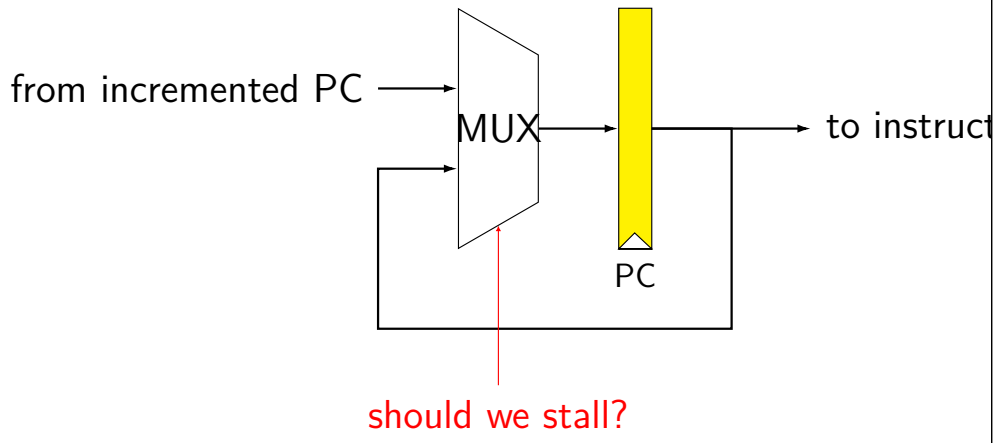
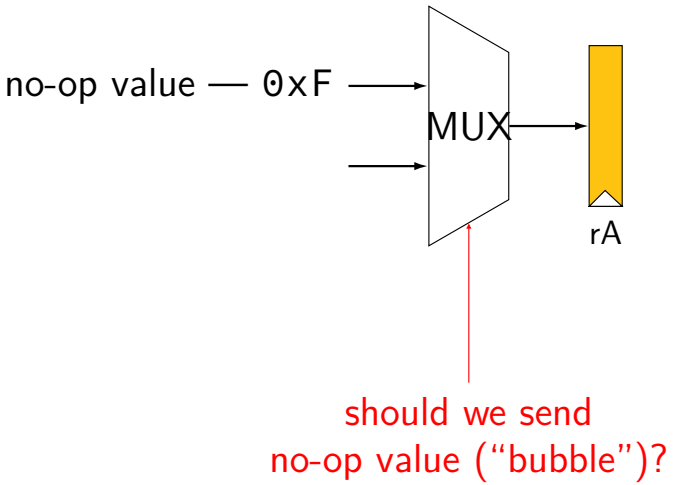
Midterm 2 Review

