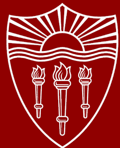
CS356

Review for Final Exam

Marco Paolieri (paolieri@usc.edu)

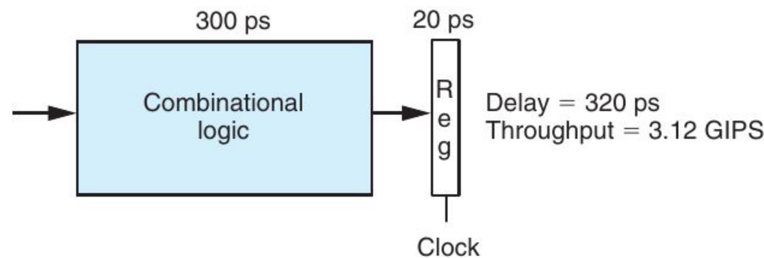
Illustrations from CS:APP3e textbook



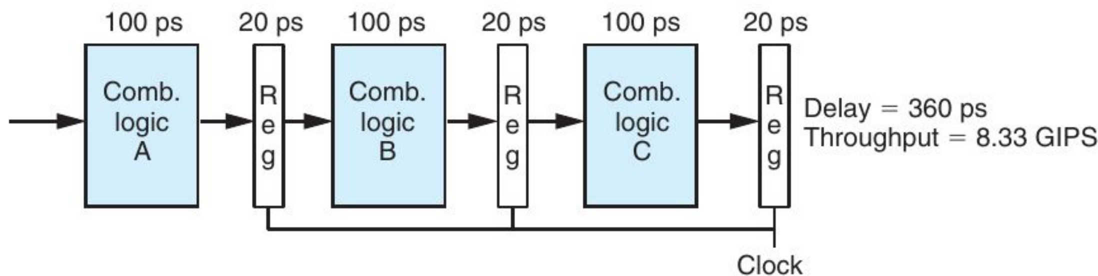
USC University of Southern California

Processor Organization

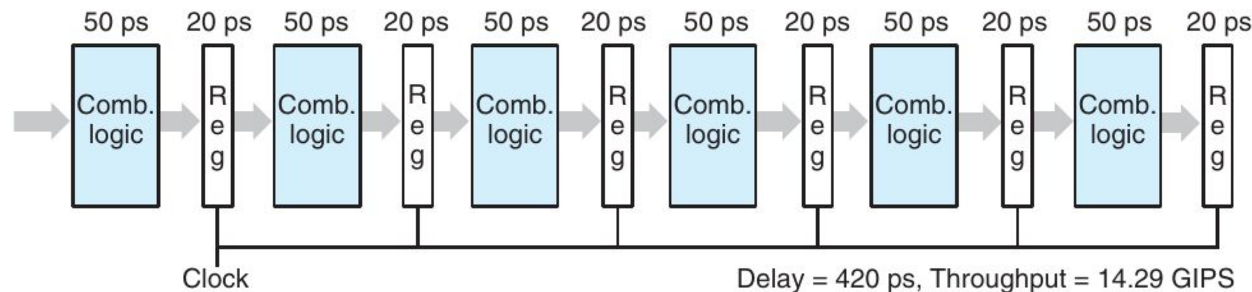
Pipeline: Computing Throughput and Delay



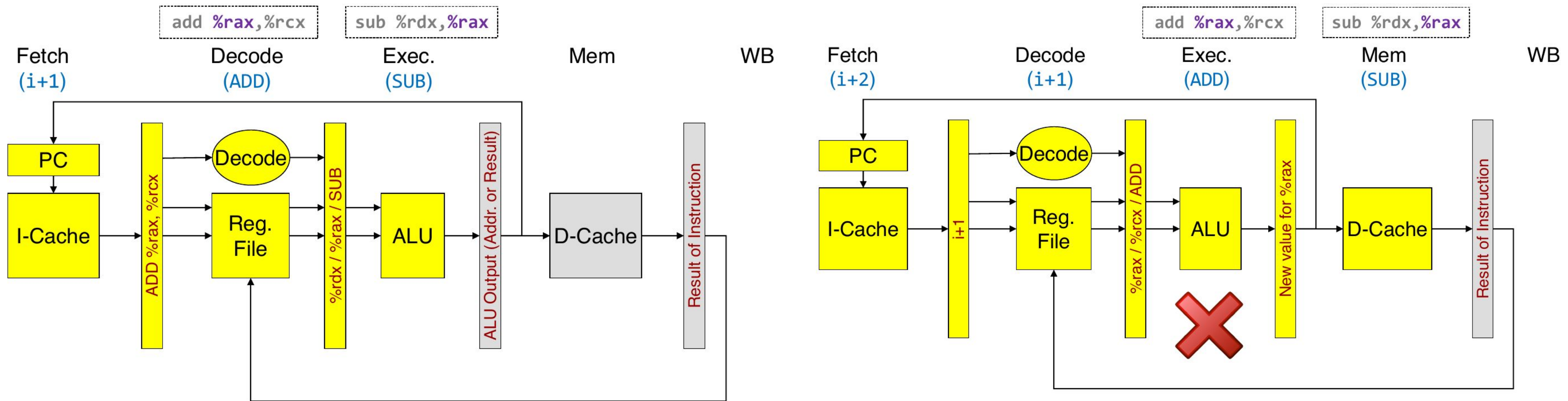
n	clock (ps)	tput (GIPS)
1	320	3.125
2	170	5.882
3	120	8.333
4	95	10.526
5	80	12.500
6	70	14.286




```
clock = 300/n + 20
tput  = 1/clock
delay = n*clock
```

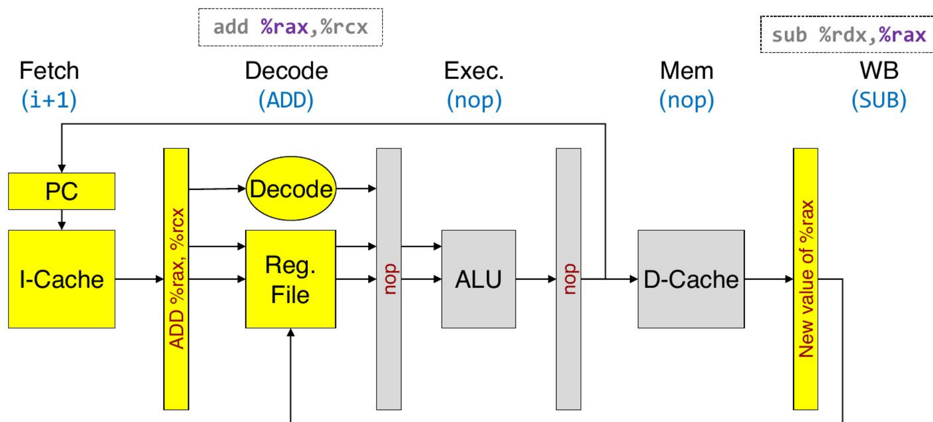


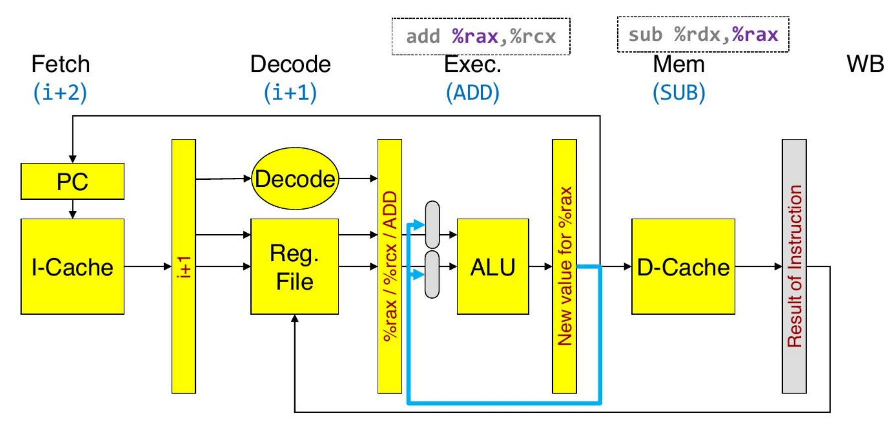
Pipeline Hazards: Stalling and Forwarding



