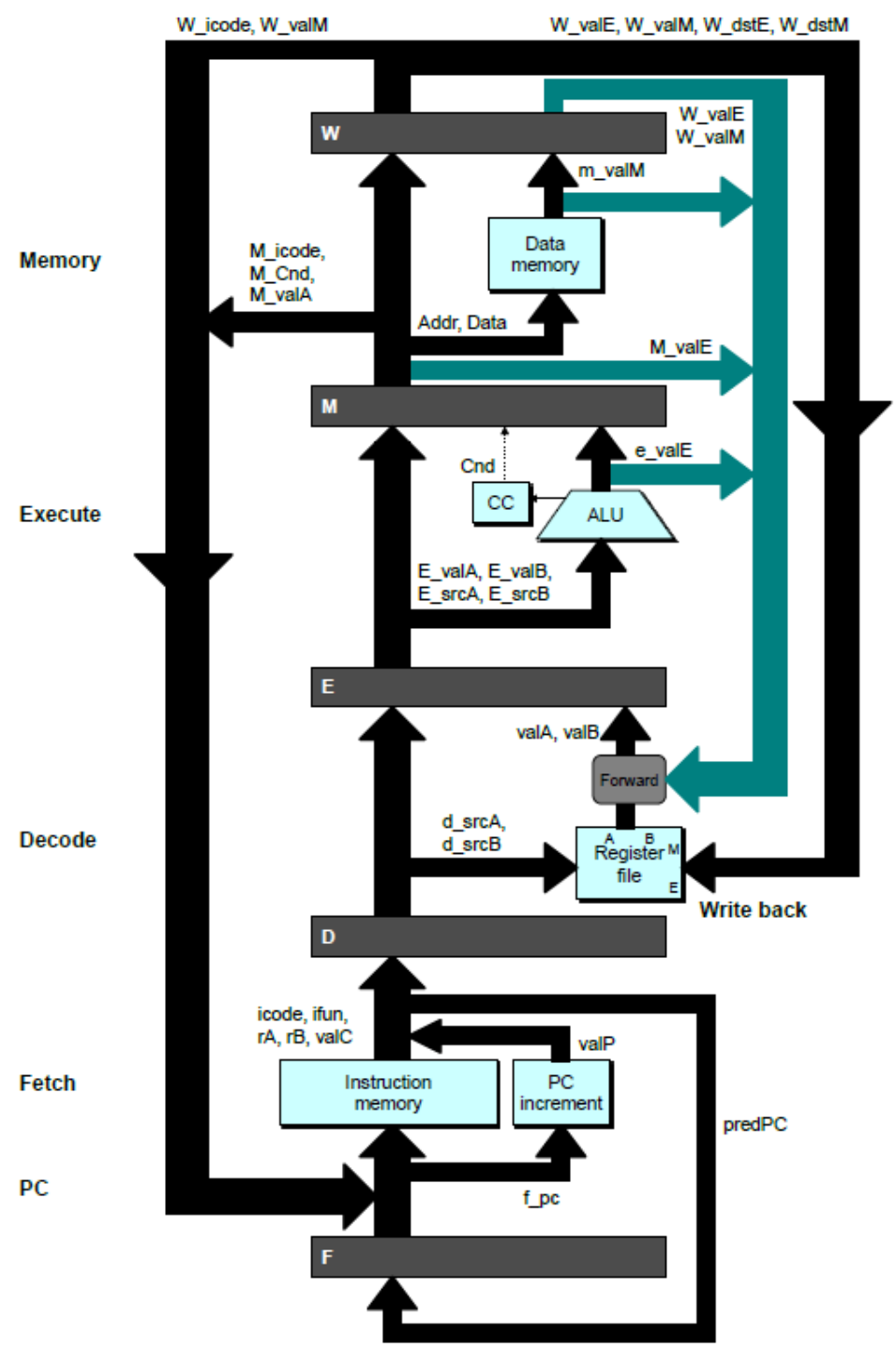
Register `%rdx`

- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

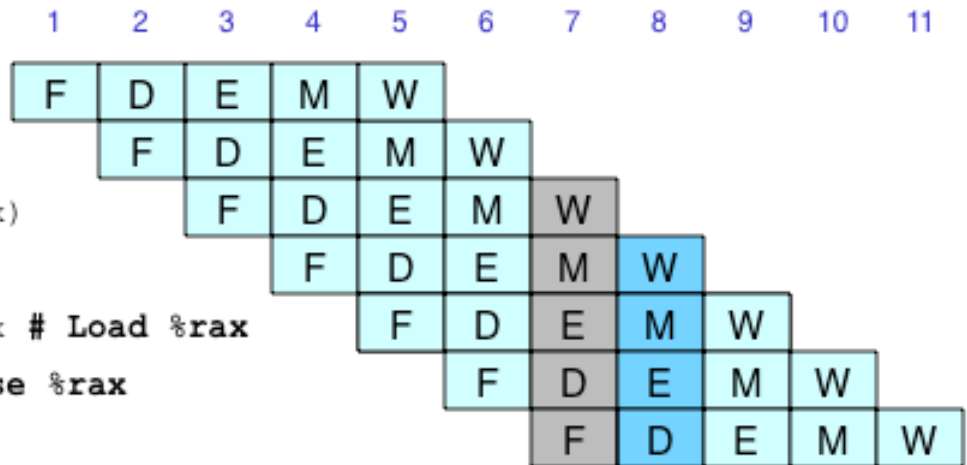
Hardware Design



Limitation of Forwarding

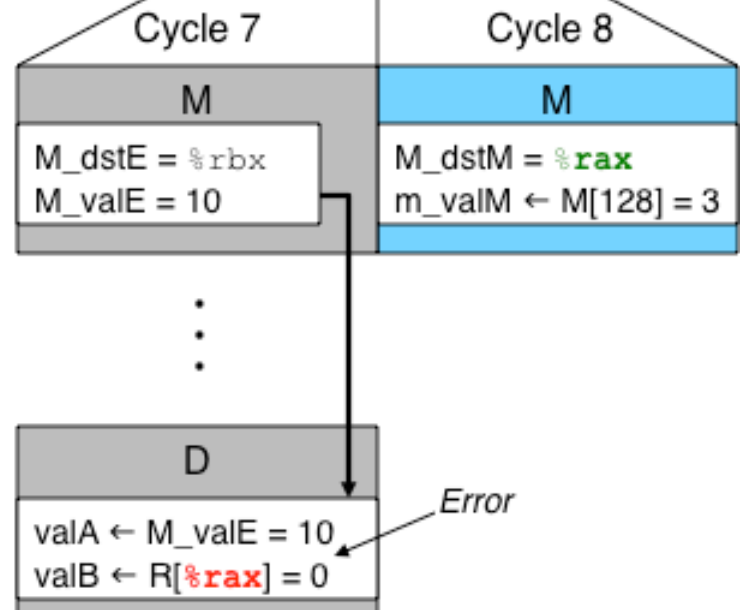
demo-luh.js

```
0x000: irmovq $128,%rdx
0x00a: irmovq $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```



Load-use dependency

- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8



Avoiding Load/Use Hazard

demo-luh.js

0x000: irmovq \$128,%rdx

0x00a: irmovq \$3,%rcx

0x014: rmmovq %rcx, 0(%rdx)

0x01e: irmovq \$10,%rbx

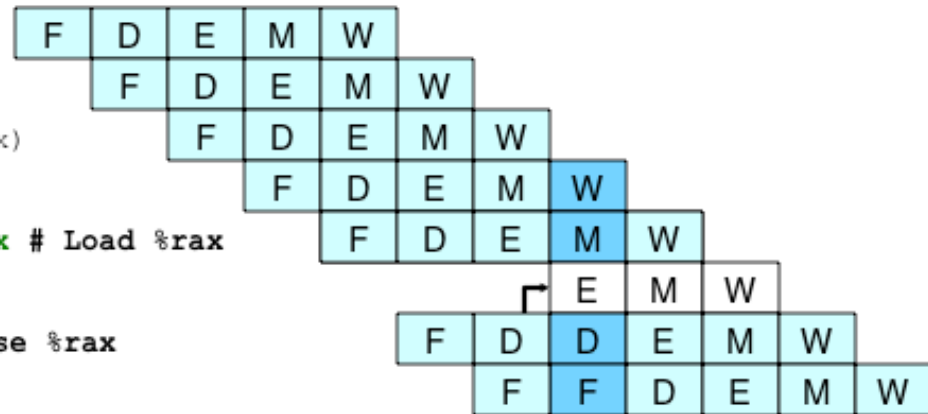
0x028: mrmovq 0(%rdx),%rax # Load %rax

bubble

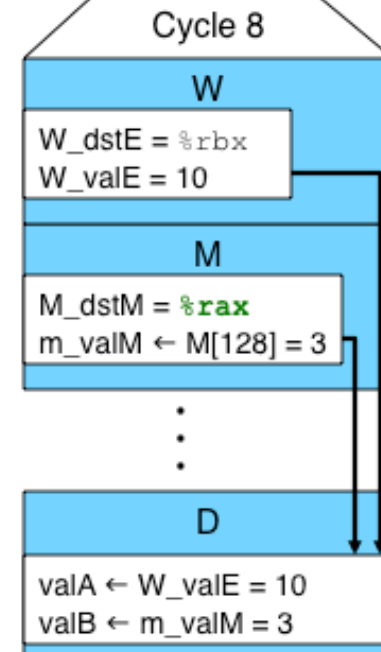