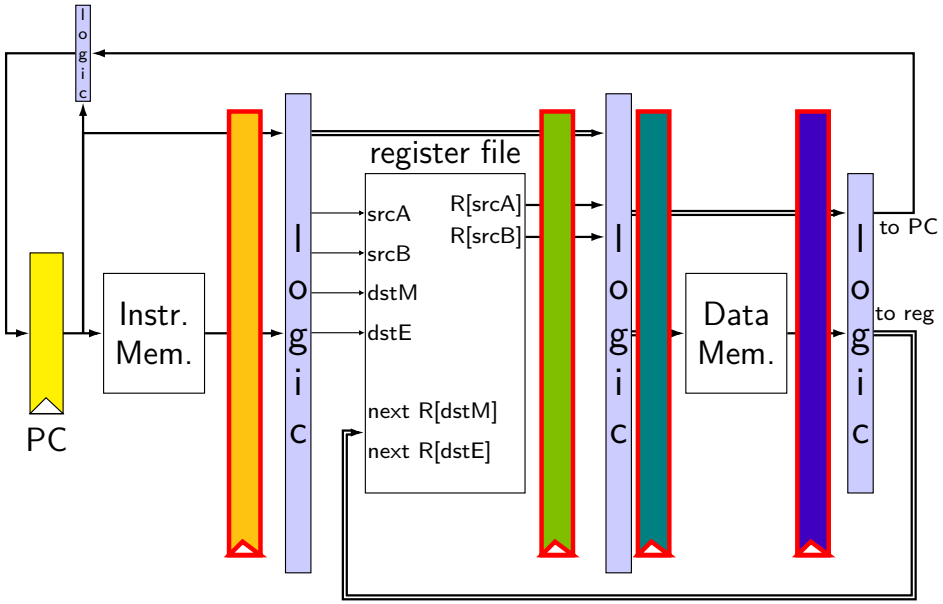
fetch/fetch logic — advance or not



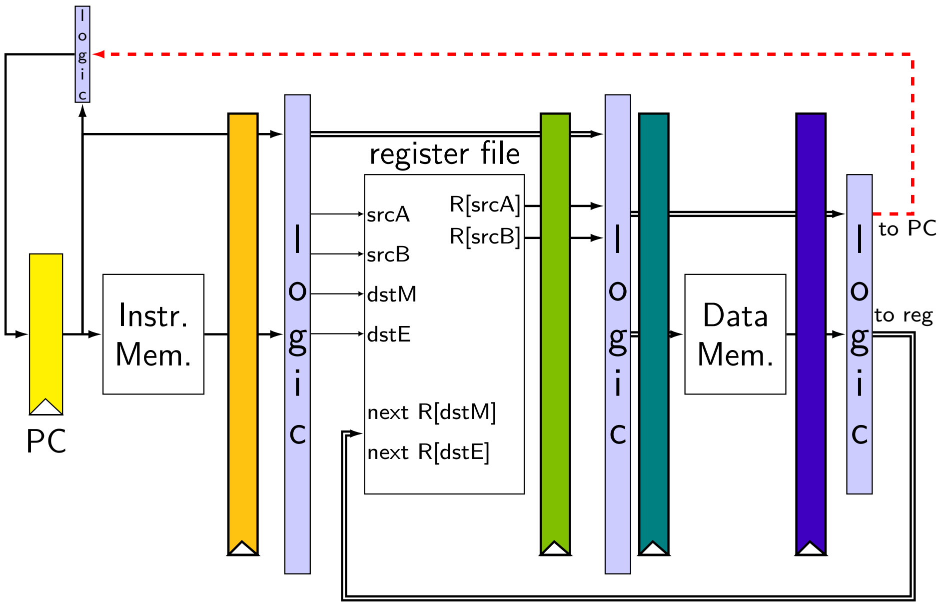
fetch/decode logic — bubble or not



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

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);

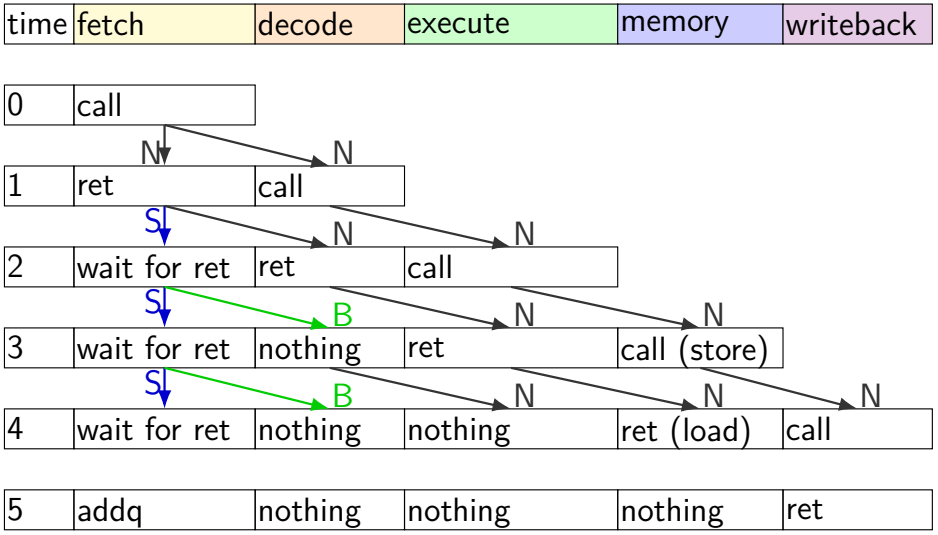
ret stall

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);

