Exam Review

selected anonymous feedback: VM

too few concrete examples of translation in lecture

students were poorly prepared for memory HW

> "holy hell, dont be giving us this memory hw and not give us the necessary knowledge and resources to fucking do it. we are supposed to go to class to learn, not to the office hours where our TAs help us learn more than you. give us at least an example problem/solution or some thing next time if you aren't going to teach us"

physical page number versus (full) physical address was confusing

Samira's lecture was good (presumably a different person than above)

logistics

Gilmer Hall 130 (big lecture hall), 7PM, 9 May

3 hours

final format/content

midterm + cumulative final stapled together

- most content from after last midterm
- but cumulative questions, too

about twice the length of a normal midterm

similar format (multiple choice/very short answer)

things we've covered (1)

x86-64 assembly — loops, addressing modes, etc.

C — arrays vs pointers, undefined behavior

ISAs — RISC v CISC, Y86-64 encoding

SEQ — CPU structure, register/memory timings

HDL2D

PIPE — pipelining tradeoffs, our five stages

things we've covered (2)

pipelining hazards — stalling, forwarding, branches

modern CPUs — out-of-order, data-flow model

caching — set-associative caches, write-through/back/etc.

performance — cache, loop optimizations; inlining

processes and exceptions

virtual memory

process

address space — virtual idea of memory

- via page tables + address translation

single control flow

- OS transfers control with exceptions

compiler limitations

needs to generate code that does the same thing...

- ...even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

- needs to assume it might do anything

can't predict what inputs/values will be

- e.g. lots of loop iterations or few?

can't understand code size versus speed tradeoffs

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

9

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

9

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

9

loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp
endOfLoop:
```

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        addq    8(%rdi,%rdx,8), %rax
        addq    $2, %rdx
        jmp     loop
        // plus handle leftover?
```


loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp
endOfLoop:
```

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        addq    8(%rdi,%rdx,8), %rax
        addq    $2, %rdx
        jmp     loop
        // plus handle leftover?
```


aliasing

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
}
```

the compiler **cannot** generate this:

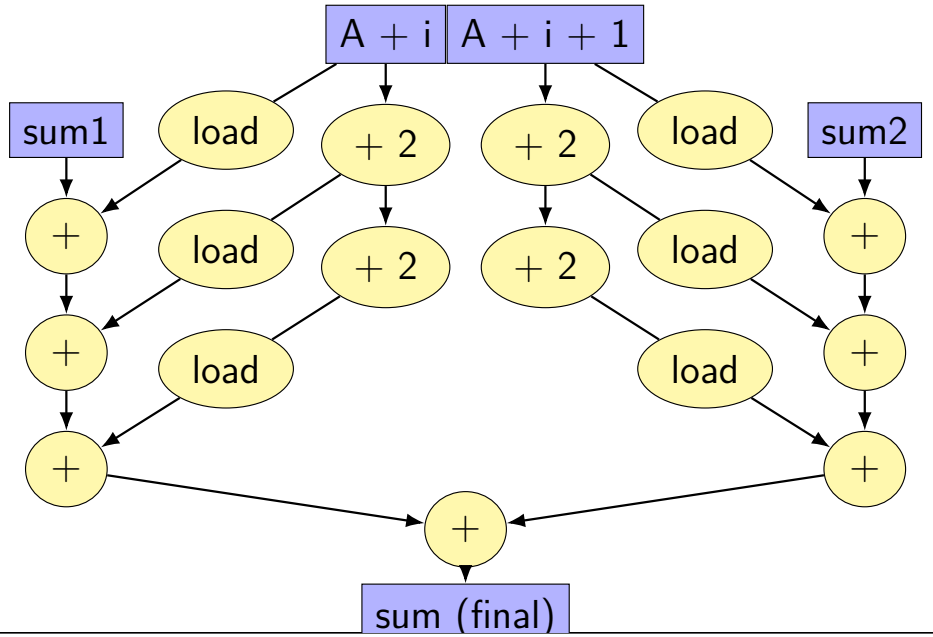
```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax ← *py
        addq    %rax, %rax   // rax ← 2 * *py
        addq    %rax, (%rsi) // *px ← 2 * *py
        ret
```


constant multiplies/divides (2)

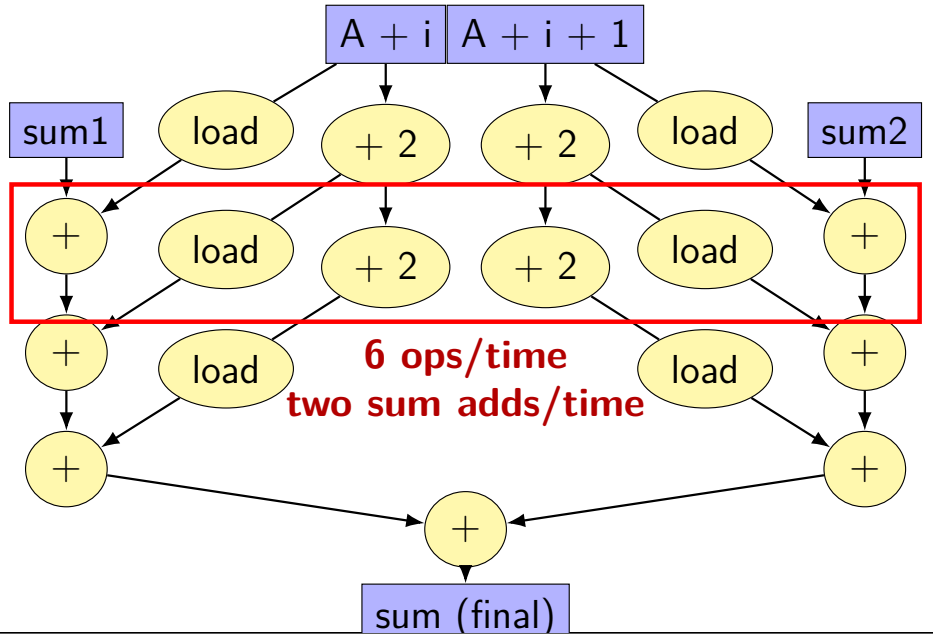
```
int oneHundredth(int x) {
    return x / 100;
}
```

```
oneHundredth:
        movl    %edi, %eax
        movl    $1374389535, %edx
        sarl    $31, %edi
        imull   %edx
        sarl    $5, %edx
        movl    %edx, %eax
        subl    %edi, %eax
        ret
```

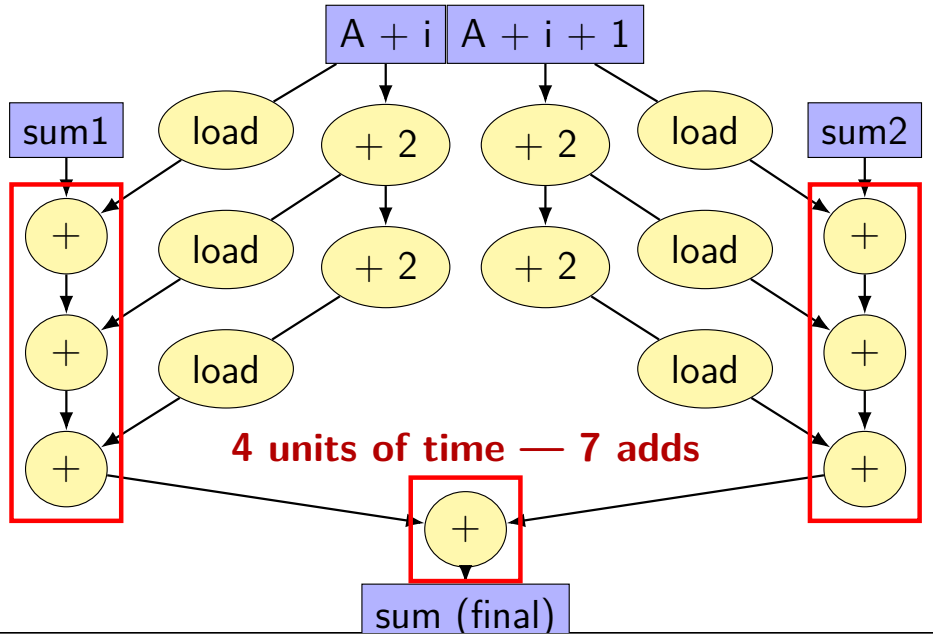
better data-flow



better data-flow



better data-flow

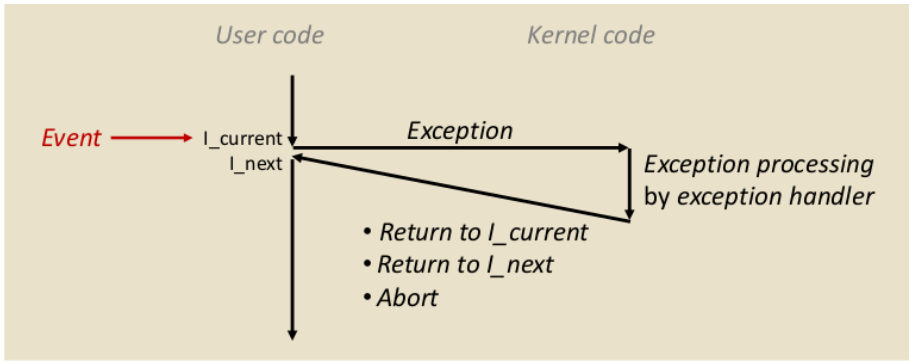


vector instructions

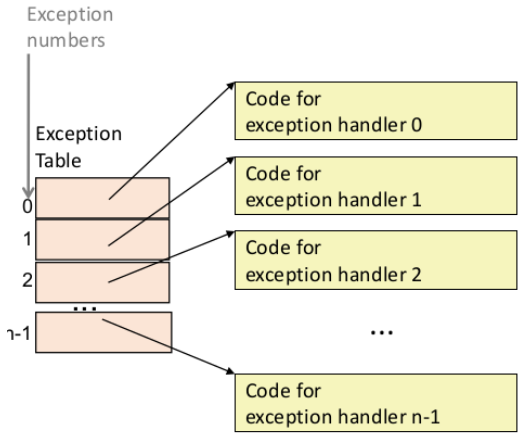
```
void add(int * restrict a, int * restrict b) {
    for (int i = 0; i < 128; ++i)
        a[i] += b[i];
}
```

```
add:
        xorl    %eax, %eax              // init. loop counter
the_loop:
        movdqu  (%rdi,%rax), %xmm0      // load 4 from A
        movdqu  (%rsi,%rax), %xmm1      // load 4 from B
        paddd   %xmm1, %xmm0            // add 4 elements!
        movups  %xmm0, (%rdi,%rax)      // store 4 in A
        addq    $16, %rax               // +4 ints = +16
        cmpq    $512, %rax              // 512 = 4 * 128
        jne     the_loop
        rep ret
```

exceptions

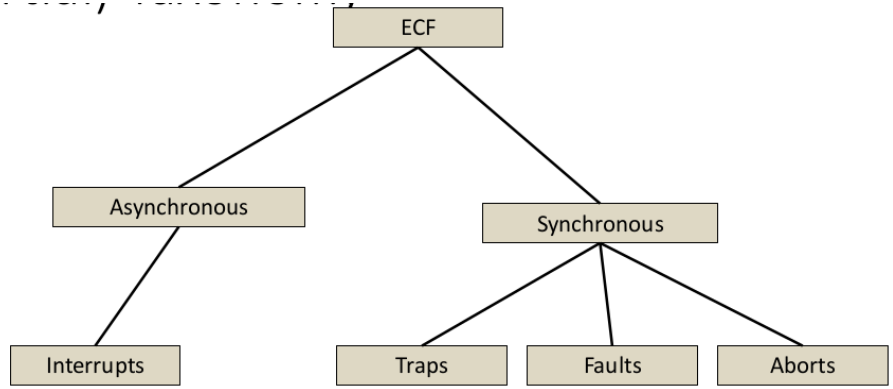


the exception table

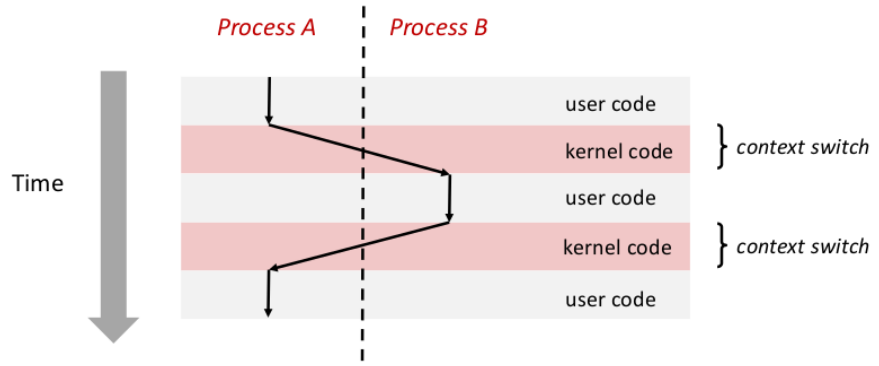


- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

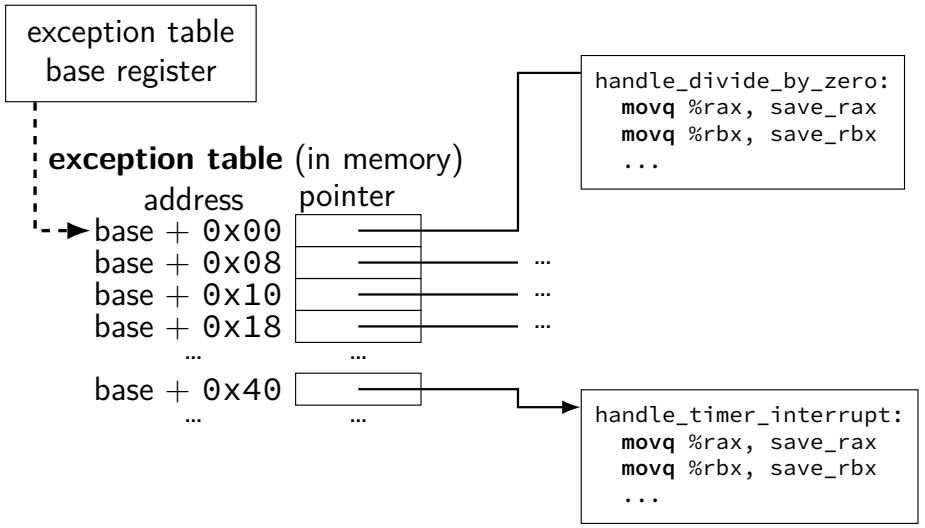
taxonomy



context switching



locating exception handlers



context

all registers values

%rax %rbx, …, %rsp, …

condition codes

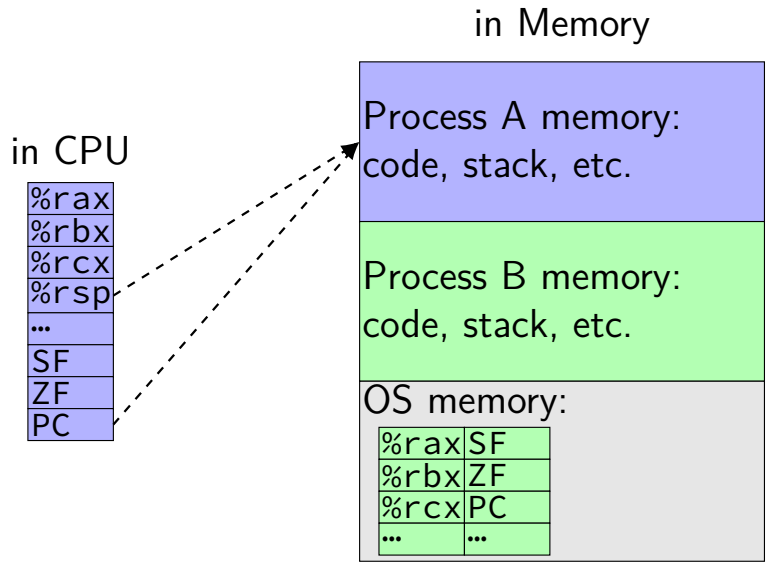
program counter

i.e. all visible state in your CPU except memory

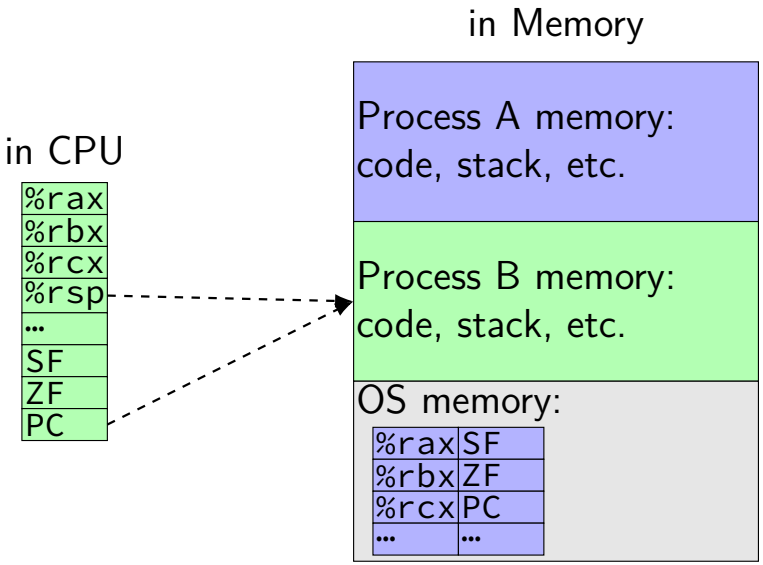
context switch pseudocode