0x032: addq %rbx,%rax # Use %rax

0x034: halt

1 2 3 4 5 6 7 8 9 10 11 12



- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

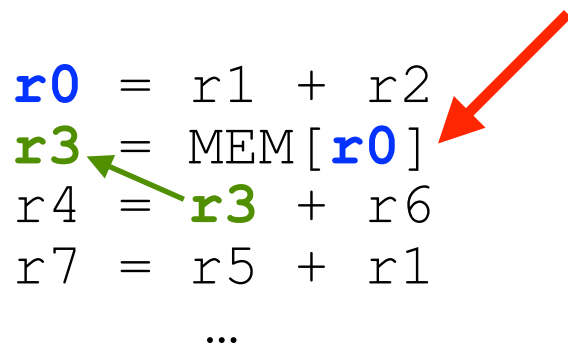


Out-of-order Execution

- Compiler could do this, but has limitations
- Generally done in hardware

Long-latency instruction.
Forces the pipeline to stall.

r0 = r1 + r2
r3 = MEM[**r0**]
r4 = **r3** + r6
r7 = r5 + r1
...



r0 = r1 + r2
r3 = MEM[**r0**]
r7 = r5 + r1
...
r4 = **r3** + r6

Out-of-order Execution

$r0 = r1 + r2$
 $r3 = \text{MEM}[r0]$
 $r4 = r3 + \mathbf{r6}$
 $\mathbf{r6} = r5 + r1$
...

Is this correct?



$r0 = r1 + r2$
 $r3 = \text{MEM}[r0]$
 $\mathbf{r6} = r5 + r1$
...
 $r4 = r3 + \mathbf{r6}$