Stalling

Forwarding



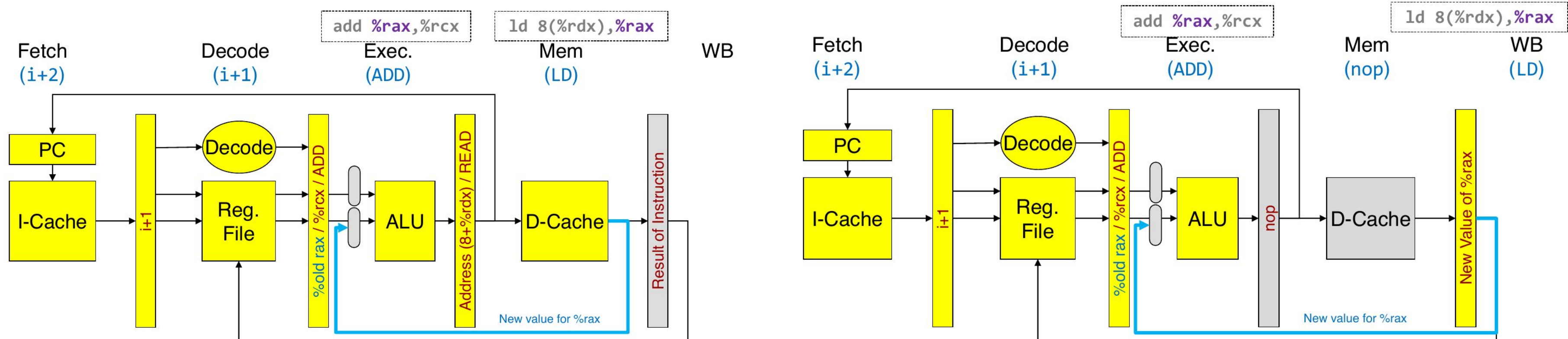


Structural Hazard: Load for next instruction

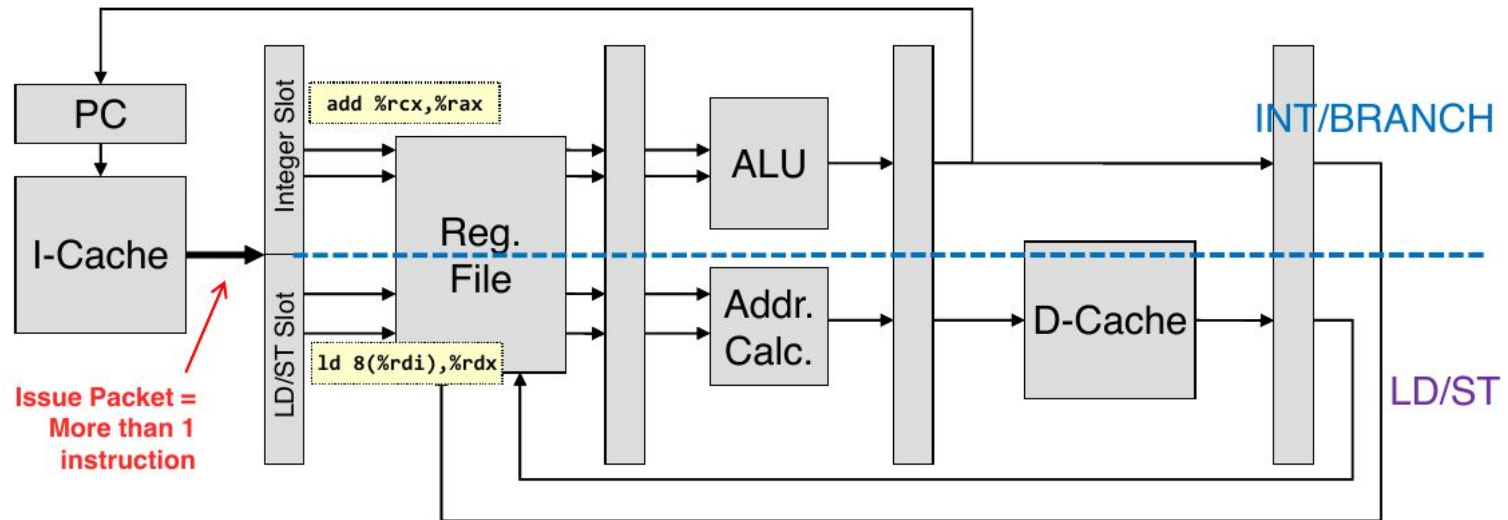
```
ld 8(%rdx), %rax
add %rax, %rcx
```

While `ld` is saving `%rdx` into a register (phase M), `add` is already using its input to compute a result in phase E.

- Forwarding is not enough! We need the output of D-Cache, not the input...
- Use **stalling and forwarding together**.
  - `add` is stalled by 1 phase
  - `ld` passes back the new value of `%rdx` during phase WB



2-way Very Large Instruction Word Machine



- No forwarding between instructions of an “issue packet”
- Full forwarding to instructions behind in the pipeline
- Stall 1 cycle at “load for next instruction”

2-way VLIW Machine: Scheduling Example

```
void incr5(int *a, int n) {
    for (; n != 0; n--, a++)
        *a += 5;
}
```

```
incr5:
.L1:
      ld     0(%rdi), %r9
      // nop required here
      add    $5, %r9
      st     %r9, 0(%rdi)
      add    $4, %rdi
      add    $-1, %esi
      jne    $0, %esi, .L1
```

Unoptimized Schedule (no gain wrt single pipeline)

```
=== INTEGER SLOT ===          ===  LD/ST SLOT  ===
                              ld 0(%rdi), %r9
add $-1, %esi
add $5, %r9
                              st %r9, 0(%rdi)
add $4, %rdi
jne $0, %esi, .L1
```

Optimized Schedule (move up increase of si/di)

```
=== INTEGER SLOT ===          ===  LD/ST SLOT  ===
add $-1, %esi                 ld 0(%rdi), %r9
add $4, %rdi
add $5, %r9
jne $0, %esi, .L1             st %r9, -4(%rdi)
```

From 6/6 = 1 instructions per cycle to 6/4 = 1.5

Loop Unrolling

Sometimes we don't have enough instruction for parallel pipelines.

Idea: copy body *k* times and iterate only *n/k* times (assume *n* multiple of *k*)

- Different copies of body can run in parallel.

```
void incr5(int *a, int n) {
    for (; n != 0; n-= 4, a+=4) {
        *a += 5;
        *(a+1) += 5;
        *(a+2) += 5;
        *(a+3) += 5;
    }
}
```

```
incr5:
.L1:
0      ld     0(%rdi), %r9
0      add    $5, %r9
0      st     %r9, 0(%rdi)
1      ld     4(%rdi), %r9
1      add    $5, %r9
1      st     %r9, 4(%rdi)
2      ld     8(%rdi), %r9
2      add    $5, %r9
2      st     %r9, 8(%rdi)
3      ld     12(%rdi), %r9
3      add    $5, %r9
3      st     %r9, 12(%rdi)
       add    $16, %rdi
       add    $-4, %esi
       jne    $0, %esi, .L1
```

```
old-incr5:
.L1:
0      ld     0(%rdi), %r9
0      add    $5, %r9
0      st     %r9, 0(%rdi)
       add    $4, %rdi
       add    $-1, %esi
       jne    $0, %esi, .L1
```

Still can't run in parallel: all copies use the register **%r9**

⇒ **Read-After-Write (RAW)**

⇒ **Register renaming**

Loop Unrolling and Register Renaming

```
incr5:
.L1:
0     ld     0(%rdi), %r9
0     add    $5, %r9
0     st     %r9, 0(%rdi)
1     ld     4(%rdi), %r10
1     add    $5, %r10
1     st     %r10, 4(%rdi)
2     ld     8(%rdi), %r11
2     add    $5, %r11
2     st     %r11, 8(%rdi)
3     ld     12(%rdi), %r12
3     add    $5, %r12
3     st     %r12, 12(%rdi)
      add    $16, %rdi
      add    $-4, %esi
      jne    $0, %esi, .L1
```

Optimized Schedule

```
=== INTEGER SLOT ===        ===  LD/ST SLOT  ===
                            ld 0(%rdi), %r9
add $-4, %esi               ld 4(%rdi), %r10
add $5, %r9                 ld 8(%rdi), %r11
add $5, %r10                ld 12(%rdi), %r12
add $5, %r11                st %r9, 0(%rdi)
add $5, %r12                st %r10, 4(%rdi)
add $16, %rdi               st %r11, 8(%rdi)
jne $0, %esi, .L1           st %r12, -4(%rdi)
```

IPC = 15/8

Notice: We exploit independence of loop bodies.