```
context_switch(last, next):
  copy_preexception_pc last->pc
  mov rax,last->rax
  mov rcx, last->rcx
  mov rdx, last->rdx
  ...
  mov next->rdx, rdx
  mov next->rcx, rcx
  mov next->rax, rax
  jmp next->pc
```


contexts (A running)

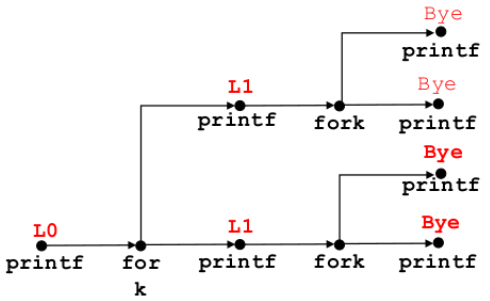


contexts (B running)



fork Example: Two consecutive forks

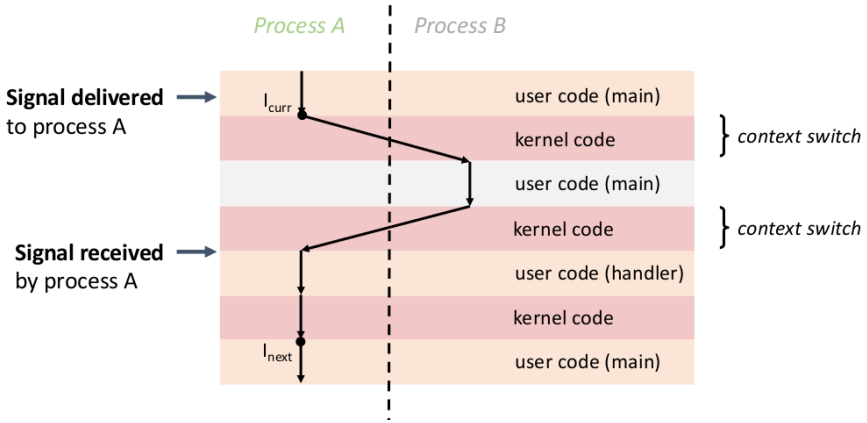
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
                forks.c
```



Feasible output:
```
L0
L1
Bye
Bye
L1
Bye
Bye
```

Infeasible output:
```
L0
Bye
L1
Bye
L1
Bye
Bye
```


signal handling flow



signal delivery

every process has vector of pending signals

(mostly) true if pending, false otherwise

every process has vector of blocked signals

signal delivered if pending, unblocked

signal handler safety

could interrupt manipulating global data structures

user code: keep simple — avoid sharing data structures between signal handler/other code

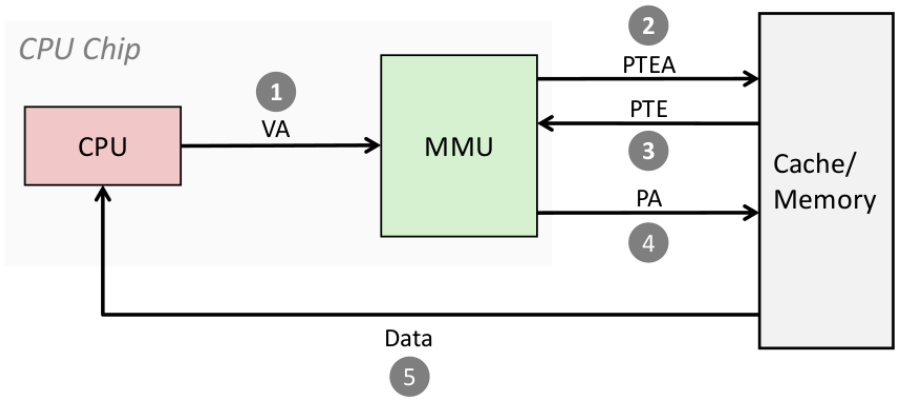
or use signal blovcking when manipulating shared data

library code: async-signal-safe functions

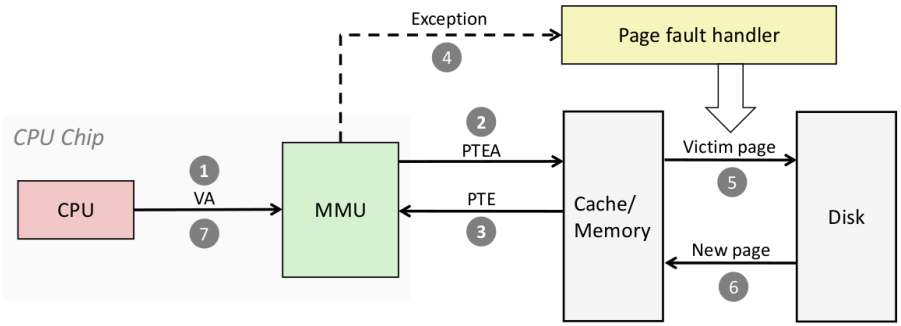
gaurneteed not to use global data structures
not printf, malloc, etc.

volatile — tells compiler "don't put in register"

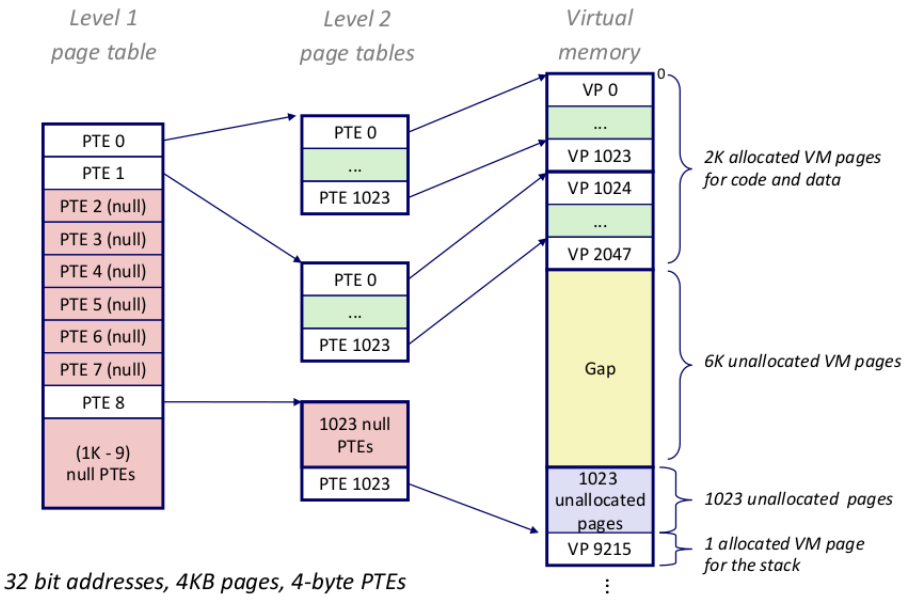
page hit flow



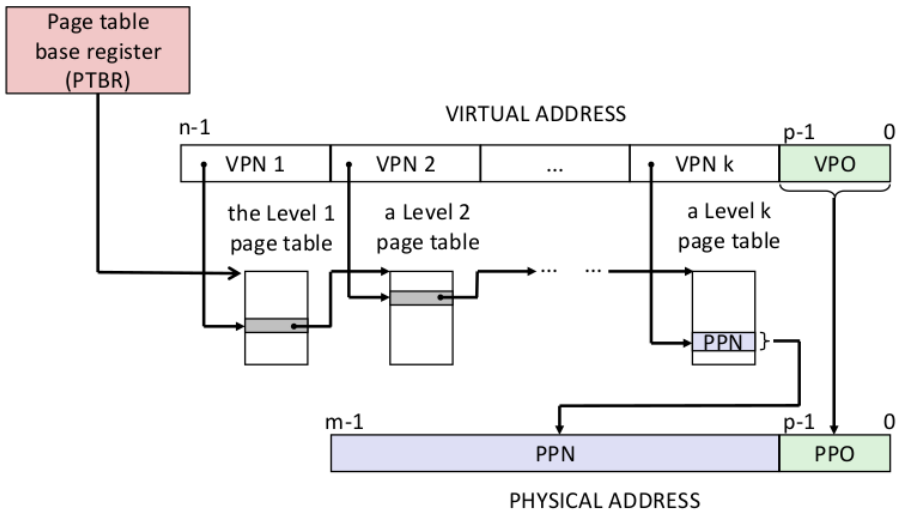
page fault flow



two-level page table

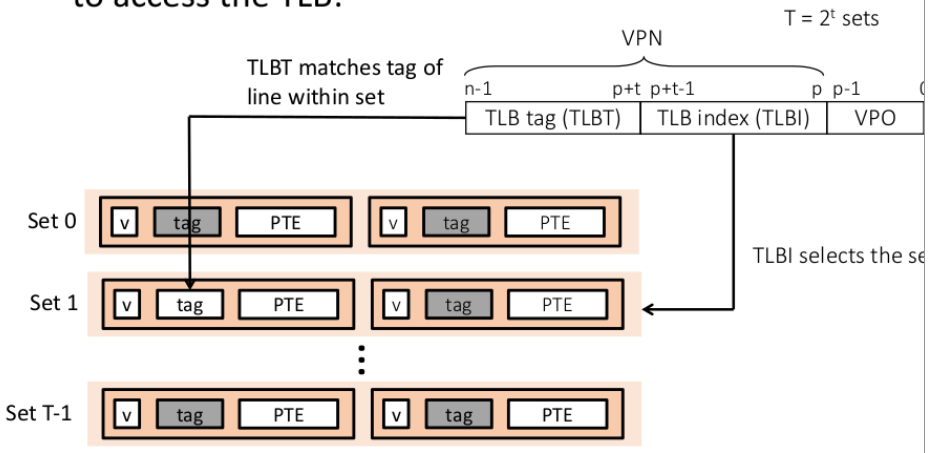


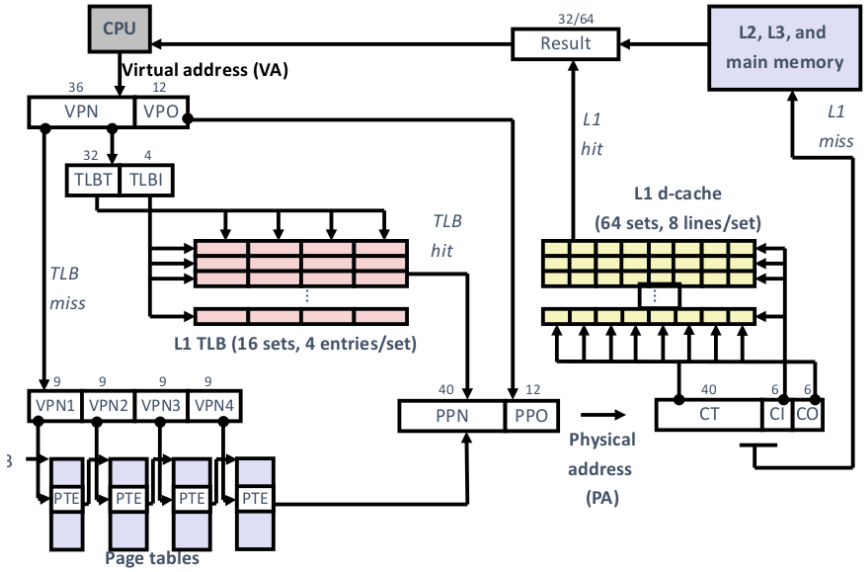
k-level translation



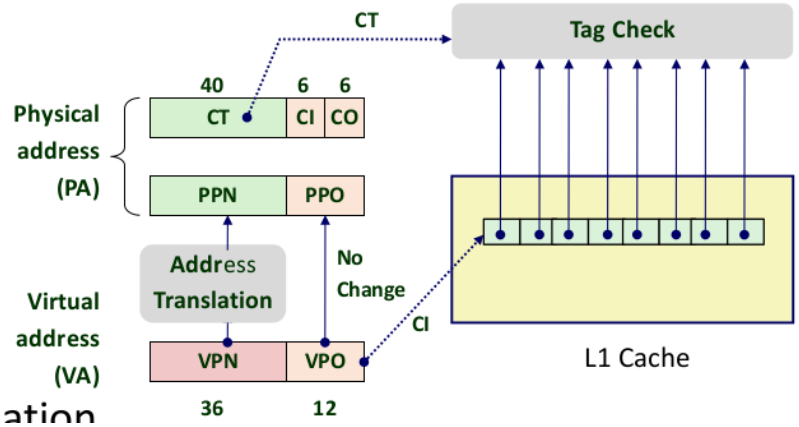
Accessing the TLB

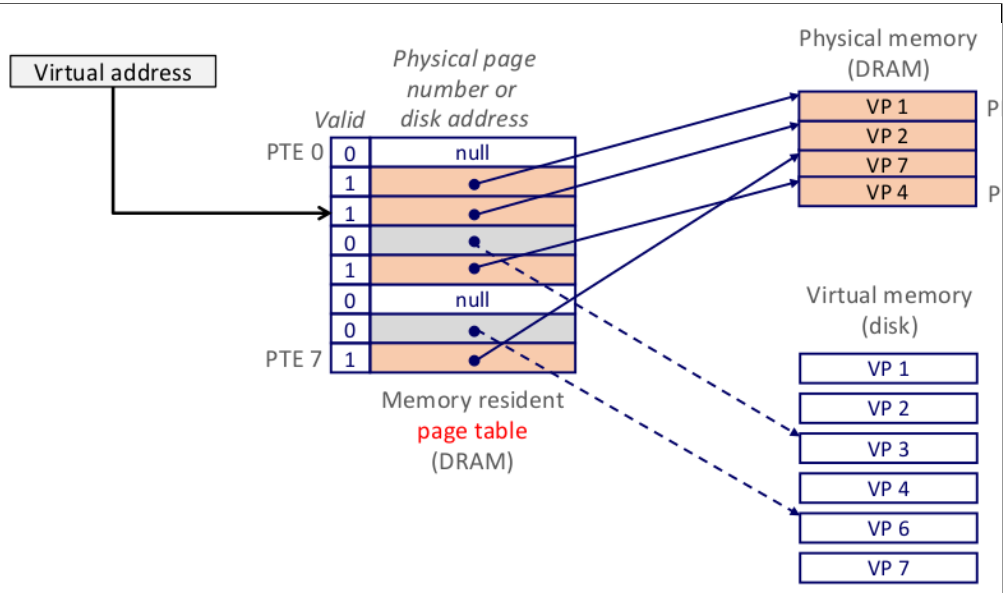
- MMU uses the VPN portion of the virtual address to access the TLB:





virtually-indexed, physically-tagged





signal handler unsafety (0)

```
void foo() {
    /* SIGINT might happen while foo() is running */
    char *p = malloc(1024);
    ...
}

/* signal handler for SIGINT
   (registered elsewhere with sigaction() */
void handle_sigint() {
    printf("You pressed control-C.\n");
}
```


signal handler unsafety (1)

```
void foo() {
    /* This malloc() call interrupted */
    char *p = malloc(1024);
    ...
}
void *malloc(size_t size) {
    ...
    to_return = next_to_return;
    /* SIGNAL HAPPENS HERE */
    next_to_return += size;
    return to_return;
}
void handle_sigint() {
    printf("You pressed control-C.\n");
}
```