$r0 = r1 + r2$
 $r3 = \text{MEM}[r0]$
 $\mathbf{r4} = r3 + r6$
 $\mathbf{r4} = r5 + r1$
...

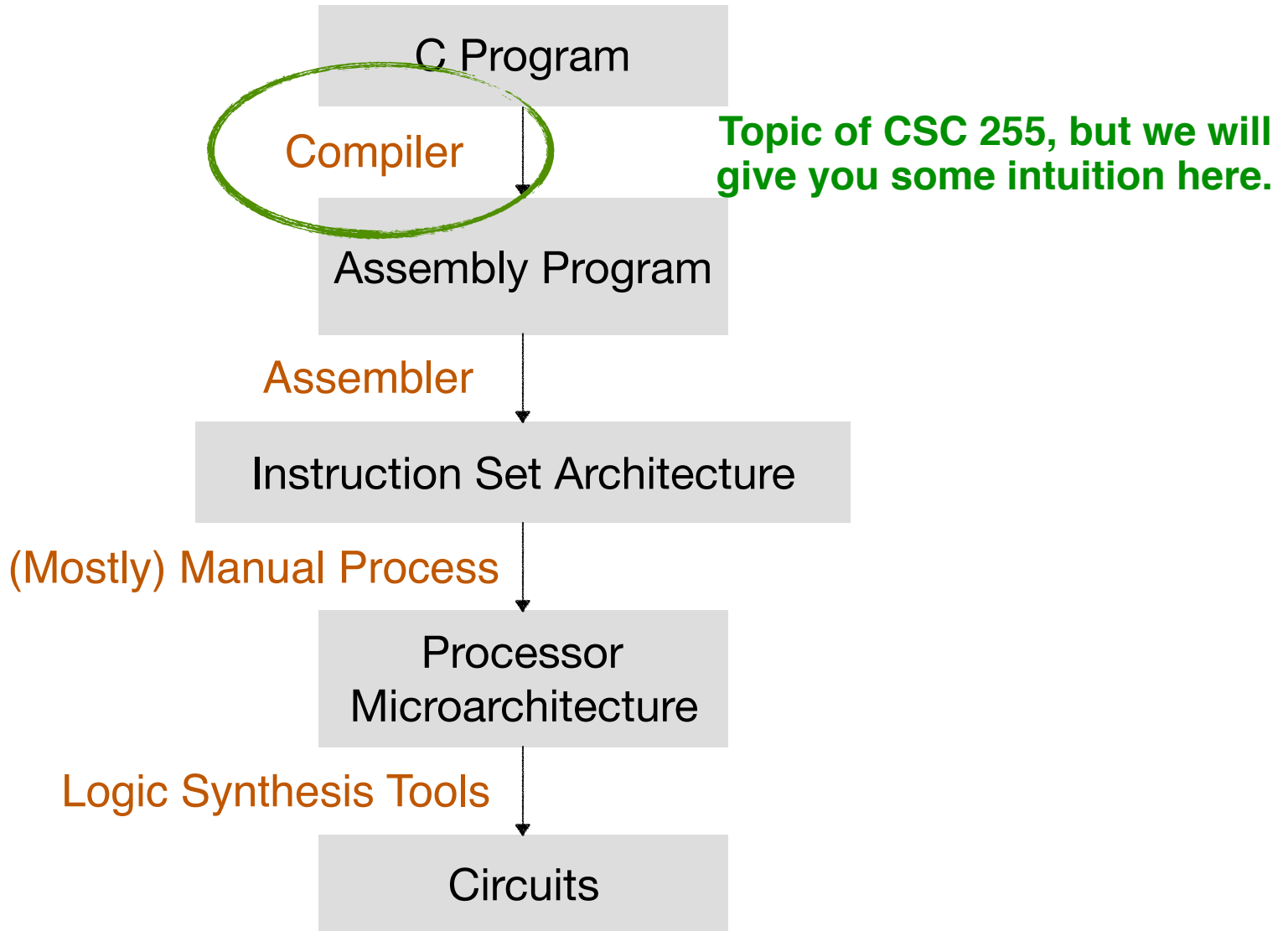
Is this correct?



$r0 = r1 + r2$
 $r3 = \text{MEM}[r0]$
 $\mathbf{r4} = r5 + r1$
...
 $\mathbf{r4} = r3 + r6$

“**Tomasolu Algorithm**” is the algorithm that is most widely implemented in modern hardware to get out-of-order execution right.

So far in 252...



Code Optimization Overview

- Three entities can optimize the program: programmer, compiler, and hardware
- The best thing to speed up a program is to pick a good algorithm. Compilers/hardware can't do that in general.
 - Quicksort: $O(n \log n) = K * n * \log(n)$
 - Bubblesort: $O(n^2) = K * n^2$
- Algorithm choice decides overall complexity (big O), compiler/hardware decides the constant factor in the big O notation
- Compiler and hardware implementations decide the K.
- Programmers can write code that makes it easier to compiler and hardware to improve performance.

Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
 - Code motion/precomputation
 - Strength reduction
 - Sharing of common subexpressions
- Optimization Blockers
 - Procedure calls
 - Memory aliasing
- Exploit Hardware Microarchitecture

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor
- Code Motion
 - Reduce frequency with which computation performed
 - If it will always produce same result
 - Especially moving code out of loop