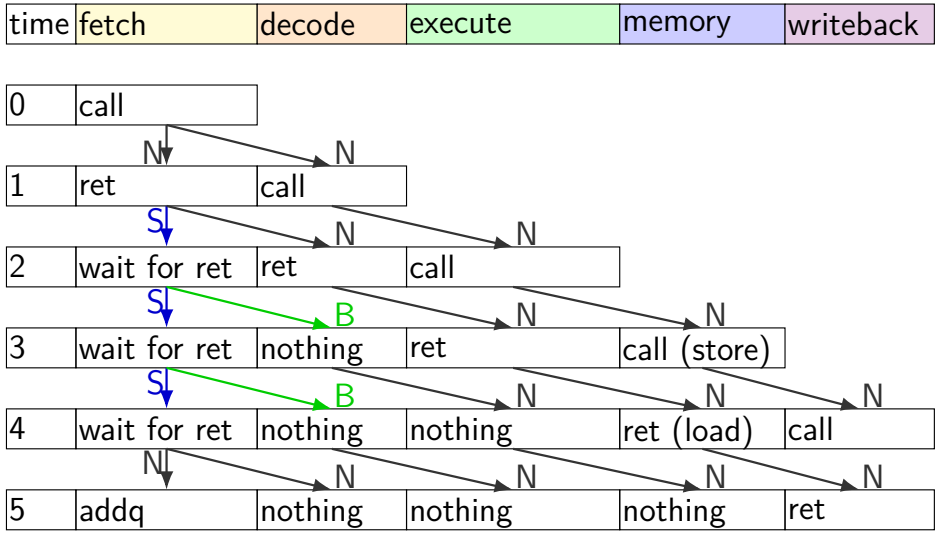
ret stall



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

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

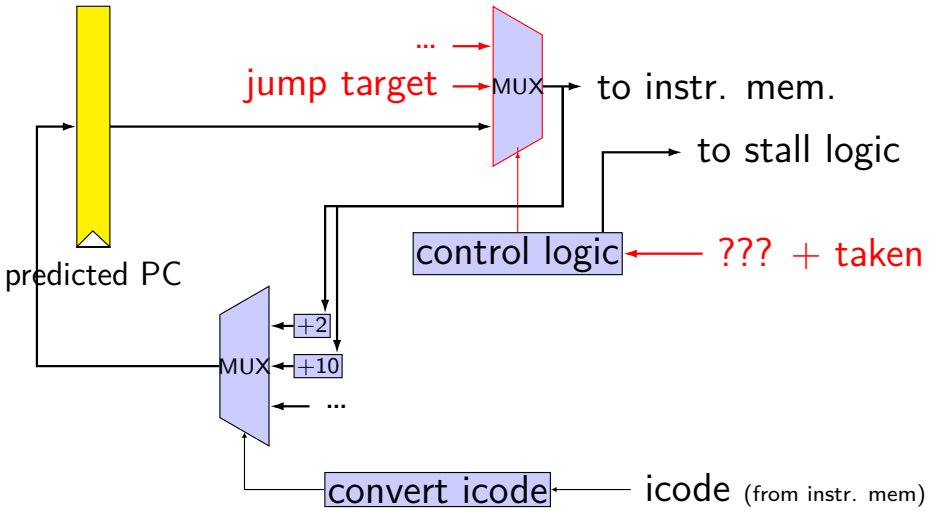
ret stall



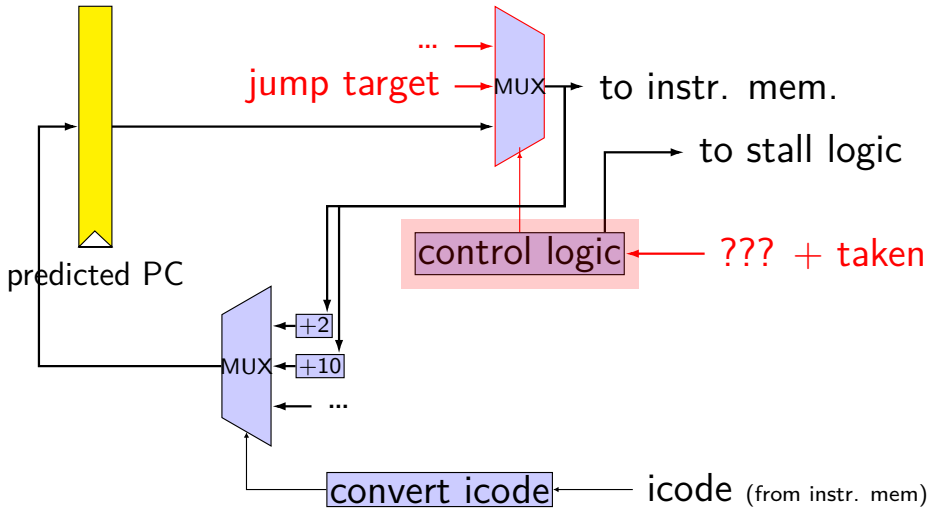
stall (S) = keep old value; normal (N) = use new value

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