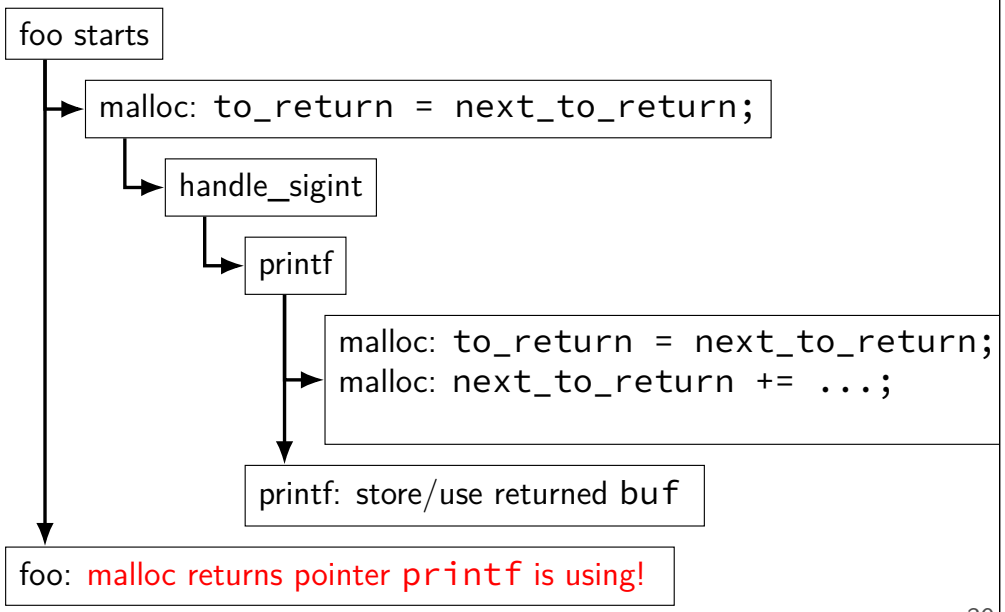

signal handler unsafety (1)

```
void foo() {
    /* This malloc() call interrupted */
    char *p = malloc(1024);
    ...
}
void *malloc(size_t size) {
    ...
    to_return = next_to_return;
    /* SIGNAL HAPPENS HERE */
    next_to_return += size;
    return to_return;
}
void handle_sigint() {
    printf("You pressed control-C.\n");
}
```

signal handler unsafety (2)

```
void handle_sigint() {
    printf("You pressed control-C.\n");
}

int printf(...) {
    static char *buf;
    ...
    buf = malloc()
    ...
}
```


signal handler unsafety: timeline



signal handler unsafety (3)

```
foo() {
  char *p = malloc(1024)... {
    to_return = next_to_return;
    handle_sigint() { /* signal delivered here */
      printf("You pressed control-C.\n") {
        buf = malloc(...) {
          to_return = next_to_return;
          next_to_return += size;
          return to_return;
        }
        ...
      }
    }
    next_to_return += size;
    return to_return;
  }
  /* now p points to buf used by printf! */
}
```


signal handler unsafety (3)

```
foo() {
  char *p = malloc(1024)... {
    to_return = next_to_return;
    handle_sigint() { /* signal delivered here */
      printf("You pressed control-C.\n") {
        buf = malloc(...) {
          to_return = next_to_return;
          next_to_return += size;
          return to_return;
        }
        ...
      }
    }
    next_to_return += size;
    return to_return;
  }
  /* now p points to buf used by printf! */
}
```

signal handler safety

POSIX (standard that Linux follows) defines "async-signal-safe" functions

these must work correctly in signal handlers no matter what they interrupt

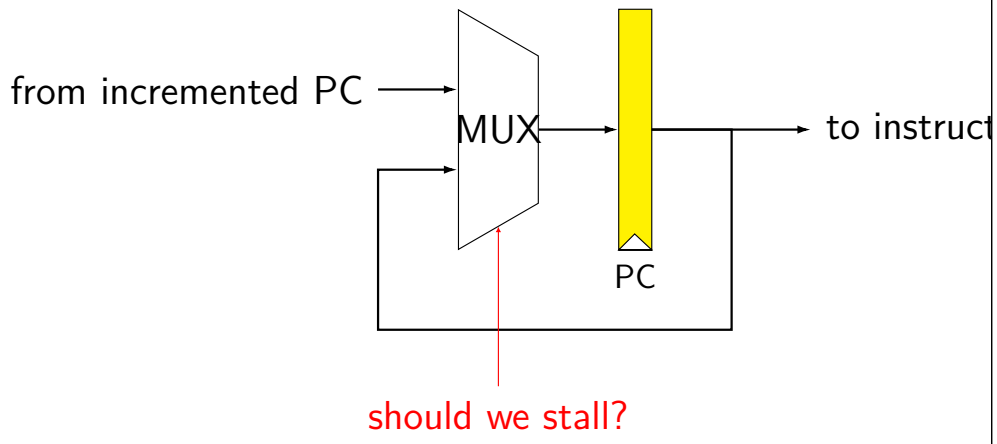
includes: `write`, `_exit`

does not include: `printf`, `malloc`, `exit`

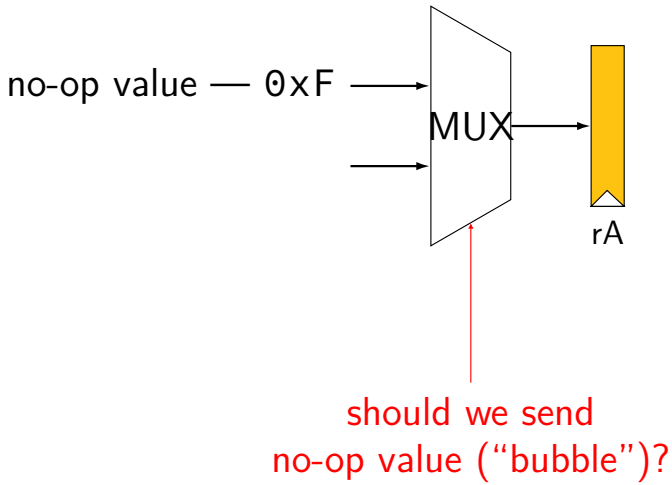
Exam 2

Exam Review

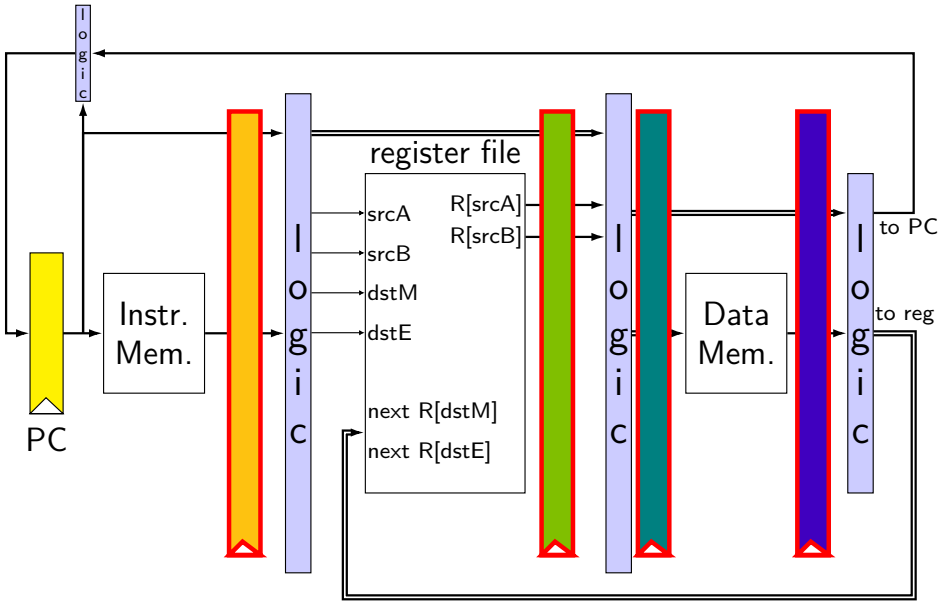
fetch/fetch logic — advance or not



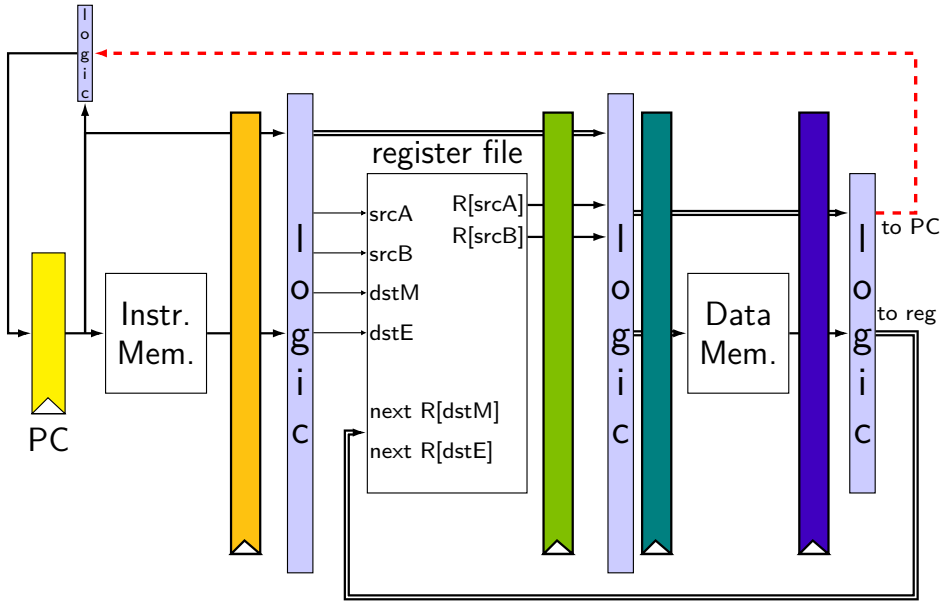
fetch/decode logic — bubble or not



SEQ + pipeline registers



SEQ + pipeline registers



ret stall

time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

N N

stall (S) = keep old value; normal (N) = use new value

bubble (N) = use default (no-op);

ret stall

time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

stall (S) = keep old value; normal (N) = use new value
bubble (N) = use default (no-op);

ret stall

time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

stall (S) = keep old value; normal (N) = use new value
bubble (N) = use default (no-op);

ret stall

time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

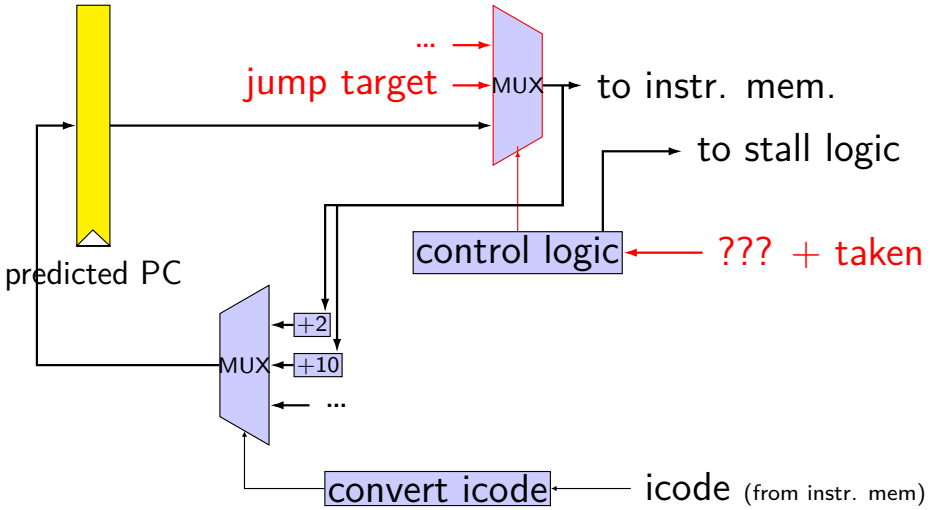
stall (S) = keep old value; normal (N) = use new value
bubble (N) = use default (no-op);

ret stall

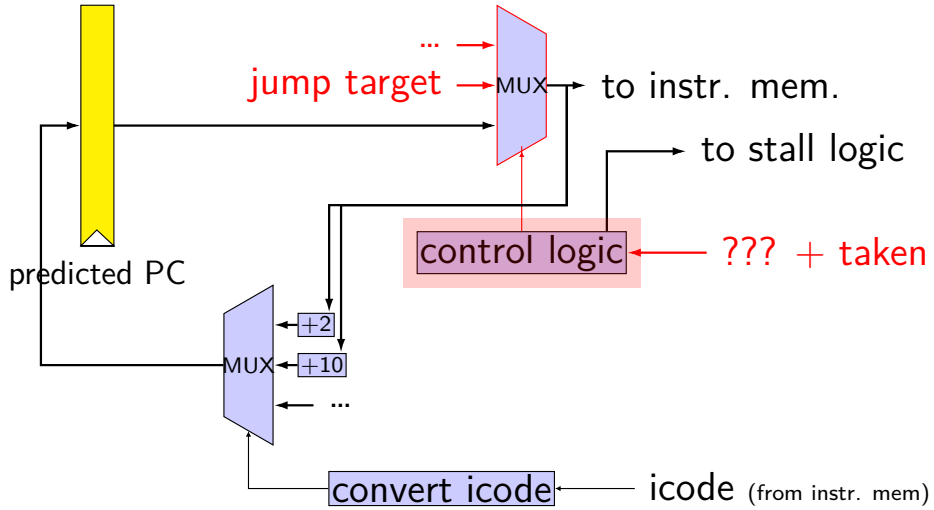
time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

stall (S) = keep old value; normal (N) = use new value
bubble (N) = use default (no-op);