```
void set_row(double *a, double *b,  
            long i, long n)  
{  
    long j;  
    for (j = 0; j < n; j++)  
        a[n*i+j] = b[j];  
}
```



```
long j;  
int ni = n*i;  
for (j = 0; j < n; j++)  
    a[ni+j] = b[j];
```

Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b,
             long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
long j;
int ni = n*i;
for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```



```
set_row:
    testq    %rcx, %rcx                # Test n
    jle      .L1                      # If 0, goto done
    imulq    %rcx, %rdx                # ni = n*i
    leaq     (%rdi,%rdx,8), %rdx       # rowp = A + ni*8
    movl     $0, %eax                 # j = 0
.L3:
    movsd    (%rsi,%rax,8), %xmm0      # t = b[j]
    movsd    %xmm0, (%rdx,%rax,8)      # M[A+ni*8 + j*8] = t
    addq     $1, %rax                 # j++
    cmpq     %rcx, %rax               # j:n
    jne      .L3                      # if !=, goto loop
.L1:
    rep ; ret                          # done:
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
 - $16 * x \quad \rightarrow \quad x \ll 4$
 - Depends on cost of multiply or divide instruction
 - On Intel Nehalem, integer multiply requires 3 CPU cycles. Division takes even more cycles. Shift can generally be done in 1 cycle.
- Use the `leaq` instruction

```
long m12(long x)
{
    return x*12;
}
```

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax           # return t<<2
```

Common Subexpression Elimination

- Reuse portions of expressions
- GCC will do this with `-O1`

3 multiplications: $i*n$, $(i-1)*n$, $(i+1)*n$

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j ];
down =  val[(i+1)*n + j ];
left =  val[i*n      + j-1];
right = val[i*n      + j+1];
sum = up + down + left + right;
```



```
leaq    1(%rsi), %rax    # i+1
leaq    -1(%rsi), %r8    # i-1
imulq   %rcx, %rsi      # i*n
imulq   %rcx, %rax      # (i+1)*n
imulq   %rcx, %r8       # (i-1)*n
addq    %rdx, %rsi      # i*n+j
addq    %rdx, %rax      # (i+1)*n+j
addq    %rdx, %r8       # (i-1)*n+j
```

1 multiplication: $i*n$

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```



```
imulq   %rcx, %rsi      # i*n
addq    %rdx, %rsi      # i*n+j
movq    %rsi, %rax      # i*n+j
subq    %rcx, %rax      # i*n+j-n
leaq    (%rsi,%rcx), %rcx # i*n+j+n
```

Today: Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
 - Code motion/precomputation
 - Strength reduction
 - Sharing of common subexpressions
- Optimization Blockers
 - Procedure calls
 - Memory aliasing
- Exploit Hardware Microarchitecture

Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Calling Strlen

```
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- **Strlen performance**
 - Has to scan the entire length of a string, looking for null character.
 - $O(N)$ complexity
- **Overall performance**
 - N calls to strlen
 - Overall $O(N^2)$ performance

Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```
void lower(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Optimization Blocker: Procedure Calls

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

size_t total_lencount = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    total_lencount += length;
    return length;
}
```

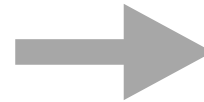
Why couldn't compiler move `strlen` out of loop?

- Procedure may have side effects, e.g., alters global state each time called
- Function may not return same value for given arguments

Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
 - Assume the worst case, weak optimizations near them
 - There are interprocedural optimizations (IPO), but they are expensive
 - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.
- Remedies:
 - Use of inline functions
 - Do your own code motion

```
inline void swap(int *m, int *n) {  
    int tmp = *m;  
    *m = *n;  
    *n = tmp;  
}  
  
void foo () {  
    swap(&x, &y);  
}
```



```
void foo () {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
{ 0, 1, 2,
  4, 8, 16,
  32, 64, 128};
```

Value of b:

init: [x, x, x]

i = 0: [3, x, x]

i = 1: [3, 28, x]

i = 2: [3, 28, 224]

A Potential Optimization

```
/* Sum rows of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Every iteration updates
memory location `b[i]`.
Memory accesses are
slow, so...



```
double val = 0;
for (j = 0; j < n; j++)
    val += a[i*n + j];
b[i] = val;
```

Every iteration updates `val`,
which could stay in register.
Update memory only once.

Why can't a compiler perform this optimization?

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

b →

```
double a[9] =
{ 0, 1, 2,
  3, 22, 16,
  32, 64, 128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

Value of b:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

Optimization Blocker: Memory Aliasing

- Aliasing
 - Two different memory references (array elements or pointers) specify the same memory location
 - Easy to have in C
 - Since C allows address/pointer arithmetic
 - Direct access to storage structures
 - Get in habit of introducing local variables
 - Accumulating within loops
 - Your way of telling compiler not to check for aliasing