Exercise: 2-way VLIW Scheduling

```
void f1(int *A, int *B, int N) {
    for( ; N != 0; A--, B--, N--) {
       int temp = *A;
        *A = temp + *B + 9;
        *B = temp;
    }
}
```

```
.L1:
    ld (%rdi),%eax    ; load temp=*A
    ld (%rsi),%ebx    ; load *B
    add %eax,%ebx     ; add temp+*B
    add $9,%ebx       ; add 9
    st %ebx,(%rdi)    ; store *A
    st %eax,(%rsi)    ; store *B
    add $-4,%rdi      ; dec. A ptr.
    add $-4,%rsi      ; dec. B ptr.
    add $-1,%rdx
    jne $0,%rdx,.L1 ; loop
```

Unoptimized Schedule

=== INTEGER SLOT ===	=== LD/ST SLOT ===
	ld (%rdi),%eax
	ld (%rsi),%ebx
add %eax,%ebx	
add $9,%ebx	
	st %ebx,(%rdi)
	st %eax,(%rsi)
add $-4,%rdi	
add $-4,%rsi	
add $-1,%rdx	
jne $0,%rdx,.L1	

You can move or modify code, but cannot apply loop unrolling or register renaming.

Solution: 2-way VLIW Scheduling

Unoptimized Schedule

```
=== INTEGER SLOT ===     ===  LD/ST SLOT  ===
                         ld (%rdi),%eax
                         ld (%rsi),%ebx

add %eax,%ebx
add $9,%ebx
                         st %ebx,(%rdi)
                         st %eax,(%rsi)
add $-4,%rdi
add $-4,%rsi
add $-1,%rdx
jne $0,%rdx,.L1
```

Move Up and Modify Offsets

```
=== INTEGER SLOT ===     ===  LD/ST SLOT  ===
add $-4,%rdi             ld (%rsi),%eax
add $-4,%rsi             ld (%rsi),%ebx
add $-1,%rdx             st %eax,4(%rsi)
add %eax,%ebx
add $9,%ebx
jne $0,%rdx,.L1          st %ebx,4(%rdi)
```

IPC = 10 instructions / 6 clocks = 1.67

Note: intermediate instruction between load into **%ebx** and its use by **add**

Next Exercise: Unroll the loop once (2 total iterations) with register renaming.

Unrolling the loop with register renaming

Loop Unrolling / Register Renaming

```
.L1:
    ld (%rdi),%eax      ; load temp=*A
    ld (%rsi),%ebx      ; load *B
    add %eax,%ebx       ; add temp+*B
    add $9,%ebx         ; add 9
    st %ebx,(%rdi)      ; store *A
    st %eax,(%rsi)      ; store *B
    ld -4(%rdi),%r8d    ; 2nd iter
    ld -4(%rsi),%r9d    ;
    add %r8d,%r9d       ;
    add $9,%r9d         ;
    st %r9d,-4(%rdi)    ;
    st %r8d,-4(%rsi)    ;
    add $-8,%rdi        ; dec. A ptr.
    add $-8,%rsi        ; dec. B ptr.
    add $-2,%rdx
    jne $0,%rdx,.L1     ; loop
```

Move Up and Modify Offsets

```
=== INTEGER SLOT ===    ===  LD/ST SLOT  ===
add $-8,%rdi            ld (%rdi),%eax
add $-8,%rsi            ld (%rsi),%ebx
add $-2,%rdx            ld 4(%rdi),%r8d
add %eax,%ebx           ld 4(%rsi),%r9d
add $9,%ebx             st %eax,8(%rsi)
add %r8d,%r9d           st %ebx,8(%rdi)
add $9,%r9d             st %r8d,4(%rsi)
jne $0,%rdx,.L1         st %r9d,4(%rdi)
```

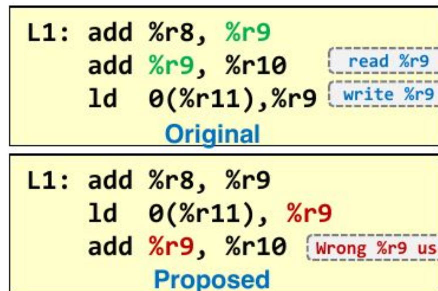
Increased Offset

Reversed %rsi / %rdi

IPC = 16 instructions / 8 clocks = 2

Note: intermediate instructions between loads and uses of a register.

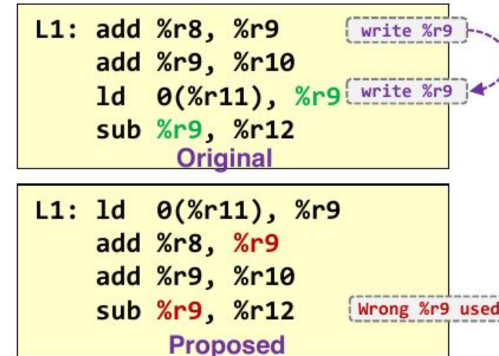
Exercise: Solve WAR/WAW hazards



LD instruction is REALLY independent (only needs %r11).

Could LW instruction be moved between the 2 add's?

Not as is, WAR hazard



Could LD instruction be run in parallel with or before first add?

Not as is, WAW hazard

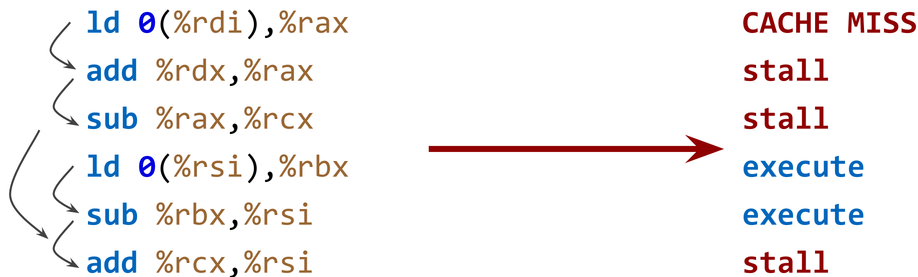
Solve WAR/WAW hazards of the following code through renaming.

```
ld 0(%rdi),%rax
add %rcx,%rax
sub %rbx,%rcx
ld 0(%rsi),%rbx
sub %rsi,%rbx
add %rbx,%rbx
```

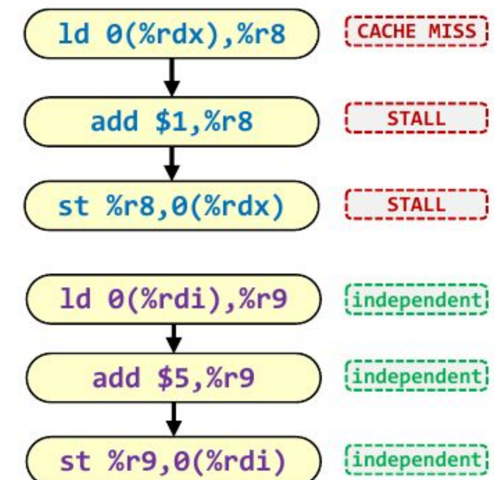
```
ld 0(%rdi),%rax
add %rcx,%rax
sub %rbx,%rcx
ld 0(%rsi),%r8
sub %rsi,%r8
add %r8,%r8
```

Exercise: Out-of-order Dynamic Scheduling

In the following code, assume the first `ld` instruction stalls due to a cache miss. Assuming an out-of-order, dynamically scheduled processor (that performs **automatic register renaming**), which instructions would be allowed to execute (i.e. are independent) and which instructions would need to stall due to the `ld` miss?