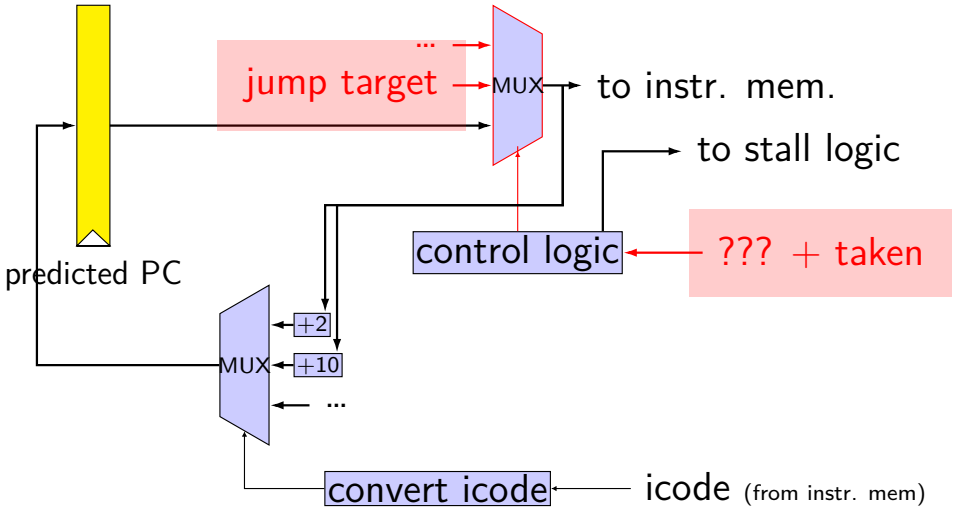
PC update (rearranged)



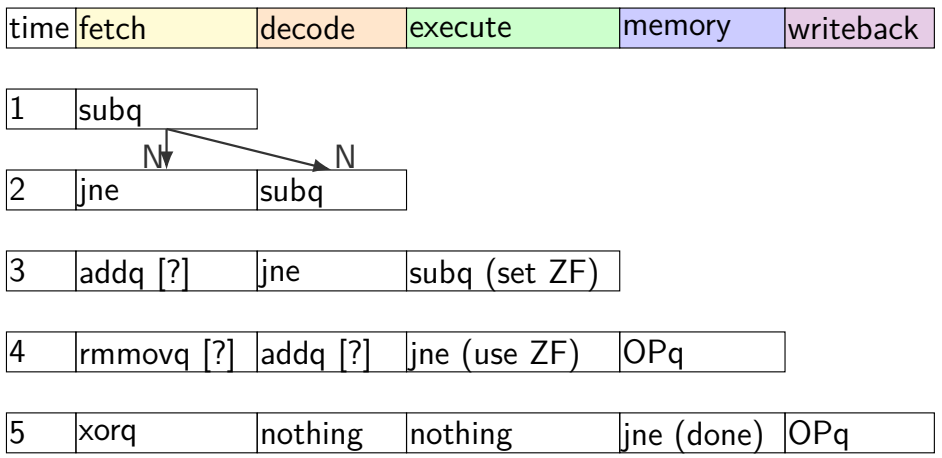
PC update (rearranged)



PC update (rearranged)

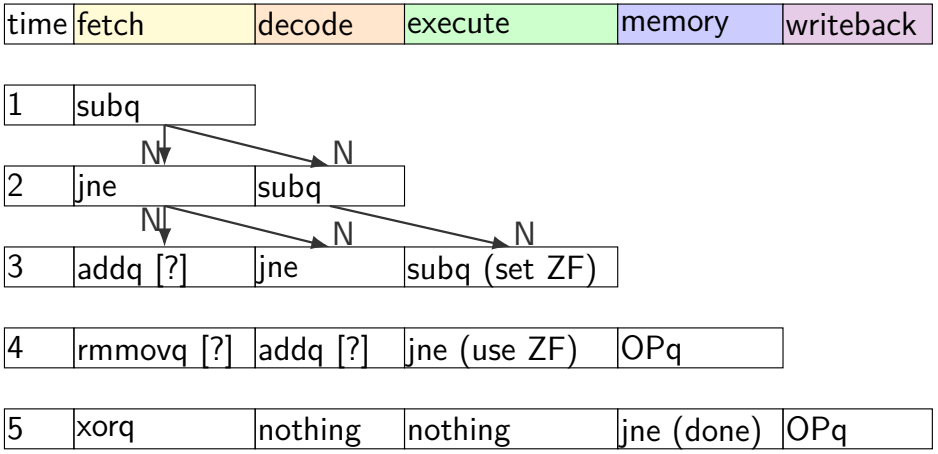


squashing with stall/bubble



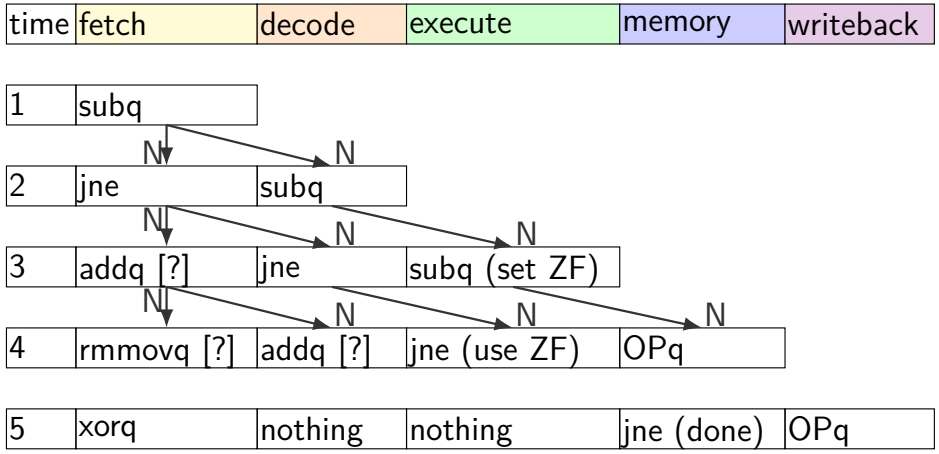
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

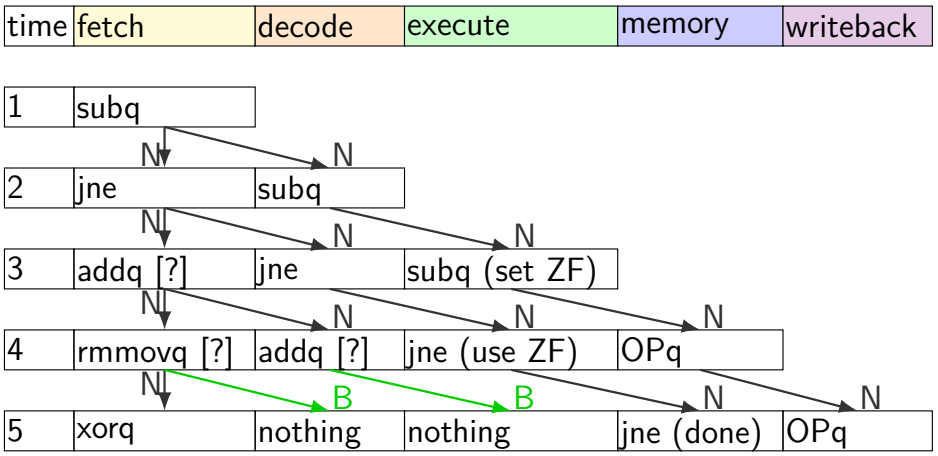
squashing with stall/bubble



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

49

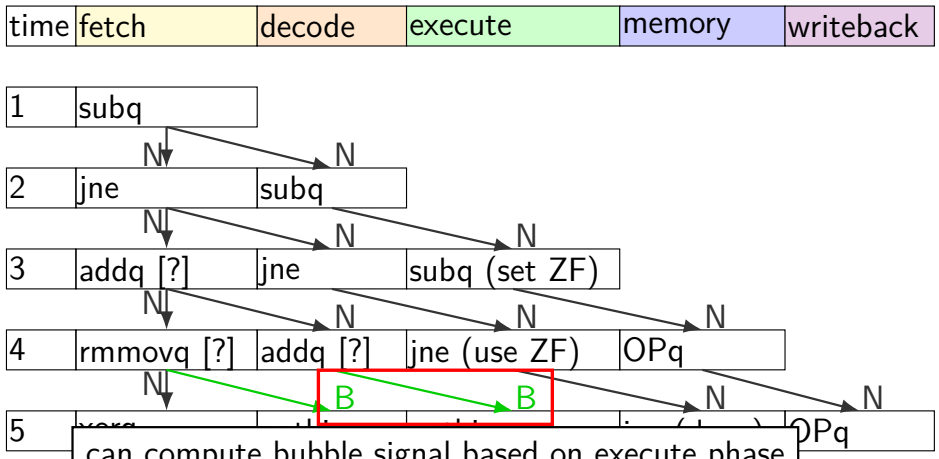
squashing with stall/bubble



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

49

squashing with stall/bubble

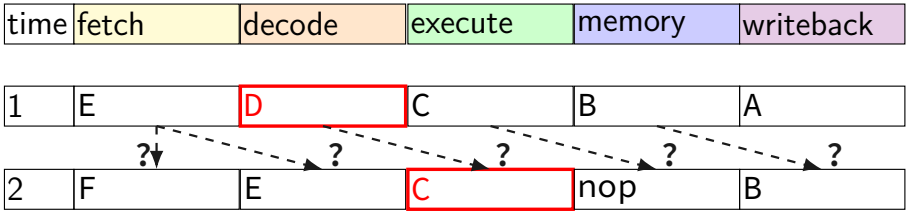


can compute bubble signal based on execute phase
won't even start CC write for addq

stal ew value
bubble (B) = use default (no-op);

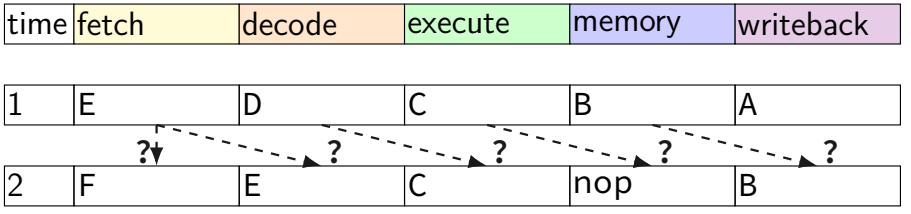
49

exercise: squash + stall (2)



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

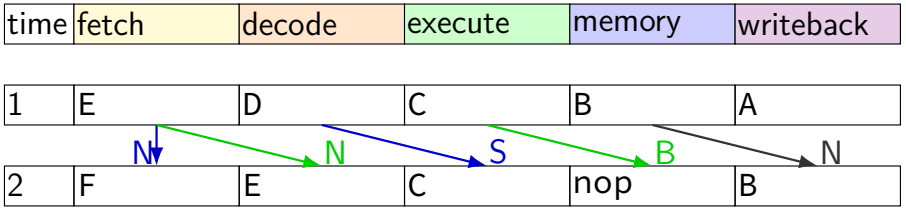
exercise: squash + stall (2)



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

exercise: what are the ?s
write down your answers,
then compare with your neighbors

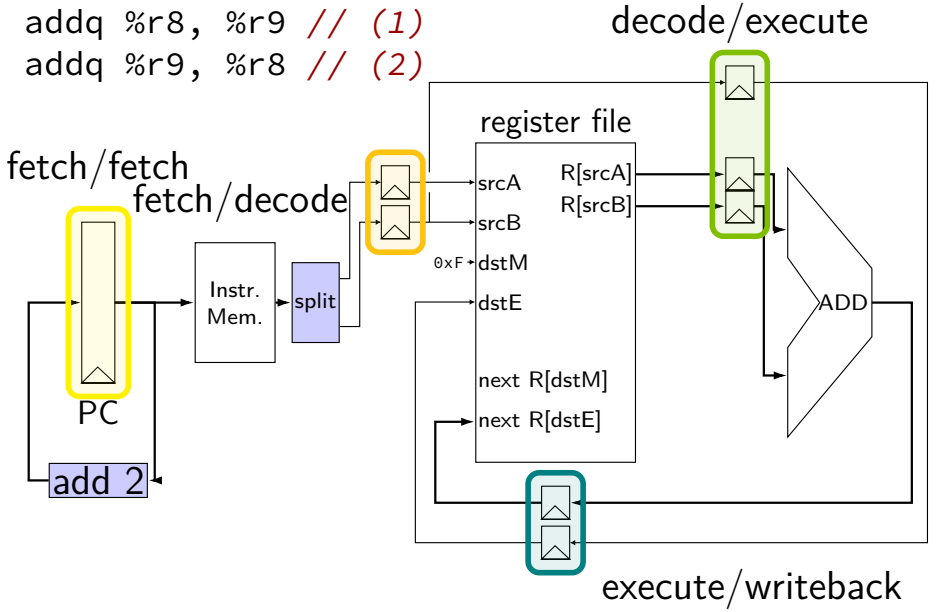
exercise: squash + stall (2)



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

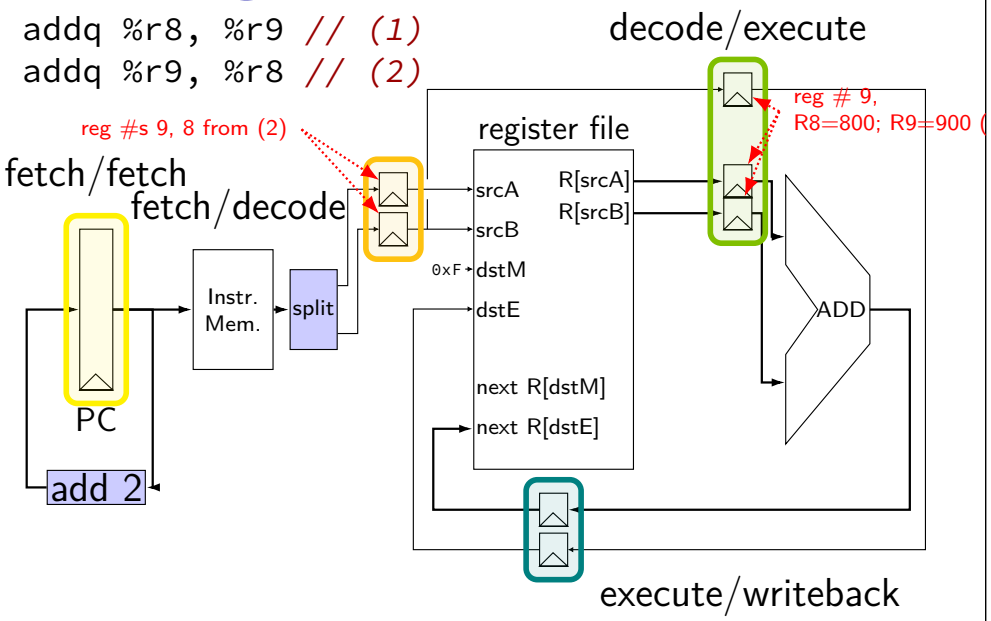
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



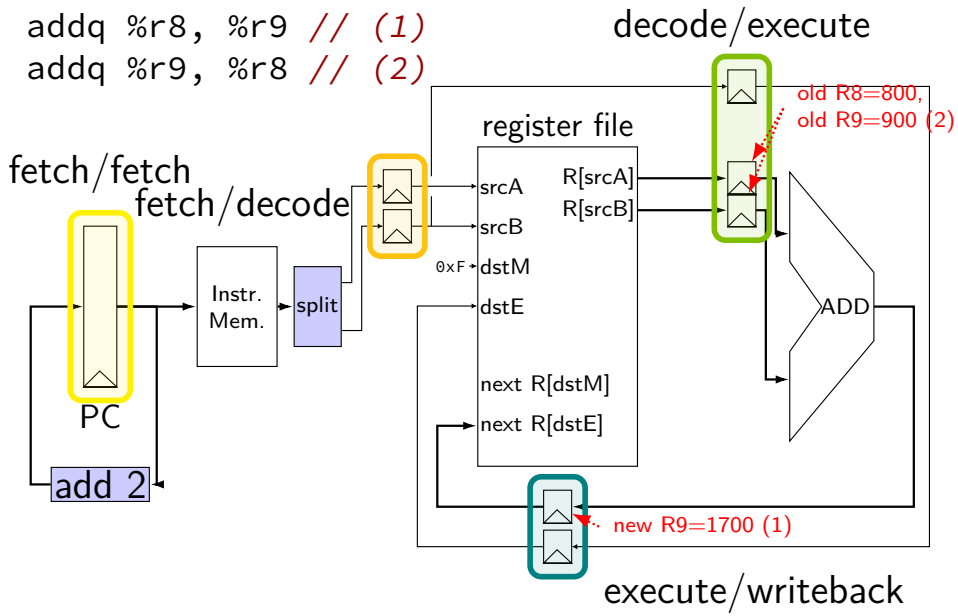
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



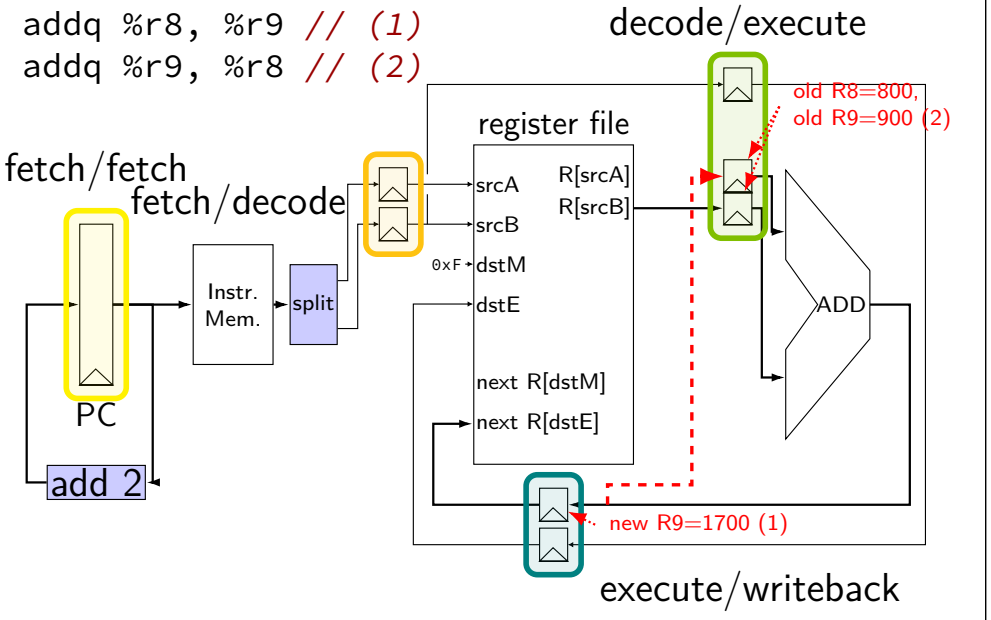
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



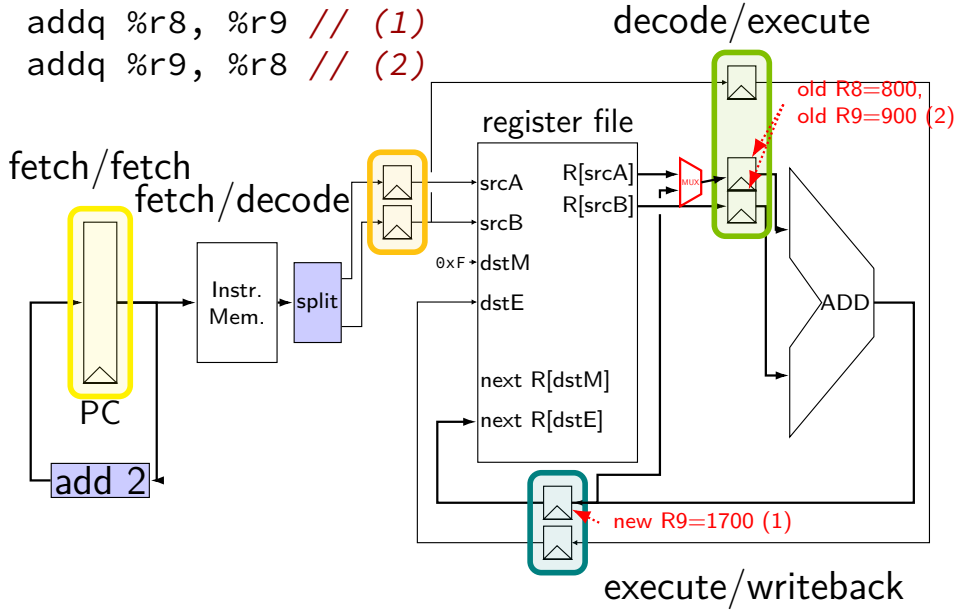
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```

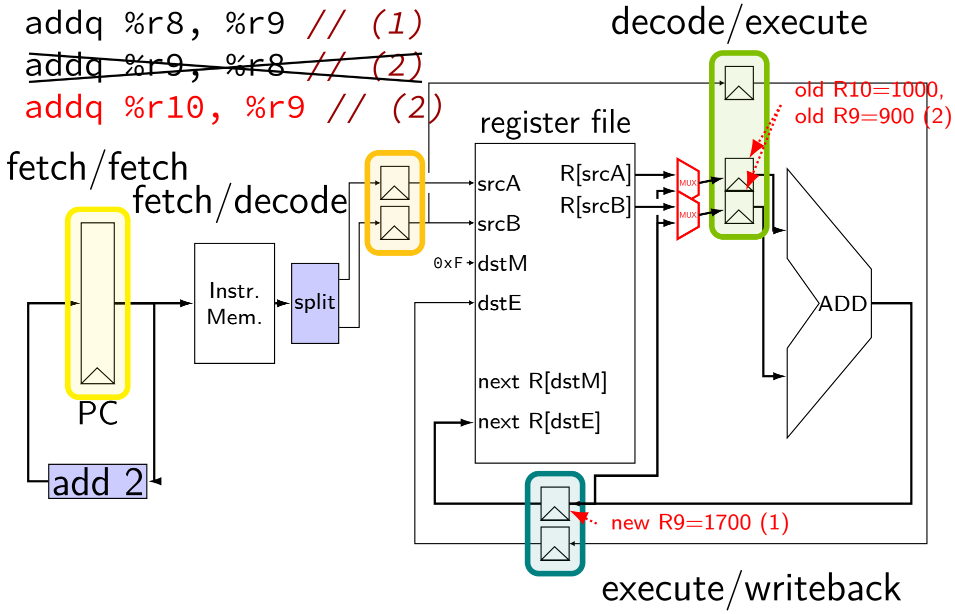


forwarding

```
addq %r8, %r9 // (1)
```

~~`addq %r9, %r8 // (2)`~~

```
addq %r10, %r9 // (2)
```



some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
addq %r8, %r9	F	D	E	M	W				
subq %r9, %r11		F	D	E	M	W			
mrmovq 4(%r11), %r10			F	D	E	M	W		
rmmovq %r9, 8(%r11)				F	D	E	M	W	
xorq %r10, %r9					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
addq %r8, %r9	F	D	E	M	W				
subq %r9, %r11		F	D	E	M	W			
mrmovq 4(%r11), %r10			F	D	E	M	W		
rmmovq %r9, 8(%r11)				F	D	E	M	W	
xorq %r10, %r9					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
addq %r8, %r9	F	D	E	M	W				
subq %r9, %r11		F	D	E	M	W			
mrmovq 4(%r11), %r10			F	D	E	M	W		
rmmovq %r9, 8(%r11)				F	D	E	M	W	
xorq %r10, %r9					F	D	E	M	W

some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
addq %r8, %r9		F	D	E	M	W				
subq %r9, %r11			F	D	E	M	W			
mrmovq 4(%r11), %r10				F	D	E	M	W		
rmmovq %r9, 8(%r11)					F	D	E	M	W	
xorq %r10, %r9						F	D	E	M	W

some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
addq %r8, %r9		F	D	E	M	W				
subq %r9, %r11			F	D	E	M	W			
mrmovq 4(%r11), %r10				F	D	E	M	W		
rmmovq %r9, 8(%r11)					F	D	E	M	W	
xorq %r10, %r9						F	D	E	M	W

some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
addq %r8, %r9		F	D	E	M	W				
subq %r9, %r11			F	D	E	M	W			
mrmovq 4(%r11), %r10				F	D	E	M	W		
rmmovq %r9, 8(%r11)					F	D	E	M	W	
xorq %r10, %r9						F	D	E	M	W

some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
addq %r8, %r9		F	D	E	M	W				
subq %r9, %r11			F	D	E	M	W			
mrmovq 4(%r11), %r10				F	D	E	M	W		
rmmovq %r9, 8(%r11)					F	D	E	M	W	
xorq %r10, %r9						F	D	E	M	W

multiple forwarding paths (1)

cycle #	0	1	2	3	4	5	6	7	8
addq %r10, %r8	F	D	E	M	W				
addq %r11, %r8		F	D	E	M	W			
addq %r12, %r8			F	D	E	M	W		

multiple forwarding paths (1)

cycle #	0	1	2	3	4	5	6	7	8
addq %r10, %r8	F	D	E	M	W				
addq %r11, %r8		F	D	E	M	W			
addq %r12, %r8			F	D	E	M	W		

forwarding after decode

cycle #	0	1	2	3	4	5	6	7	8
mrmovq 0(%r10), %r8	F	D	E	M	W				
rmmovq %r8, 0(%r10)		F	D	E	M	W			
addq %r12, %r8			F	D	E	M	W		

forwarding after decode

cycle #	0	1	2	3	4	5	6	7	8
mrmovq 0(%r10), %r8	F	D	E	M	W				
rmmovq %r8, 0(%r10)		F	D	E	M	W			
addq %r12, %r8			F	D	E	M	W		

forwarding after decode

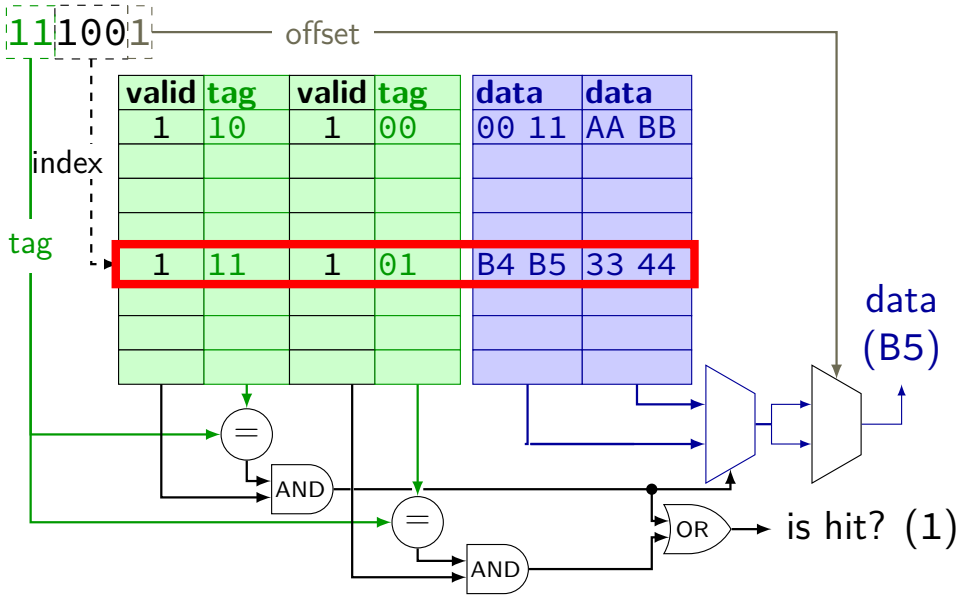
cycle #	0	1	2	3	4	5	6	7	8
`mrmovq 0(%r10), %r8`	F	D	E	M	W				
`rmmovq %r8, 0(%r10)`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

exercise: stalls and forwarding (3)

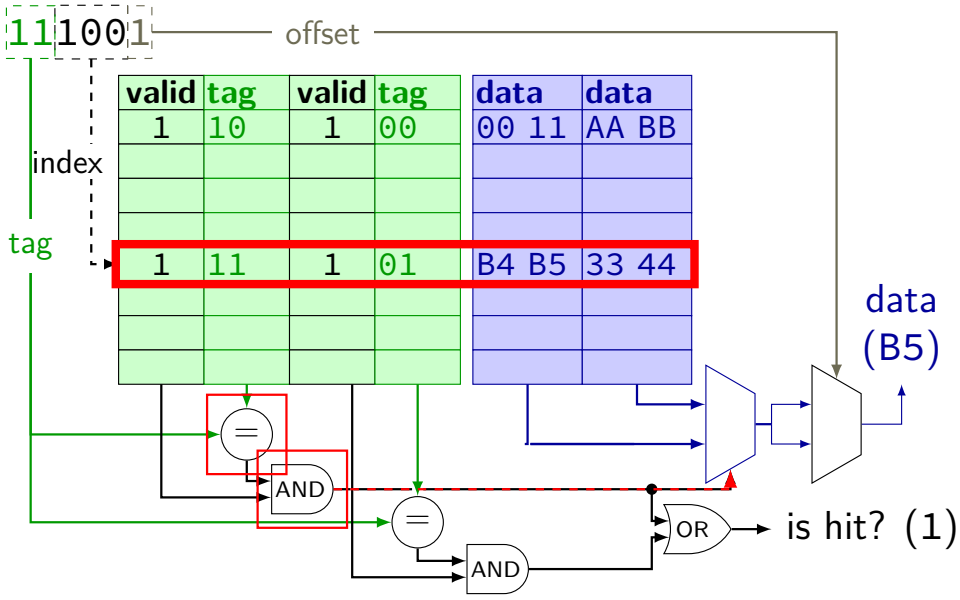
```
addq %rax, %rax
jne foo // taken
foo: mrmovq 0(%rax), %rbx
addq %rbx, %rcx
mrmovq 0(%rbx), %rcx
```

Are there stalls? Where does forwarding happen?