Similar example from class:



Caches

Cache Organization: K-way set-associative

1 valid bit per line
t tag bits per line
$B = 2^b$ bytes per cache block

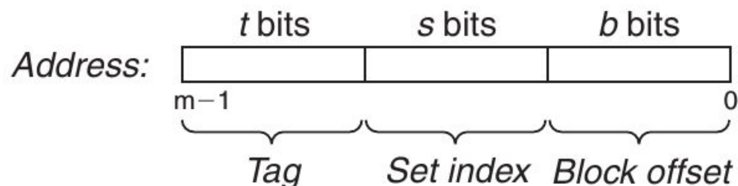
Set 0:
Valid | Tag | 0 | 1 | · · · | $B-1$
Valid | Tag | 0 | 1 | · · · | $B-1$
K lines per set

Set 1:
Valid | Tag | 0 | 1 | · · · | $B-1$
Valid | Tag | 0 | 1 | · · · | $B-1$

$S = 2^s$ sets

Set $S-1$:
Valid | Tag | 0 | 1 | · · · | $B-1$
Valid | Tag | 0 | 1 | · · · | $B-1$

Cache size: $C = B \times K \times S$ data bytes



Memory: addresses of m bits
⇒ $M = 2^m$ memory locations

Cache:

- $S = 2^s$ **cache sets**
- Each set has K **lines**
- Each line has: **data block** of $B = 2^b$ bytes, **valid bit**, $t = m - (s+b)$ **tag bits**

How to check if the word at an address is in the cache?

Exercise: Cache Size and Address

Problem

A processor has a **32-bit** memory address space. The memory is broken into blocks of **32 bytes** each. The cache is capable of storing **16 kB**.

- How many blocks can the cache store?
- Break the address into tag, set, byte offset for **direct-mapping cache**.
- Break the address into tag, set, byte offset for a **4-way set-associative cache**.

Solution

- 16 kB / 32 bytes per block = 512 blocks.
- Direct-mapping: 18-bit tag (rest), 9-bit set address, 5-bit block offset.
- 4-way set-associative: each set has 4 lines, so there are 512 / 4 = 128 sets.
    - 20-bit tag (rest)
    - 7-bit set address
    - 5-bit block offset

Exercise: Cache Size and Address

Problem

A processor has a **36-bit** memory address space. The memory is broken into blocks of **64 bytes** each. The cache is capable of storing **1 MB**.

- How many blocks can the cache store?
- Break the address into tag, set, byte offset for **direct-mapping cache**.
- Break the address into tag, set, byte offset for a **8-way set-associative cache**.

Solution

- 1 MB / 64 bytes per block = 2**(20-6) = 16k blocks.
- Direct-mapping: 16-bit tag (rest), 14-bit set address, 6-bit block offset.
- 8-way set-associative: each set has 8 lines, so there are 16k / 8 = 2k sets
    - 19-bit tag (rest)
    - 11-bit set address
    - 6-bit block offset

Exercise: Direct-Mapping Performance

You are asked to optimize a cache capable of storing **8 bytes** total for the given references. There are three direct-mapped cache designs possible by varying the block size:

- C1 has one-byte blocks,
- C2 has two-byte blocks, and
- C3 has four-byte blocks.

In terms of miss rate, which cache design is best?

If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design? (Every access, hit or miss, requires an access to the cache.)

Trace (LSB)

```
  1 0000 0001
134 1000 0110
212 1101 0100
  1 0000 0001
135 1000 0111
213 1101 0101
162 1010 0010
161 1010 0001
  2 0000 0010
 44 0010 1100
 41 0010 1001
221 1101 1101
```

Solution: Direct-Mapping Performance

Address breakdown

- C1 has no block offset, 3-bit set address
- C2 has 1-bit block offset, 2-bit set address
- C3 has 2-bit block offset, 1-bit set address

How to run a trace: extract set address (3, 2, 1 bits) from LSB; on miss, load (1, 2, 4) bytes.

Running C3:

- **Get 1: miss.** Put bytes 0-3 in bucket 0.
- **Get 134: miss.** Put 132-135 in bucket 1.
- **Get 212: miss.** Put 212-215 in bucket 1.
- **Get 1: hit.**
- **Get 135: miss.** Put 132-135 in bucket 1.
- **Get 212: miss.** Put 212-215 in bucket 1.
- **Get 162: miss.** Put 160-163 in bucket 0.
- **Get 161: hit.**

Trace

MEM	LSB	C1	C2	C3
1	0000 0001	1m	0m	0m
134	1000 0110	6m	3m	1m
212	1101 0100	4m	2m	1m
1	0000 0001	1h	0h	0h
135	1000 0111	7m	3h	1m
213	1101 0101	5m	2h	1m
162	1010 0010	2m	1m	0m
161	1010 0001	1m	0m	0h
2	0000 0010	2m	1m	0m
44	0010 1100	4m	2m	1m
41	0010 1001	1m	0m	0m
221	1101 1101	5m	2m	1m
m_rate:		11/12	9/12	10/12

Solution: Performance

Average Access Time = (**Hit Time**) + (**Miss Rate**) × (**Miss Penalty**)

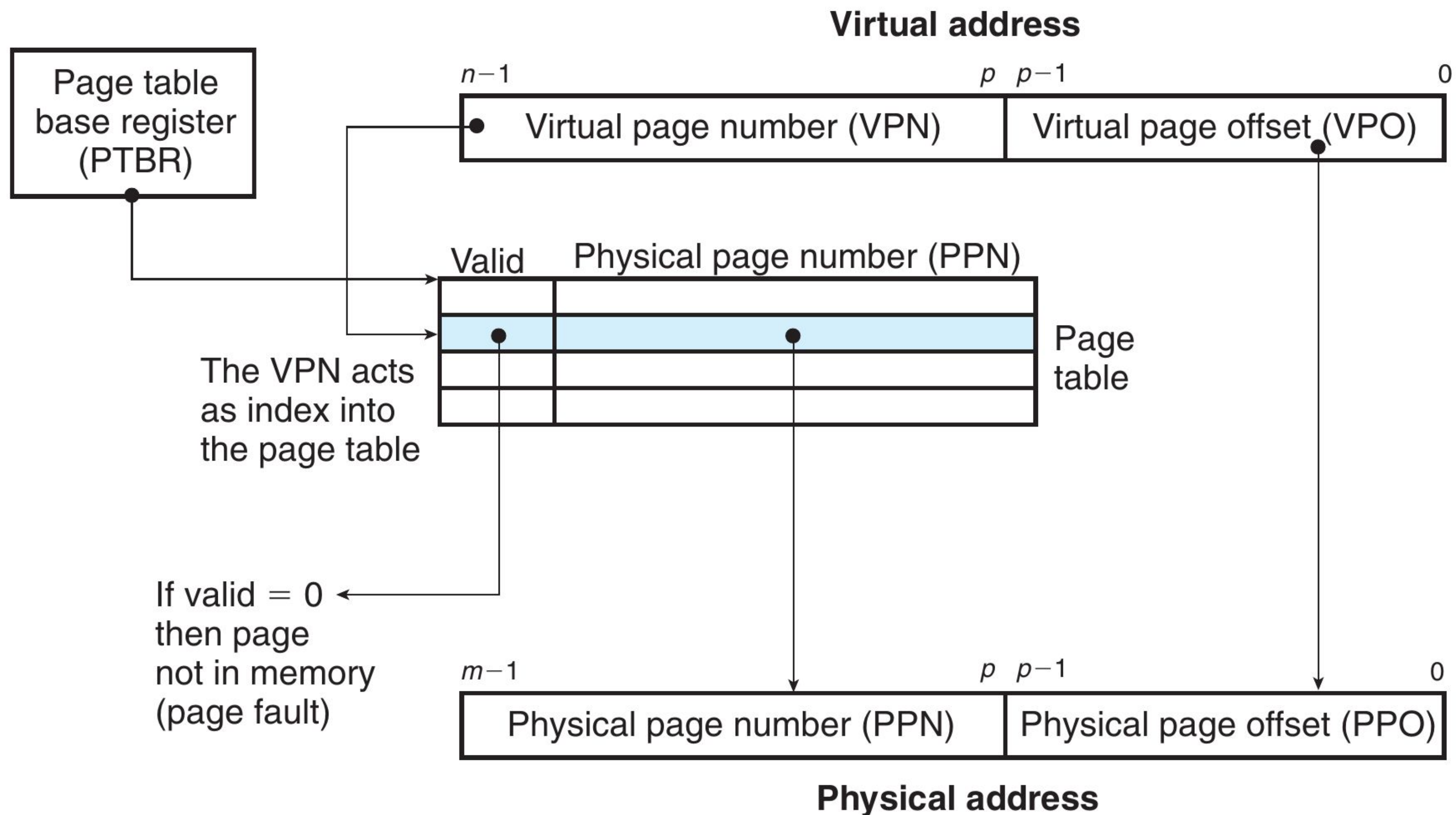
In terms of miss rate, C2 is best.

If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design? (Every access, hit or miss, requires an access to the cache.)

- For C1, access time = $2 + 11/12 \times 25 = 24.92$ cycles
- For C2, access time = $3 + 9/12 \times 25 = 21.75$ cycles
- For C3, access time = $5 + 10/12 \times 25 = 25.83$ cycles

Virtual Memory

Single-Level Page Table: **PTBR**[VPN] | VPO



Example: 32 bit virtual address, 4 kB pages ⇒ 20 bit VPN, **1M page table entries**

- Only 1 GB of physical memory ⇒ 18 bit PPN (translated address is 00...)

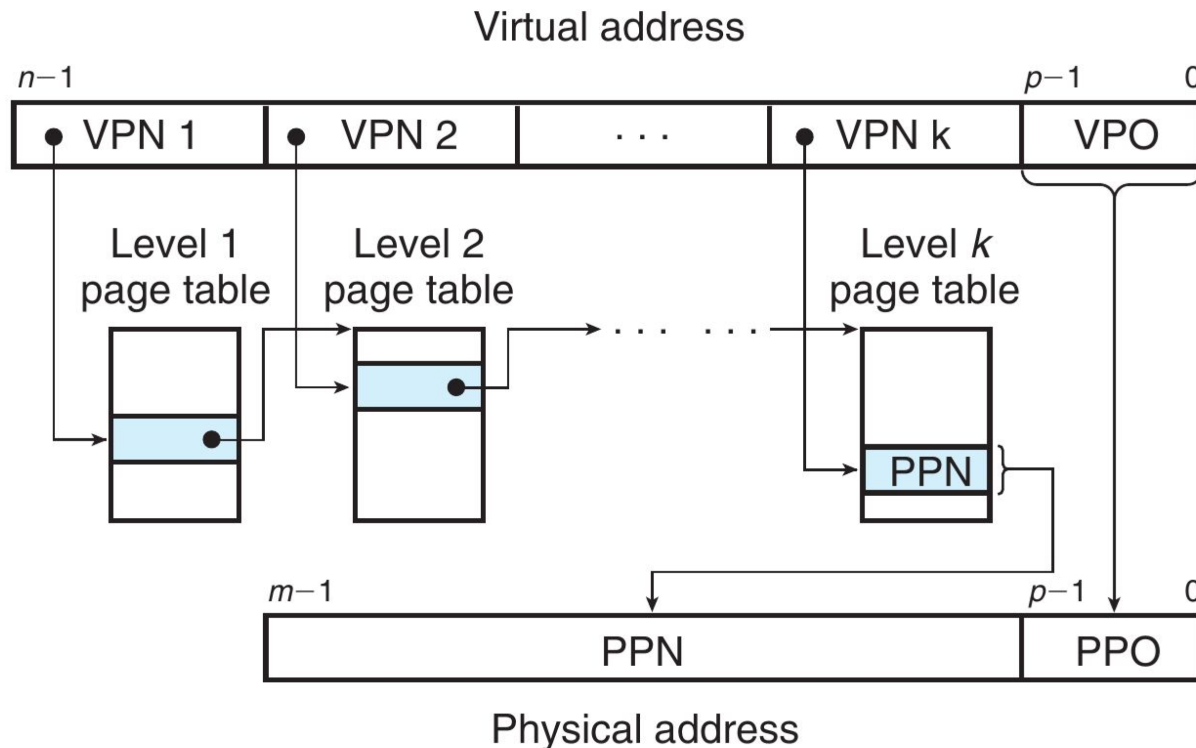
Example: Single-Level Page Table

Index	Valid	PPN
0	0	0x0E
1	1	**0x1E**
2	1	0x16
3	1	**0x06**
4	0	0x0B
5	1	0x1F
6	0	0x15
7	0	0x0A

8-bit virtual addresses, 10-bit physical addresses, 32-byte pages

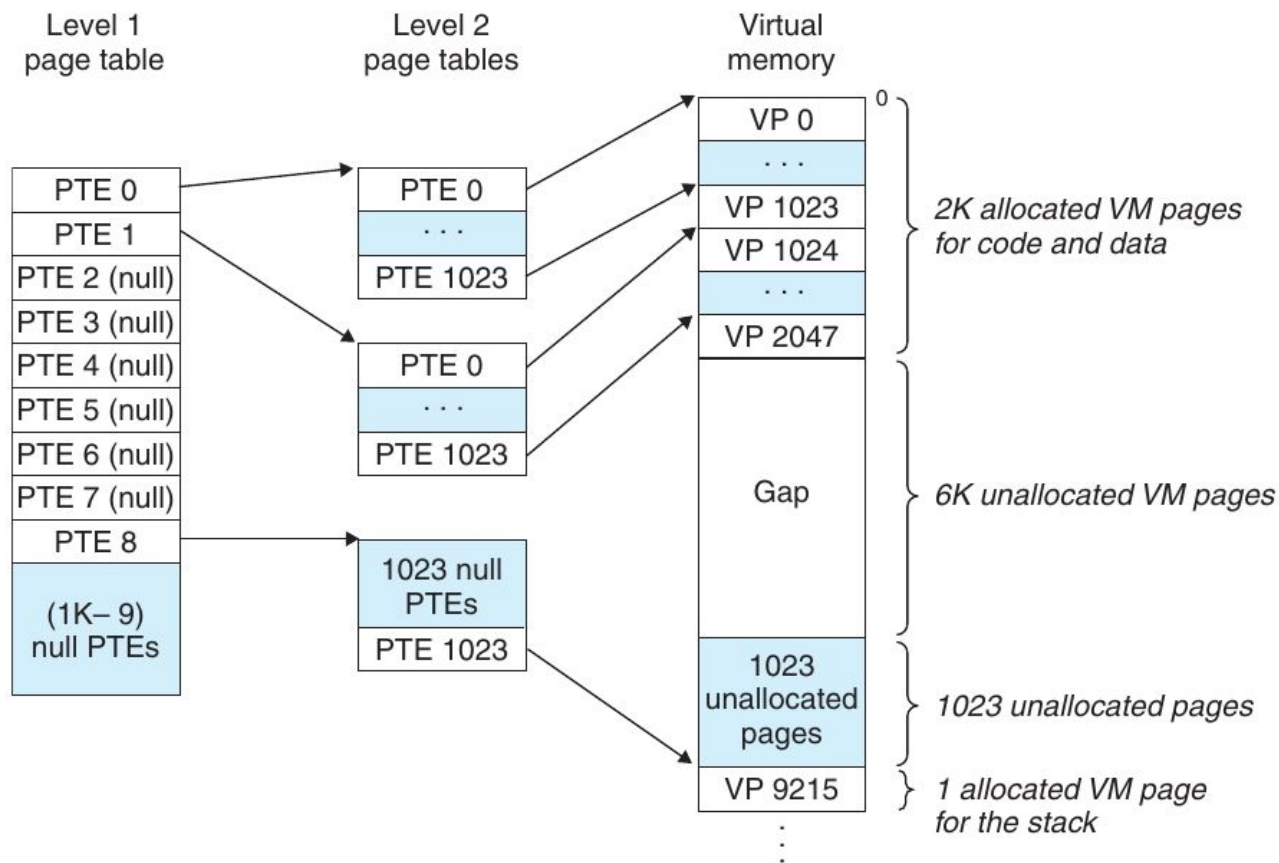
- Physical address of virtual address **0x2D**? **00101101 => 0 0011 1100 1101** (1, E)
- Physical address of virtual address **0x7A**? **01111010 => 0 0000 1101 1010**
- Physical address of virtual address **0xEF**? **11101111 => not valid**
- Physical address of virtual address **0xA8**? **10101000 => 0 0011 1110 1000**