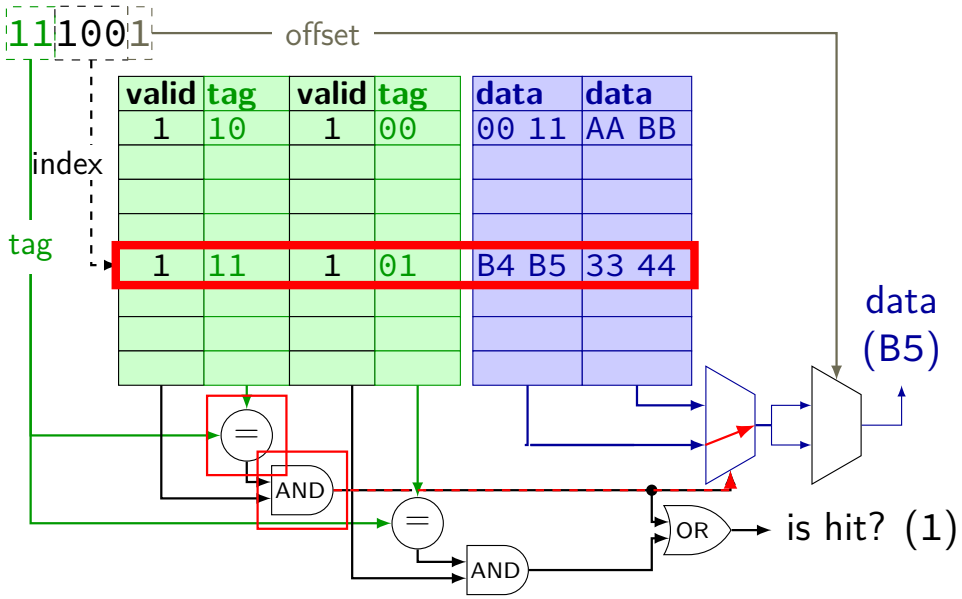
cache operation (associative)



cache operation (associative)



cache operation (associative)



Tag-Index-Offset formulas

m	memory addreses bits (Y86-64: 64)
E	number of blocks per set ("ways")
$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
$t = m - (s + b)$	tag bits
$C = B \times S \times E$	cache size (excluding metadata)

matrix squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

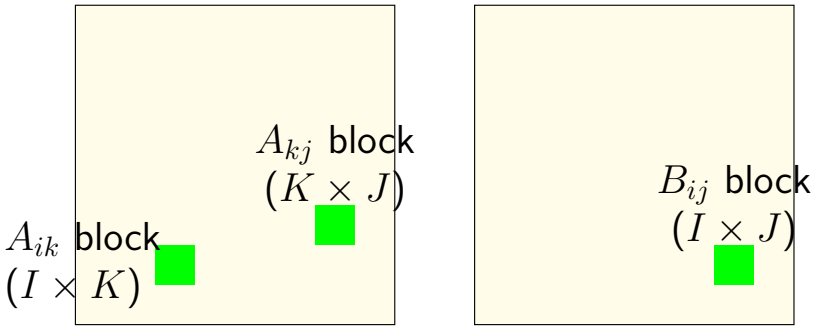
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

/* version 2: outer loop is k, middle is i */
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```


matrix squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

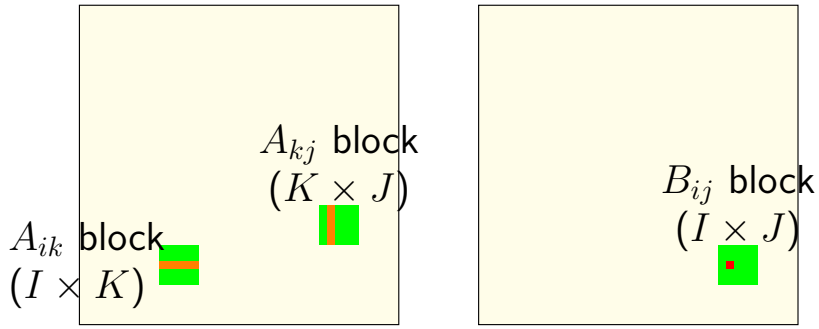
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

/* version 2: outer loop is k, middle is i */
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```

array usage: block



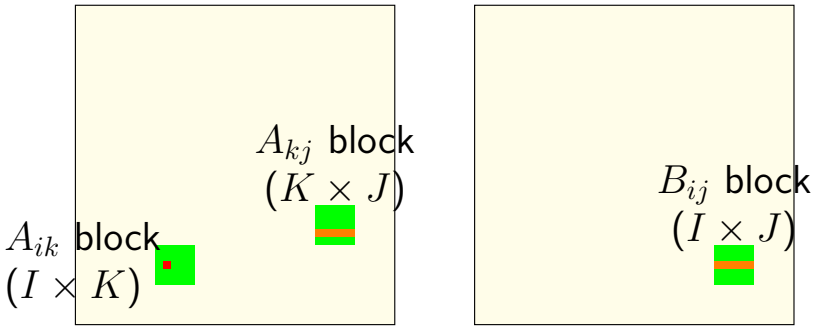
inner loop keeps "blocks" from A, B in cache

array usage: block



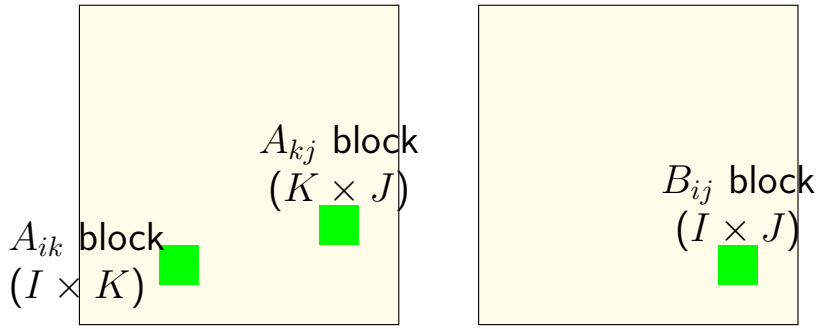
B_{ij} calculation uses strips from A
K calculations for one load (cache miss)

array usage: block



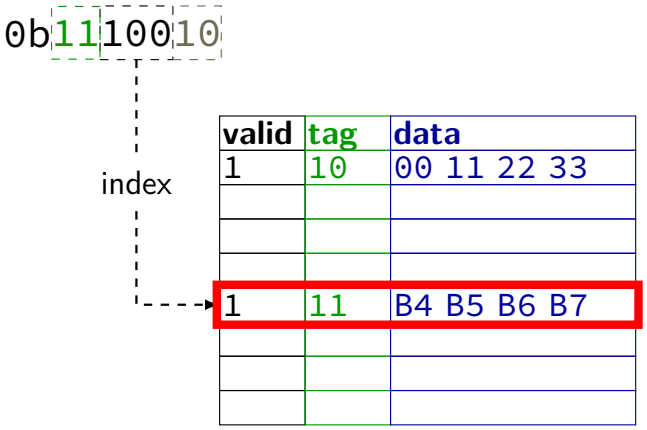
A_{ik} calculation uses strips from A, B
J calculations for one load (cache miss)

array usage: block

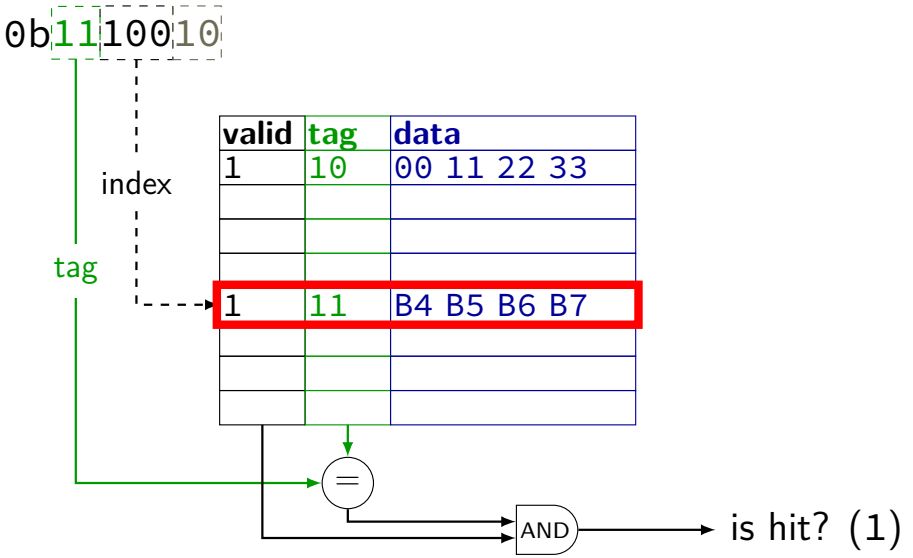


(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ loads
(assuming everything stays in cache)

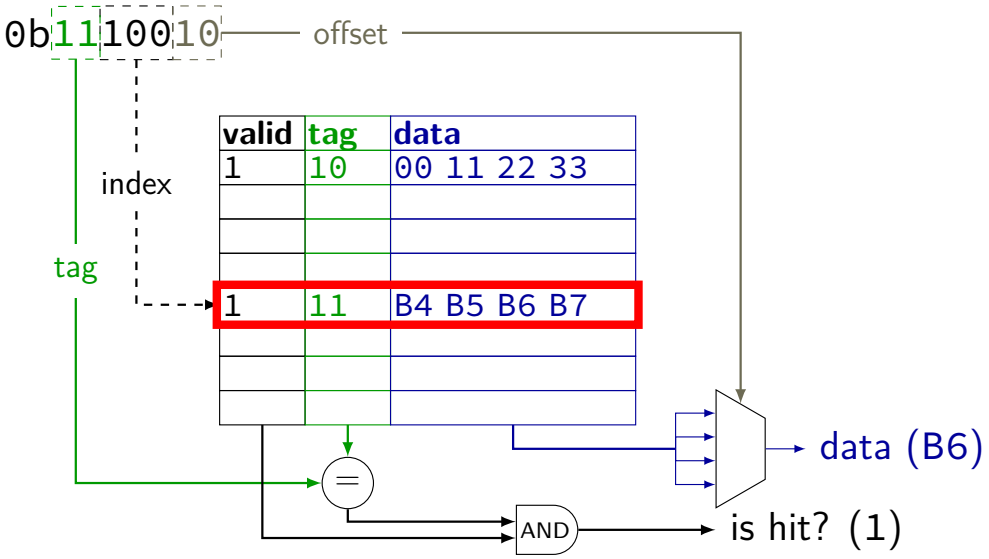
cache operation (read)



cache operation (read)



cache operation (read)



example access pattern (1)

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

2 byte blocks, 4 sets

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

example access pattern (1)

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

2 byte blocks, 4 sets

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	miss
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	0		
10	0		
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	0		
10	0		
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	01100	mem[0x60] mem[0x61]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	
01100100 (64)	

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	miss
01100100 (64)	miss

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	miss
01100100 (64)	miss

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

example access pattern (1)

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	miss
01100100 (64)	miss

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

miss caused by conflict

$m = 8$ bit addresses
$S = 4 = 2^s$ sets
$s = 2$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 5$ tag bits

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0		set 0	0		
1	0		set 1	0		

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

$m = 8$ bit addresses
$S = 2 = 2^s$ sets
$s = 1$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 6$ tag bits

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

62

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	
00000000 (00)	
01100100 (64)	

62

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	
01100100 (64)	

62

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

needs to replace block in set 0!

Tag-Index-Offset (TIO)

address 001111 = FF

cache	tag	index	offset
2 byte blocks, 4 sets	???	???	???
2 byte blocks, 8 sets	???	???	???
4 byte blocks, 2 sets	???	???	???

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE FF

4 byte blocks, 2 sets

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE FF

Tag-Index-Offset (TIO)

address 001111 = FF

cache	tag	index	offset
2 byte blocks, 4 sets	???	???	1
2 byte blocks, 8 sets	???	???	1
4 byte blocks, 2 sets	???	???	???

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE FF

$2 = 2^1$ bytes in block
1 bit to say which byte

4 byte blocks, 2 sets

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE FF

Tag-Index-Offset (TIO)

address `001111` = FF

cache	tag	index	offset
2 byte blocks, 4 sets	???	???	1
2 byte blocks, 8 sets	???	???	1
4 byte blocks, 2 sets	???	???	11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0		
11	1		

4 byt

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE FF

$4 = 2^2$ bytes in block
2 bits to say which byte

Tag-Index-Offset (TIO)

address `001111` = FF

cache	tag	index	offset
2 byte blocks, 4 sets	???	11	1
2 byte blocks, 8 sets	???		1
4 byte blocks, 2 sets	???	1	11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE FF

4 byte blocks, 2 sets

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE FF

$2^2 = 4$ sets
2 bits to index set

Tag-Index-Offset (TIO)

address `001111` = FF

cache	tag	index	offset
2 byte blocks, 4 sets	???	11	1
2 byte blocks, 8 sets	???	111	1
4 byte blocks, 2 sets	???	1	11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0		
11	1		

4

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE FF

$2^3 = 8$ sets
3 bits to index set

Tag-Index-Offset (TIO)

address `001111` = FF

cache	tag	index	offset
2 byte blocks, 4 sets	???	11	1
2 byte blocks, 8 sets	???	111	1
4 byte blocks, 2 sets	???	1	11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE FF

4 byte blocks, 2 sets

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
01			--
10			--
11			BB
110			--
111	1	00	EE FF

$2^1 = 2$ sets
1 bit to index set

Tag-Index-Offset (TIO)

address `001111` = FF

cache	tag	index	offset
2 byte blocks, 4 sets	001	11	1
2 byte blocks, 8 sets	00	111	1
4 byte blocks, 2 sets	001	1	11

tag — whatever is left over

00	1	000	00 11
01	1	001	AA BB
10	0	--	-- --
11	1	001	EE FF

4 byte blocks, 2 sets

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
111	1	00	EE FF

2004 CPU

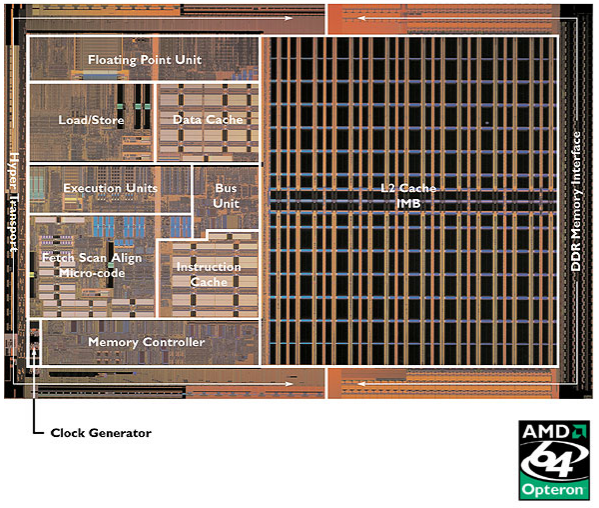
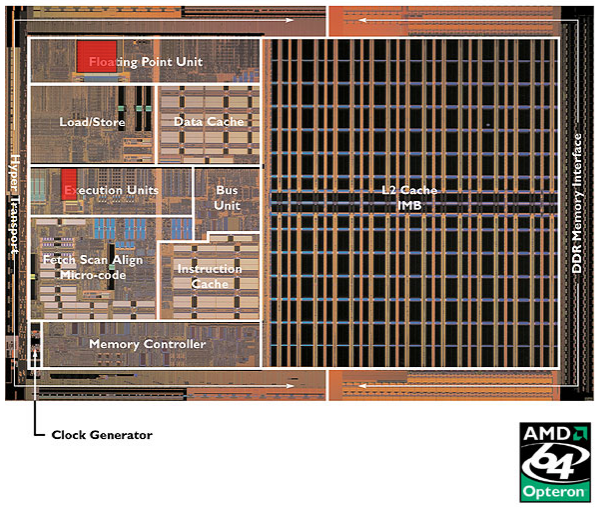


Image: approx 2004 AMD press image of Opteron die

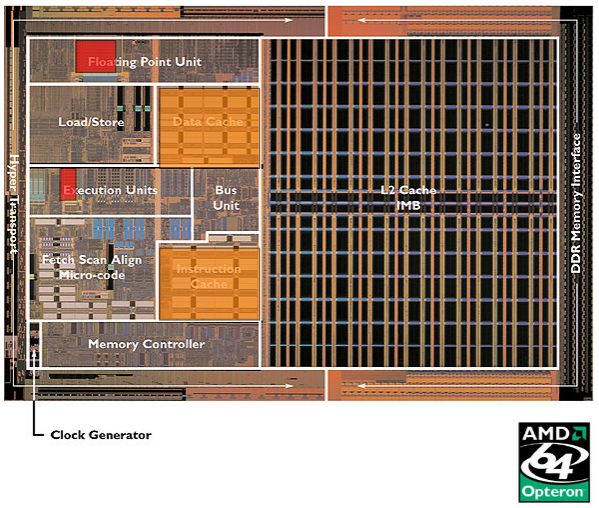
2004 CPU



Registers

Image: approx 2004 AMD press image of Opteron die

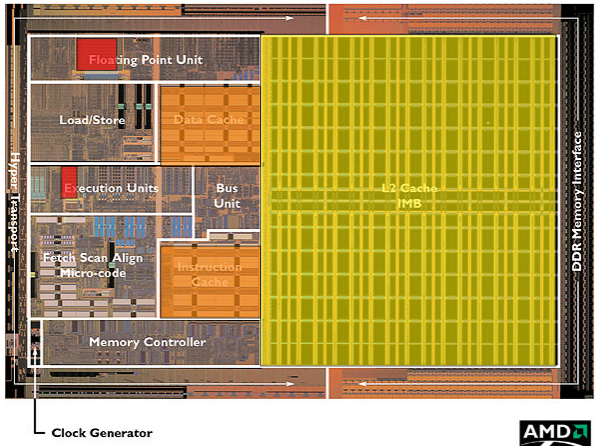
2004 CPU



Registers
L1 cache

Image: approx 2004 AMD press image of Opteron die

2004 CPU



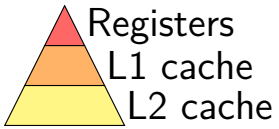
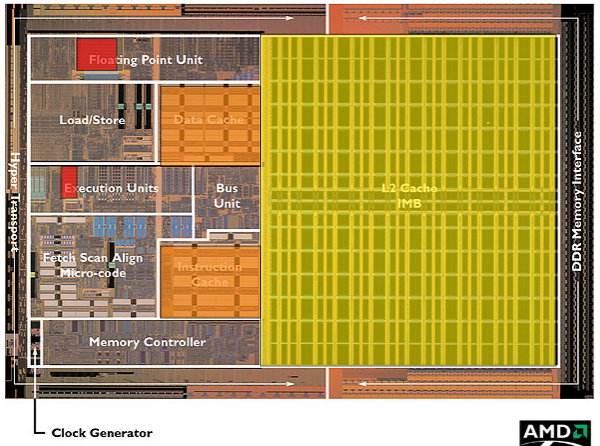


Image: approx 2004 AMD press image of Opteron die

2004 CPU



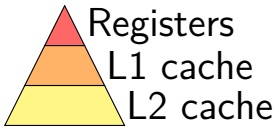
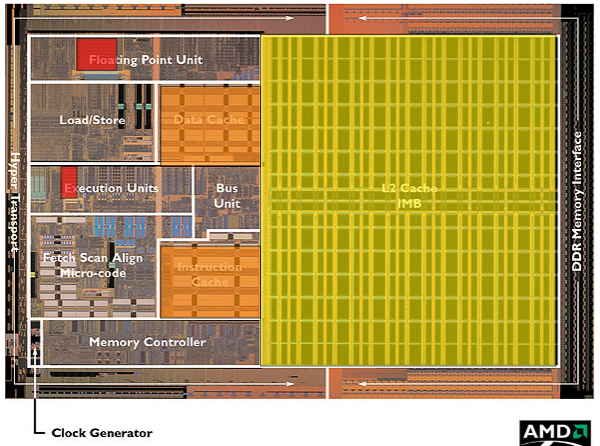


Image: approx 2004 AMD press image of Opteron die

64

2004 CPU



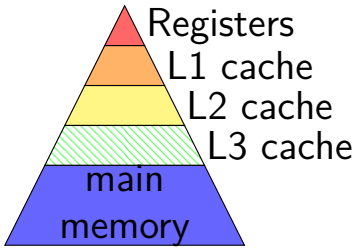
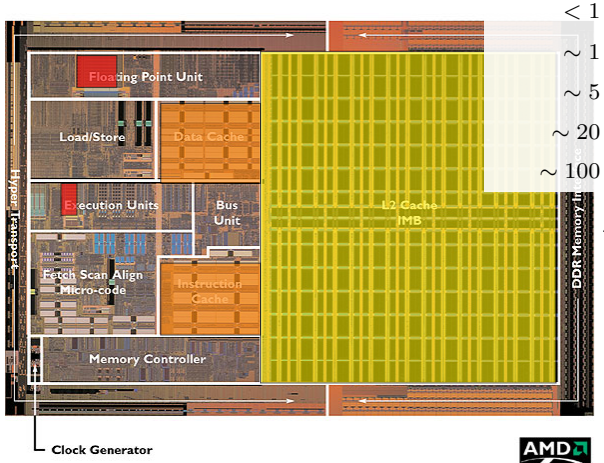


Image: approx 2004 AMD press image of Opteron die

64

2004 CPU



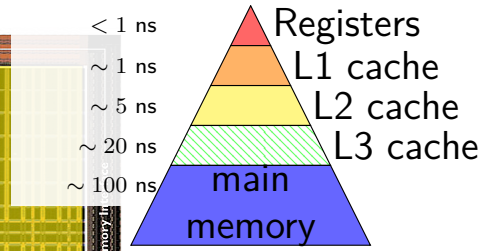
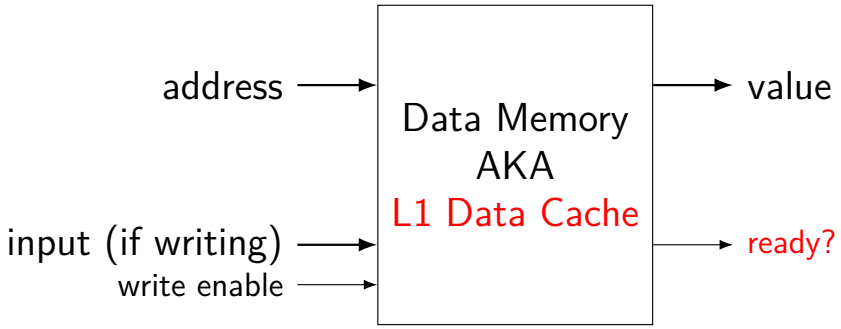


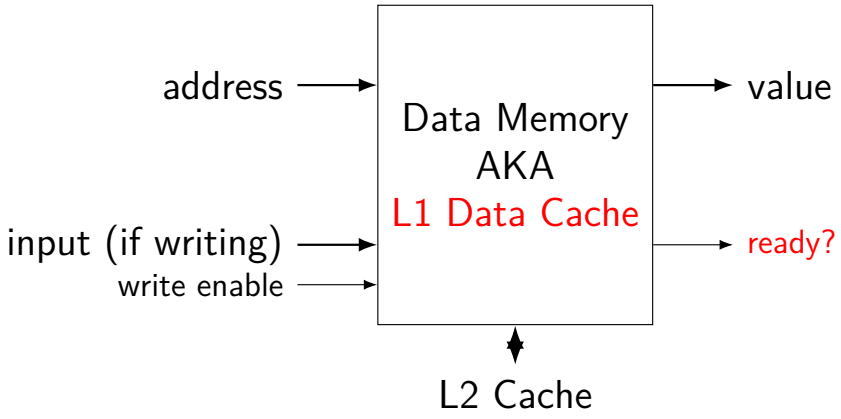
Image: approx 2004 AMD press image of Opteron die

64

cache: real memory



cache: real memory



cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

Data: Cantin and Hill, "Cache Performance for SPEC CPU2000 Benchmarks"
http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

Data: Cantin and Hill, "Cache Performance for SPEC CPU2000 Benchmarks"
http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/

cache optimizations

	miss rate	hit time	miss penalty
increase cache size	better	worse	—
increase associativity	better	worse	worse
increase block size	depends	worse	worse
add secondary cache	—	—	better
write-allocate	better	—	worse?
writeback	better	—	worse?
LRU replacement	better	?	worse?

$$\text{total time} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

Exam 1

What's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```


What's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

What’s in those files?

hello.c
```
#include <stdio.h>
int main(void) {
  puts("Hello,␣World!");
  return 0;
}
```

hello.s
```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

text (code) segment:
```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```


What’s in those files?

hello.c
```
#include <stdio.h>
int main(void) {
  puts("Hello,␣World!");
  return 0;
}
```

hello.s
```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

text (code) segment:
```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```
data segment:
```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```


What’s in those files?

hello.c
```
#include <stdio.h>
int main(void) {
  puts("Hello,␣World!");
  return 0;
}
```

hello.s
```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

text (code) segment:
```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```
data segment:
```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```


What’s in those files?

hello.c
```
#include <stdio.h>
int main(void) {
  puts("Hello,␣World!");
  return 0;
}
```

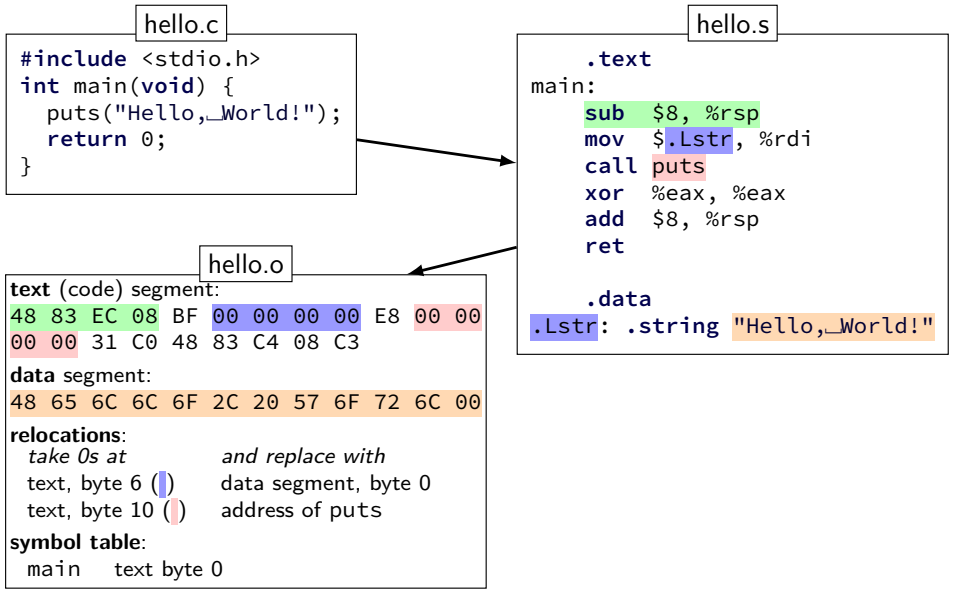
hello.s
```
    .text
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello,␣World!"
```

hello.o

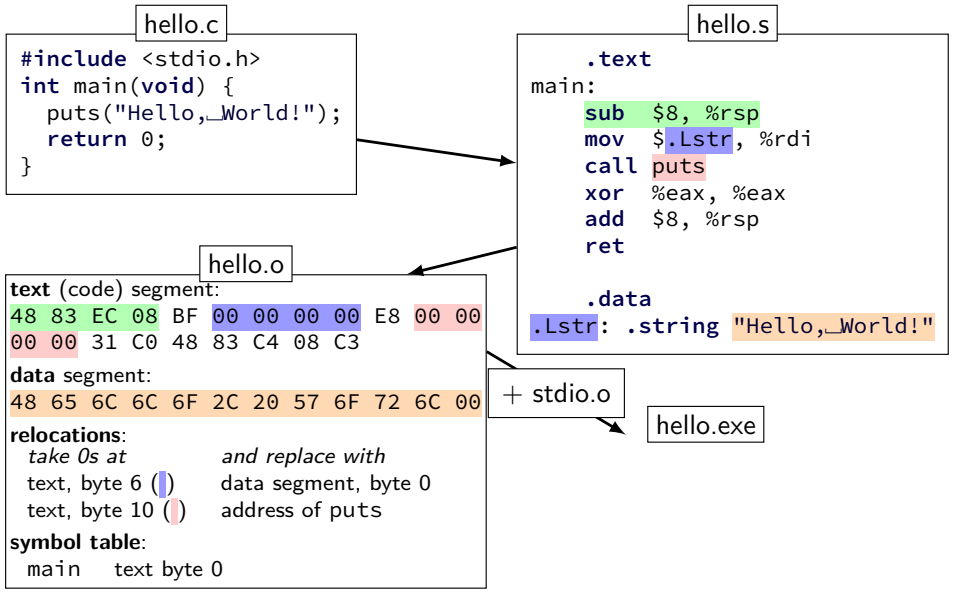
text (code) segment:
```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```
data segment:
```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```
relocations:

take 0s at	*and replace with*
text, byte 6 ( )	data segment, byte 0
text, byte 10 ( )	address of puts

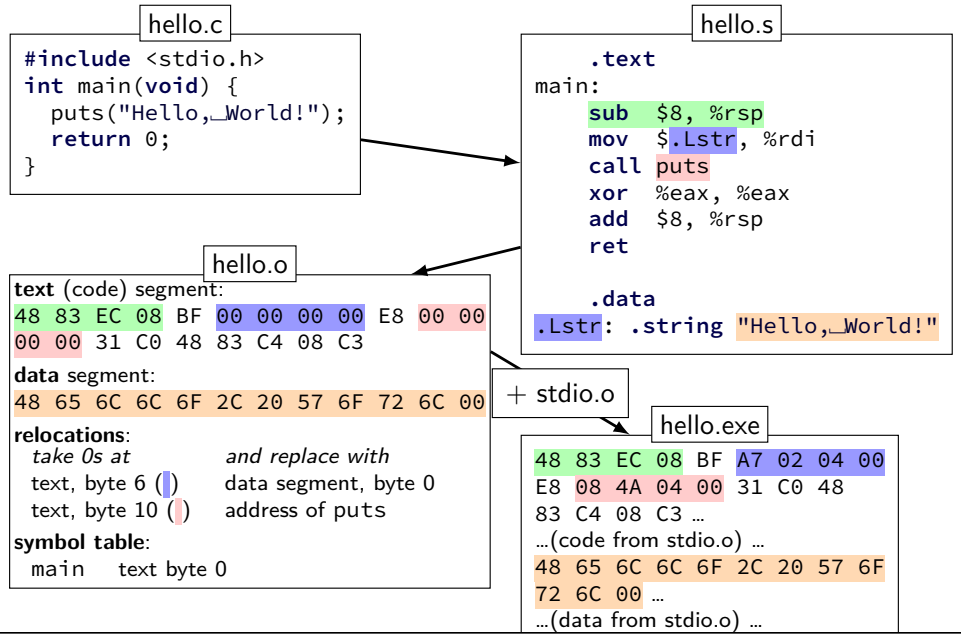
What's in those files?



What's in those files?



What's in those files?