Multi-Level Page Table: More indirections



The virtual address space can be very large for a single process.
⇒ Most of the page table entries are not used
⇒ **Idea:** use a **page directory** where entries point to next-level tables (if present)
⇒ Each level contains base of next table (if present), **last level contains PPN**

Multi-Level Page Table: Space savings



Drawback: more memory accesses, more latency...

Problem: Three-Level Page Table

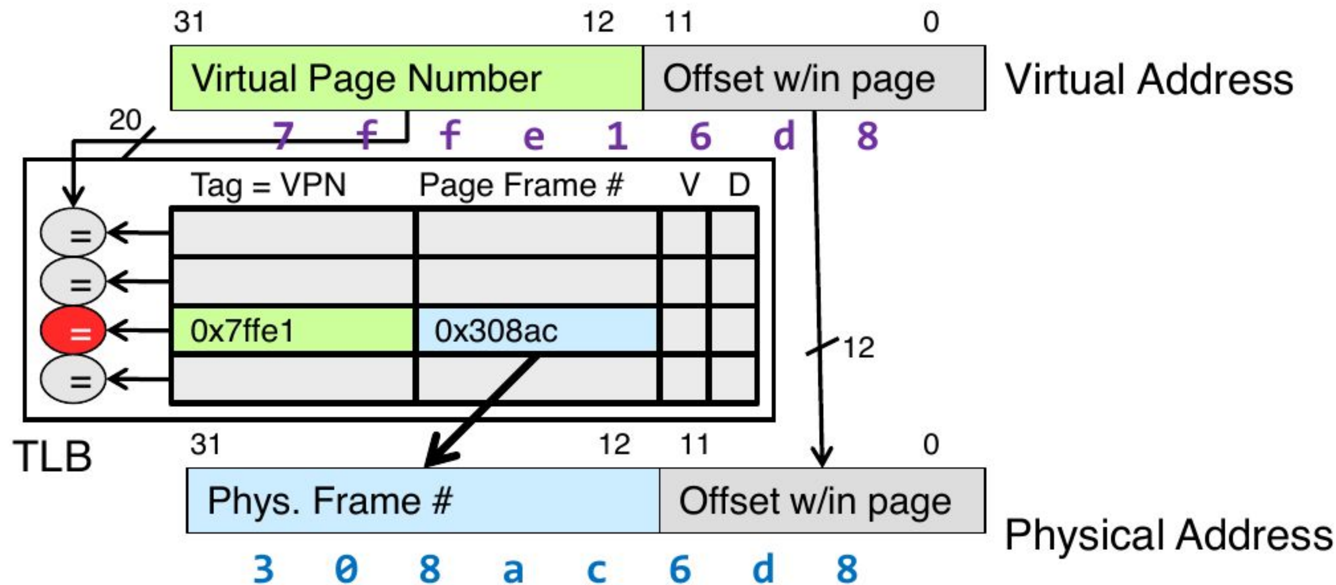
Consider a 3-level VM system with:

- 36-bit physical address space
- 32-bit virtual address space
- 4 kB pages
- Page tables implemented as look-up tables
- 256 entries for page directory
- 64 entries in second-level page table

Find out:

- The layout of virtual addresses (1st / 2nd / 3rd table offset, page offset)
- The number of entries in third-level page table
- The size of each page table (assume 4 bytes for each entry)
- Minimum size of entries of third page table?

Translation Lookaside Buffer



A *k*-level page table requires *k* memory accesses in the worse case.

Idea: cache address mappings inside the CPU (10 ns hit time).

- **VPN is the cache tag, PPN is the entire cache block**
- High degree of associativity (4-way or fully-associative: low miss rate)
- Usually smaller than data cache (fast lookup, low hit time)

Average Access Time = (**Hit Time**) + (**Miss Rate**) × (**Miss Penalty**)

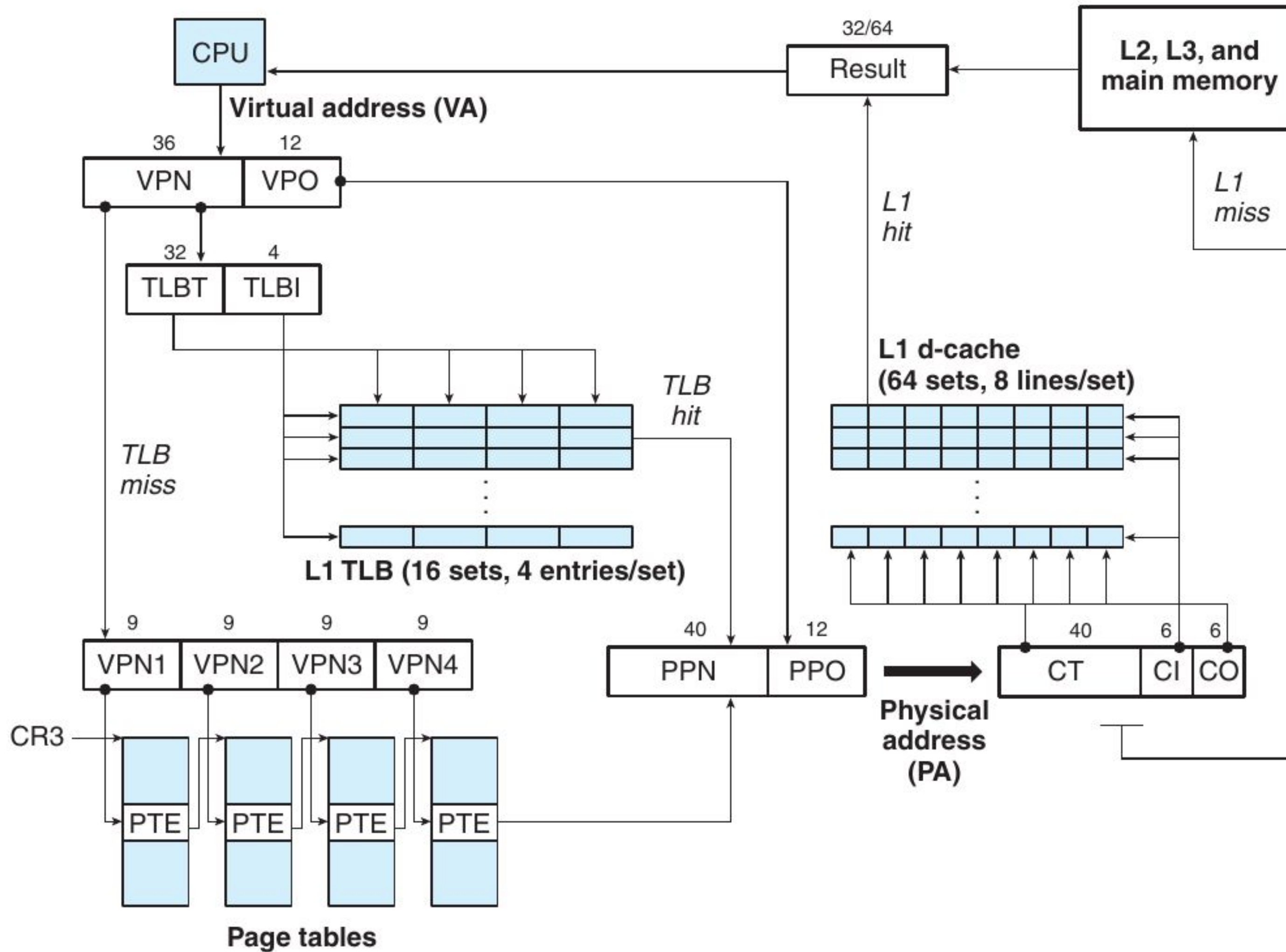
Example: 2-way set-associative TLB

Index	Valid	Tag	PPN
0	1	0x13	**0x30**
	0	0x34	0x58
1	0	0x1F	0x80
	1	0x2A	0x72
2	1	**0x1F**	0x95
	0	0x20	0xAA
3	1	0x3F	0x20
	0	0x3E	0xFF

16-bit virtual and physical addresses, 256-byte pages

- Physical address of virtual address **`0x7E85`** `== 0111 1110 1000 0101`?
- Virtual address of physical address **`0x3020`** `== 0011 0000 0010 0000`?

Intel Core i7: TLB and translation before L1



Assembly

The stack grows toward lower addresses

Pushing a value

- Decrement stack pointer (SP)
- Store new value at address pointed by SP

Example: **pushq** `%rax` is equivalent to

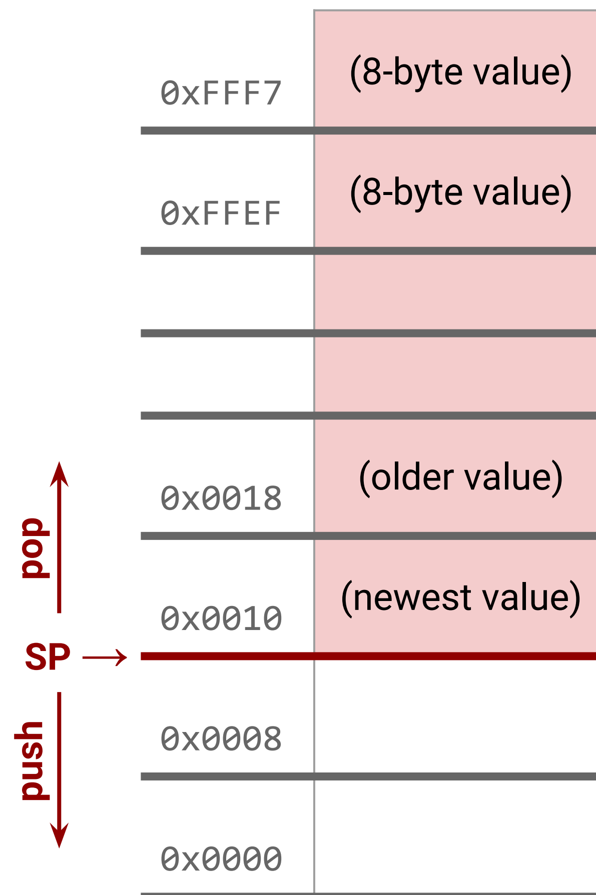
```
subq $8, %rsp
movq %rax, (%rsp)
```

Popping a value

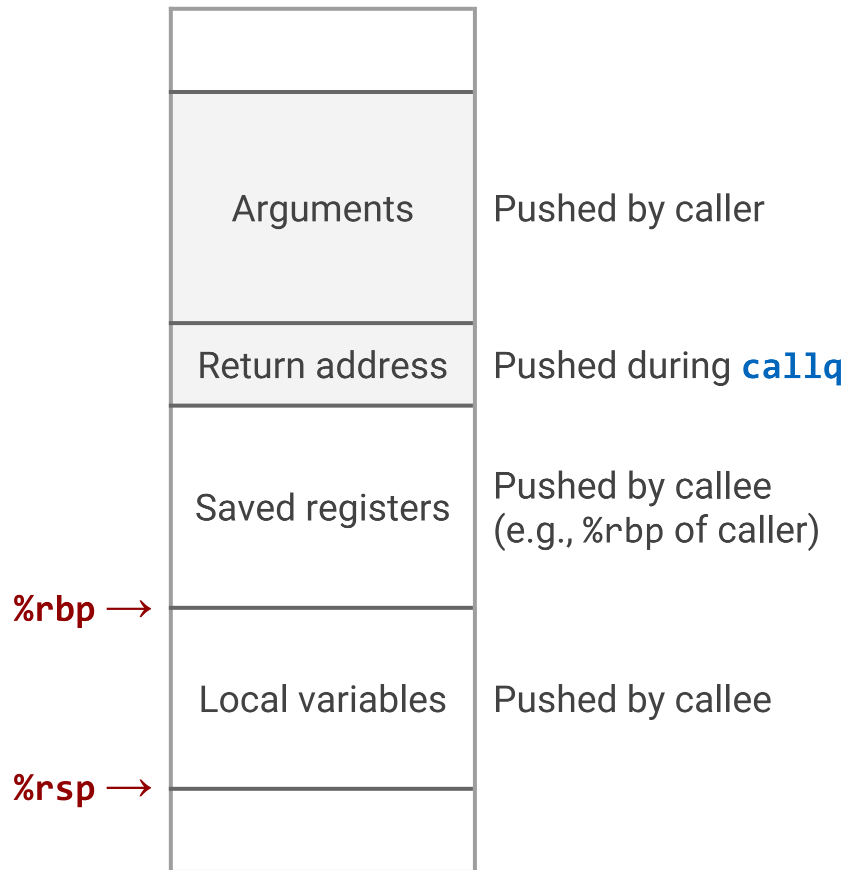
- Read value at address pointed by SP
- Increment SP

Example: **popq** `%rax` is equivalent to

```
movq (%rsp), %rax
addq $8, %rsp
```



Stack frames



Pushed by caller

Pushed during `callq`

Pushed by callee
(e.g., `%rbp` of caller)

Pushed by callee

Passing Parameters

Conventions

- First six integer/pointer arguments on **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`**, **`%r9`**
- Additional ones are pushed on the stack in **reverse order** as **8-byte words**.
- The caller must also **remove** parameters from stack after the call.
- Parameters **may be modified** by the called function.

Accessing stack parameters

- At the beginning of a function, `%rsp` points to the return address.
- Stack parameters can be addressed as: `8(%rsp)`, `16(%rsp)`, …

It is common practice to:

- Backup the initial value of `%rbp` (used by the caller): `pushq %rbp`
- Write `%rsp` (the current stack pointer) into `%rbp`: `movq %rsp, %rbp`
- Use `%rbp` to access parameters on the stack: `16(%rbp)` is the 7th param
- Restore the previous `%rbp` value at the end of the function: `popq %rbp`

(GCC optimizations avoid this use of `%rbp`, allowing its use as general register.)

Return Values and Registers

Return Values

- Integers or pointers: store return value in `%rax`
- Floating point: store return value in a floating-point register

Registers

- The caller must assume that `%rax`, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` to `%r11` may be changed by the callee (scratch registers / **caller-save**)
- Arithmetic flags are not preserved by function calls.
- The caller can assume that `%rbx`, `%rbp`, `%rsp`, and `%r12` to `%r15` will not change during function call.
    - The callee must save and restore them if necessary (**callee-save**).

Local Variables

When to use stack

Local variables must be allocated on the stack when:

- There are not enough registers.
- The address operator "&" is applied to a local variable.
- The variable is an array or a structure.