C arrays/pointers

```
TYPE array[100];
TYPE *x = array;
    /* x points to array[0] */

x[0] == *x == *(x + 0)
x[1] == *(x + 1)

x = array + 4;
    /* x points to array[4] */

sizeof(x)      == sizeof(TYPE *) == sizeof(void *
sizeof(array)  == sizeof(TYPE) * 100
sizeof(*x)     == sizeof(TYPE)
sizeof(*array) == sizeof(TYPE)
```

typedef struct (2)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;

// same as:
typedef struct other_name_for_rational {
    int numerator;
    int denominator;
} rational;
```


typedef struct (2)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;

// same as:
typedef struct other_name_for_rational {
    int numerator;
    int denominator;
} rational;
```

71

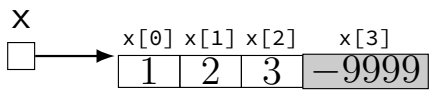
typedef struct (2)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;

// same as:
typedef struct other_name_for_rational {
    int numerator;
    int denominator;
} rational;

// almost the same as:
typedef struct {
    int numerator;
    int denominator;
} rational;
```

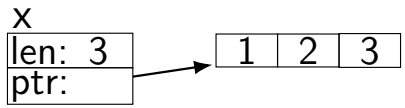
71

lists homework

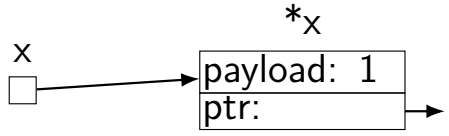
```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```




```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```




```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```

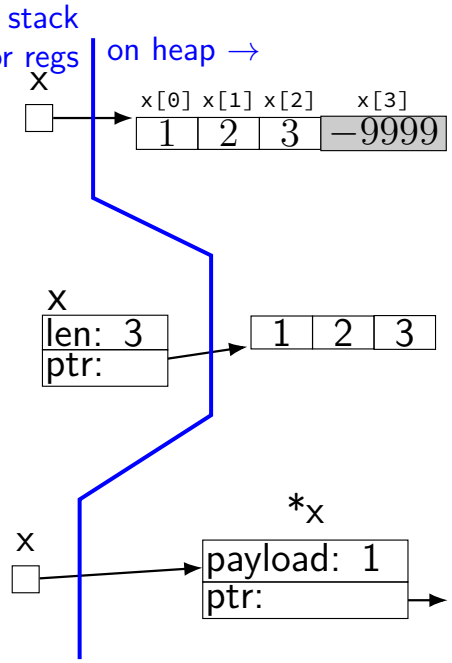


lists homework

```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```

```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```

```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```



undefined behavior

C has a standard

defines what "C" is

doesn't specify everything:

- signed integer overflow
- out-of-bounds array access
- shifts by more than type width
- writing to string constants

compilers choose different things each time

- example: optimize away handling of overflow

undefined behavior example (2)

```
int test(int number) {
    return (number + 1) > number;
}
```

Optimized:

```
test:
    movl    $1, %eax        ; eax ← 1
    ret
```

Less optimized:

```
test:
    leal    1(%rdi), %eax ; eax ← rdi + 1
    cmpl    %eax, %edi
    setl    %al             ; al ← eax < edi
    movzbl  %al, %eax       ; eax ← al
    ret
```


x86-64 calling convention

registers for first 6 arguments:

- `%rdi` (or `%edi` or `%di`, etc.), then
- `%rsi` (or `%esi` or `%si`, etc.), then
- `%rdx` (or `%edx` or `%dx`, etc.), then
- `%rcx` (or `%ecx` or `%cx`, etc.), then
- `%r8` (or `%r8d` or `%r8w`, etc.), then
- `%r9` (or `%r9d` or `%r9w`, etc.)

rest on stack

return value in `%rax`

don't memorize: Figure 3.28 in book

AT&T syntax in one slide

destination last

`()` means value in memory

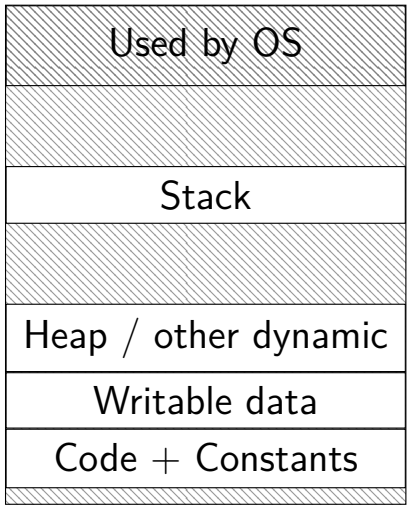
`disp(base, index, scale)` same as
`memory[disp + base + index * scale]`

- omit disp (defaults to `0`)
- and/or omit base (defaults to `0`)
- and/or scale (defualts to `1`)

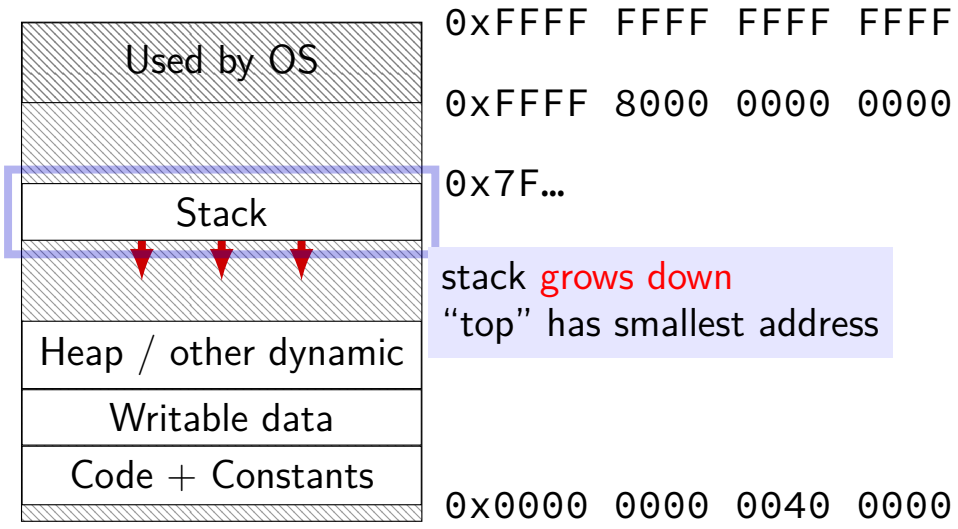
`$` means constant

plain number/label means value in memory

Program Memory (x86-64 Linux)

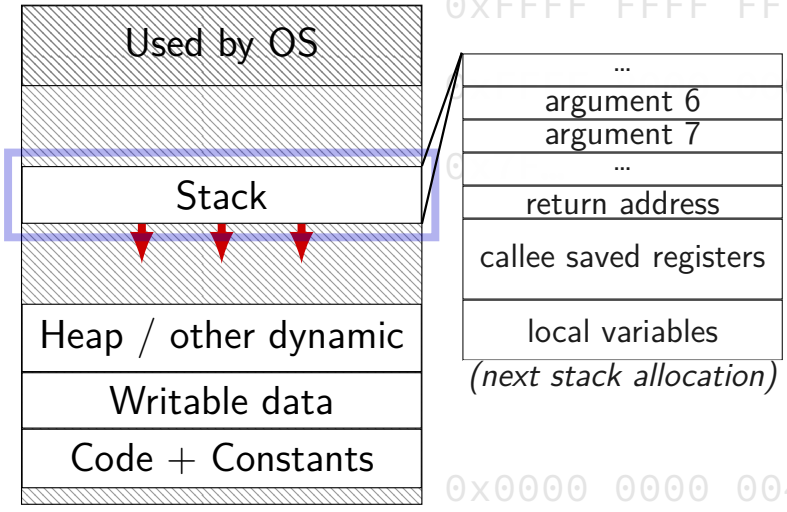


Program Memory (x86-64 Linux)

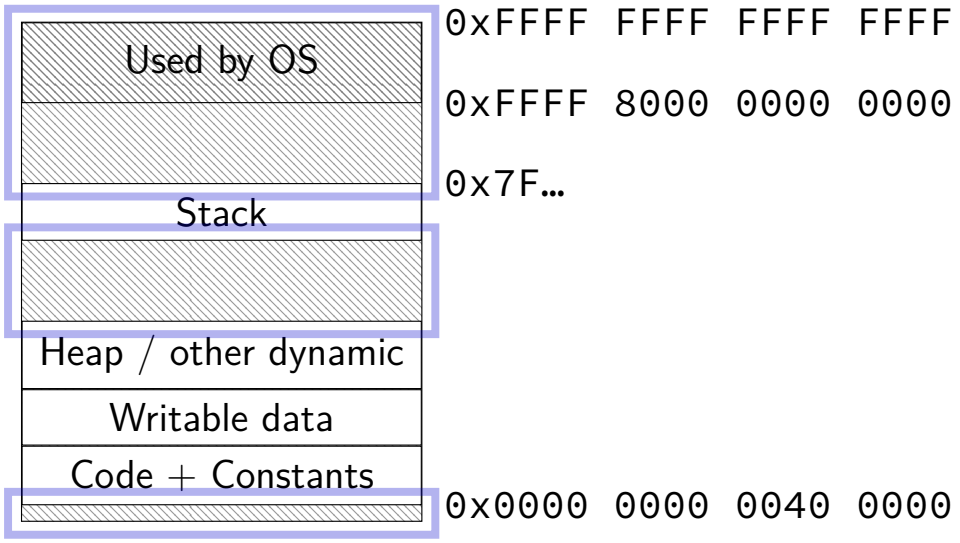


stack grows down
"top" has smallest address

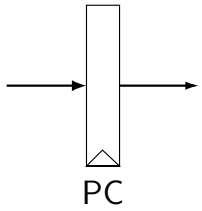
Program Memory (x86-64 Linux)



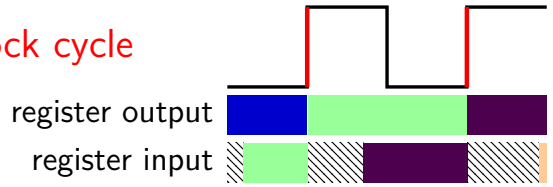
Program Memory (x86-64 Linux)



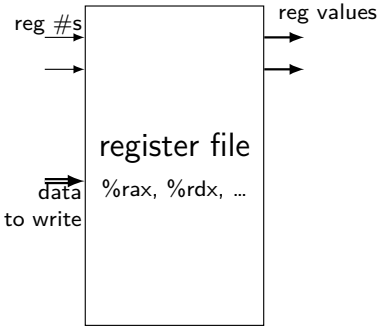
Registers



updates every clock cycle



Register file



Stages and Time

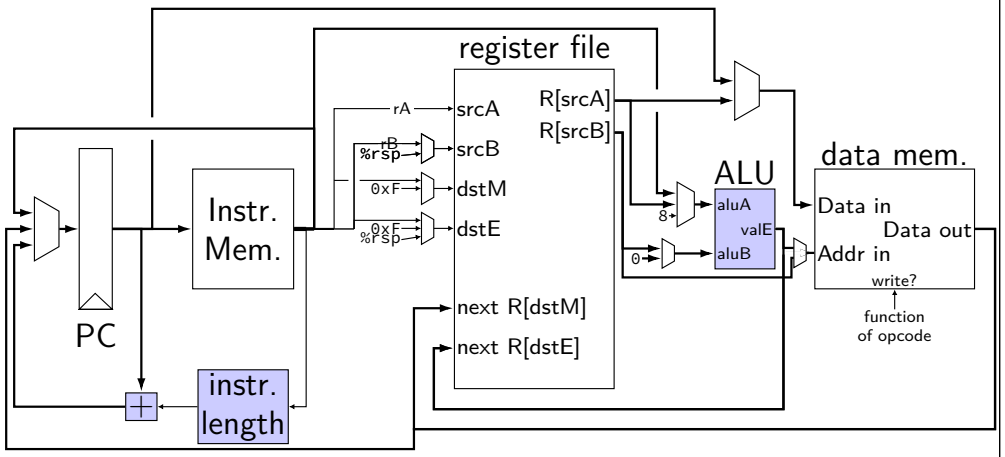
fetch / decode / execute / memory / write back / PC update

For the single-cycle design, order when these events happen `pushq %rax` instruction:

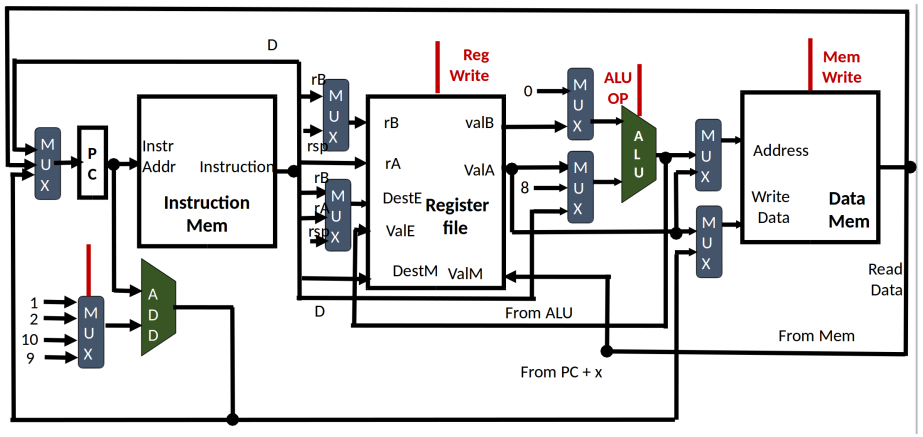
1. instruction read
2. memory changes
3. `%rsp` changes
4. PC changes

a. 1; then 2, 3, and 4 in any order
b. 1; then 2, 3, and 4 at almost the same time
c. 1; then 2; then 3; then 4
d. 1; then 3; then 2; then 4
e. 1; then 2; then 3 and 4 at almost the same time

SEQ



SEQ



Short-Circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**0**	**1**
0	0	0
1	0	1

Short-Circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**0**	**1**
0	0	0
1	0	1

Short-Circuit (&&)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**0**	**1**
0	0	0
1	0	1

Short-Circuit (&&)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**0**	**1**
0	0	0
1	0	1

Short-Circuit (&&)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

AND	**0**	**1**
0	0	0
1	0	1

Short-Circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() || one());
    printf(">␣%d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	**0**	**1**
0	0	1
1	1	1

Short-Circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	0	1
0	0	1
1	1	1

Short-Circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	0	1
0	0	1
1	1	1

84

Short-Circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	0	1
0	0	1
1	1	1

84

Short-Circuit (||)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf("> %d\n", zero() || one());
    printf("> %d\n", one() || zero());
    return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

OR	0	1
0	0	1
1	1	1

84

Left shift

```
1 << 0 == 1          0000 0001
1 << 1 == 2          0000 0010
1 << 2 == 4          0000 0100

10 << 0 == 10        0000 1010
10 << 1 == 20        0001 0100
10 << 2 == 40        0010 1000
```


Left shift

```
1 << 0 == 1          0000 0001
1 << 1 == 2          0000 0010
1 << 2 == 4          0000 0100

10 << 0 == 10        0000 1010
10 << 1 == 20        0001 0100
10 << 2 == 40        0010 1000
```

$$x \texttt{ << } y = x \times 2^y$$

Right shift

Undefined: ~~`x << -1`~~ Instead: `x >> 1`

```
1 >> 0 == 1          0000 0001
1 >> 1 == 0          0000 0000
1 >> 2 == 0          0000 0000

10 >> 0 == 10        0000 1010
10 >> 1 == 5         0000 0101
10 >> 2 == 2         0000 0010
```


Right shift

Undefined: ~~`x << -1`~~ Instead: `x >> 1`

```
1 >> 0 == 1          0000 0001
1 >> 1 == 0          0000 0000
1 >> 2 == 0          0000 0000

10 >> 0 == 10        0000 1010
10 >> 1 == 5         0000 0101
10 >> 2 == 2         0000 0010
```

$$x \texttt{ >> } y = \lfloor x \times 2^{-y} \rfloor$$

Shifts and negative numbers

`-10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Shifts and negative numbers

`-10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Option 1: binary `1111 … 1011` = $-5 = -10 \times 2^{-k}$

copy sign bit

Option 2: binary `0111 … 1011` = $2^{31} - 5$

always use zero

87

Shifts and negative numbers

`-10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Option 1: binary `1111 … 1011` = $-5 = -10 \times 2^{-k}$

copy sign bit

arithmetic

Option 2: binary `0111 … 1011` = $2^{31} - 5$

always use zero

logical

87

Shifts and negative numbers

`-10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Option 1: binary `1111 … 1011` = $-5 = -10 \times 2^{-k}$

copy sign bit

arithmetic

Option 2: binary `0111 … 1011` = $2^{31} - 5$

always use zero

logical

87

Typical RISC ISA properties

theme: simpler to implement

fewer, simpler instructions

seperate instructions to access memory

fixed-length instructions

more registers

no "loops" within single instructions

no instructions with two memory operands

few addressing modes

Y86-64: Simple condition codes (1)

If `%r9` is -1 and `%r10` is 1:

`subq %r10, %r9`

r9 becomes $-1 - (1) = -2$.

- SF $= 1$ (negative)
- ZF $= 0$ (not zero)

`andq %r10, %r10`

r10 becomes 1

- SF $= 0$ (non-negative)
- ZF $= 0$ (not zero)

Y86-64: Using condition codes

`subq SECOND, FIRST` (value = FIRST - SECOND)

`j__` or `cmov__`	condition code bit test	value test
`le`	SF $= 1$ or ZF $= 1$	value ≤ 0
`l`	SF $= 1$	value < 0
`e`	ZF $= 1$	value $= 0$
`ne`	ZF $= 0$	value $\neq 0$
`ge`	SF $= 0$	value ≥ 0
`g`	SF $= 0$ and ZF $= 0$	value > 0