To allocate (uninitialized) local variables on the stack: `subq $16, %rsp`

Conventions

- Local variables can be allocated using **any size** (e.g., 1 byte for a char)
- They must be aligned at an address that is a **multiple of their size**.
- The stack pointer **`%rsp` must be a multiple of 16 before function calls**.
- The frame pointer `%rbp` is never changed after prologue / before epilogue.
- Local variables must be allocated immediately after callee-save registers.

Putting it all together

1. **The caller prepares and starts the call**
   - Push `%rax`, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` to `%r11` if required after call
   - Save arguments on `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` or into the stack
   - Execute **`callq`** (which pushes `%rip` and jumps to subroutine)
2. **The callee prepares for execution**
   - Push `%rbx`, `%rbp`, and `%r12` to `%r15` if modified during execution.
   - Decrement `%rsp` and allocate local variables on the stack.
3. **The callee runs** (possibly, invoking other functions)
4. **The callee restores the state and returns**
   - Increment `%rsp` to deallocate local variables from the stack.
   - Pop `%rbx`, `%rbp`, `%rsp`, and `%r12` to `%r15` (if pushed)
   - Execute **`retq`** (stores the return address into `%rip`)
5. **The caller restores the state**
   - Increment `%rsp` to deallocate arguments from stack.
   - Pop saved registers from stack.

Passing Control: Disassembling

```c
#include <stdio.h>

int sum(int x, int y, int *z) {
    return x + y + *z;
}

int main() {
    int z = 10;
    printf("%d\n", sum(1, 5, &z));
    return 0;
}
```

```
sum:
        addl   %esi, %edi
        movl   %edi, %eax
        addl   (%rdx), %eax
        ret
.LC0:
        .string     "%d\n"
```

```
main:
        subq   $24, %rsp
        movl   $10, 12(%rsp)
        leaq   12(%rsp), %rdx
        movl   $5, %esi
        movl   $1, %edi
        call   sum
        movl   %eax, %esi
        leaq   .LC0(%rip), %rdi
        movl   $0, %eax
        call   printf@PLT
        movl   $0, %eax
        addq   $24, %rsp
        ret
```

Passing Parameters: Disassembling

```c
#include <stdio.h>

int sum(int x1, int x2, int x3,
    int x4, int x5, int x6, int x7) {
    return x1 + x2 + x3 + x4 +
           x5 + x6 + x7;
}

int main() {
    printf("%d\n",
           sum(1, 2, 3, 4, 5, 6, 7));
    return 0;
}
```

```
sum:
        addl   %esi, %edi
        addl   %edi, %edx
        addl   %edx, %ecx
        addl   %r8d, %ecx
        addl   %r9d, %ecx
        movl   %ecx, %eax
        addl   8(%rsp), %eax
        ret
.LC0:
        .string     "%d\n"
```

```
main:
        subq   $8, %rsp
        pushq  $7
        movl   $6, %r9d
        movl   $5, %r8d
        movl   $4, %ecx
        movl   $3, %edx
        movl   $2, %esi
        movl   $1, %edi
        call   sum
        addq   $8, %rsp
        movl   %eax, %esi
        leaq   .LC0(%rip), %rdi
        movl   $0, %eax
        call   printf@PLT
        movl   $0, %eax
        addq   $8, %rsp
        ret
```

Case study: sum over array

```c
#include <stdio.h>

int sum(int *a, int n) {
    int total = 0;

    for (int i = 0; i < n; i++) {
      total += a[i];
    }

    return total;
}

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    printf("%d\n", sum(numbers, 5));
    return 0;
}
```

```
sum:
        movl    $0, %edx
        movl    $0, %eax
        jmp     .L2
.L3:
        movslq %edx, %rcx
        addl    (%rdi,%rcx,4), %eax
        addl    $1, %edx
.L2:
        cmpl    %esi, %edx
        jl      .L3
        rep ret
.LC0:
        .string     "%d\n"
```

```
main:
        subq    $40, %rsp
        movl    $1, (%rsp)
        movl    $2, 4(%rsp)
        movl    $3, 8(%rsp)
        movl    $4, 12(%rsp)
        movl    $5, 16(%rsp)
        movq    %rsp, %rdi
        movl    $5, %esi
        call    sum
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        addq    $40, %rsp
        ret
```

Compile to assembly using: `gcc -S -Og array_sum.c`

- Try without `-Og`: What changes? Why?
- Note: use of 128 byte red zone after stack pointer.

Case study: compare arrays

```c
#include <stdio.h>

int array_cmp(int *x, int *y, int n) {
    for (int i = 0; i < n; i++) {
      int cmp = x[i]-y[i];
      if (cmp != 0) {
          return cmp;
      }
    }
    return 0;
}

int main() {
    int x[5] = {1, 2, 3, 4, 5};
    int y[5] = {1, 2, 3, 4, 7};
    printf("%d\n", array_cmp(x, y, 5));
    return 0;
}
```

```
array_cmp:
        movl   $0, %ecx
.L2:
        cmpl   %edx, %ecx
        jge    .L5
        movslq %ecx, %r8
        movl   (%rdi,%r8,4), %eax
        subl   (%rsi,%r8,4), %eax
        jne    .L1
        addl   $1, %ecx
        jmp    .L2
.L5:
        movl   $0, %eax
.L1:
        rep ret
.LC0:
        .string     "%d\n"
```

```
main:
        subq   $72, %rsp
        movl   $1, 32(%rsp)
        movl   $2, 36(%rsp)
        movl   $3, 40(%rsp)
        movl   $4, 44(%rsp)
        movl   $5, 48(%rsp)
        movl   $1, (%rsp)
        movl   $2, 4(%rsp)
        movl   $3, 8(%rsp)
        movl   $4, 12(%rsp)
        movl   $7, 16(%rsp)
        movq   %rsp, %rsi
        leaq   32(%rsp), %rdi
        movl   $5, %edx
        call   array_cmp
        movl   %eax, %esi
        leaq   .LC0(%rip), %rdi
        movl   $0, %eax
        call   printf@PLT
        movl   $0, %eax
        addq   $72, %rsp
        ret
```

- Why do we need 72 bytes on the stack for 10 `int`?

Case study: row-column product

```c
#include <stdio.h>
#define N 3
typedef int matrix[N][N];
static int matmul(matrix x, matrix y,
                  int i, int k) {
    int result = 0;
    for (int j = 0; j < N; j++) {
       result += x[i][j]*y[j][k];
    }
    return result;
}

int main() {
    int x[N][N] = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};

    int y[N][N] = {{3, 0, 1},
                   {4, 2, 8},
                   {0, 1, 7}};

    printf("%d\n", matmul(x, y, 0, 1));
    return 0;
}
```

```
matmul:
      movl   $0, %r10d
      movl   $0, %eax
      cmpl   $2, %r10d
      jg     .L7
      pushq  %rbx
.L3:  movslq %edx, %r8
      leaq   (%r8,%r8,2), %r9
      leaq   0(,%r9,4), %r8
      addq   %rdi, %r8
      movslq %r10d, %r11
      leaq   (%r11,%r11,2), %rbx
      leaq   0(,%rbx,4), %r9
      addq   %rsi, %r9
      movslq %ecx, %rbx
      movl   (%r9,%rbx,4), %r9d
      imull  (%r8,%r11,4), %r9d
      addl   %r9d, %eax
      addl   $1, %r10d
      cmpl   $2, %r10d
      jle    .L3
      popq   %rbx
      ret
.L7:  ret
```

```
main:
      subq   $104, %rsp
      movl   $1, 48(%rsp)
      [...]
      movl   $9, 80(%rsp)
      movl   $3, (%rsp)
      [...]
      movl   $7, 32(%rsp)
      movq   %rsp, %rsi
      leaq   48(%rsp), %rdi
      movl   $1, %ecx
      movl   $0, %edx
      call   matmul
      movl   %eax, %esi
      leaq   .LC0(%rip), %rdi
      movl   $0, %eax
      call   printf@PLT
      movl   $0, %eax
      addq   $104, %rsp
      ret
.LC0:
      .string     "%d\n"
```