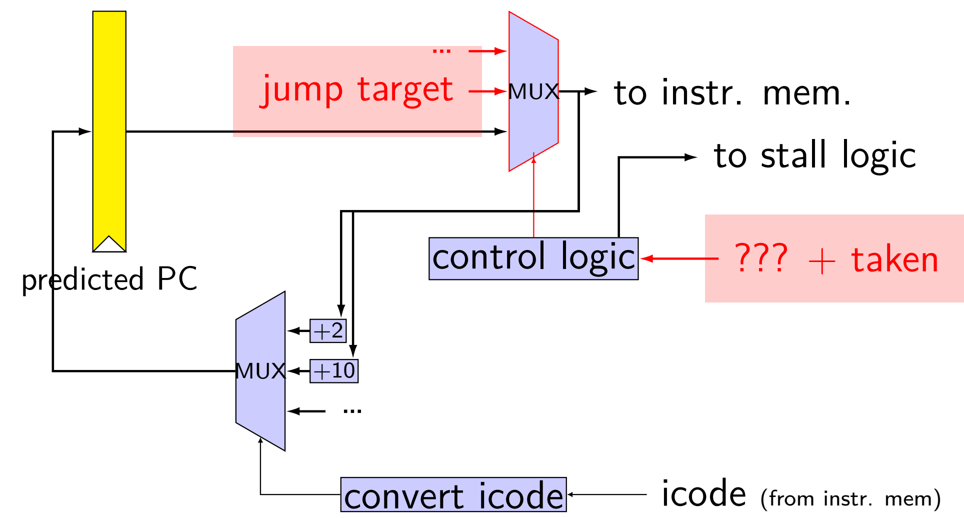
PC update (rearranged)



PC update (rearranged)



PC update (rearranged)



squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

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

squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

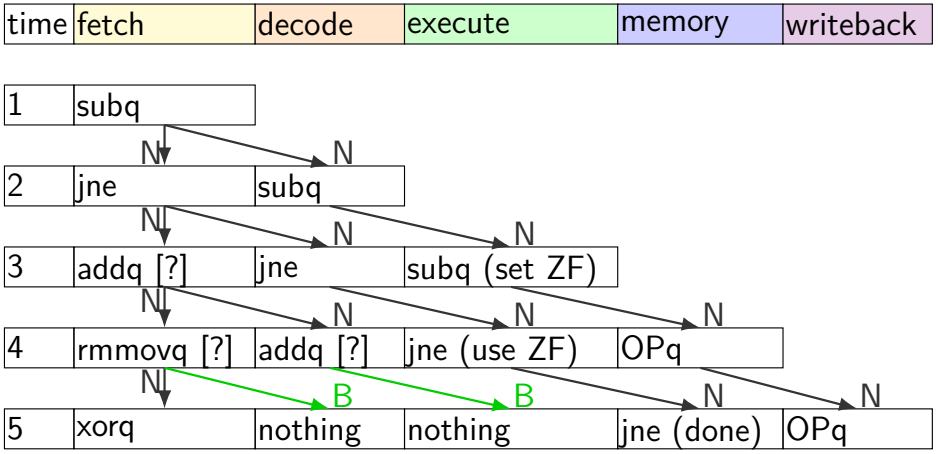
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

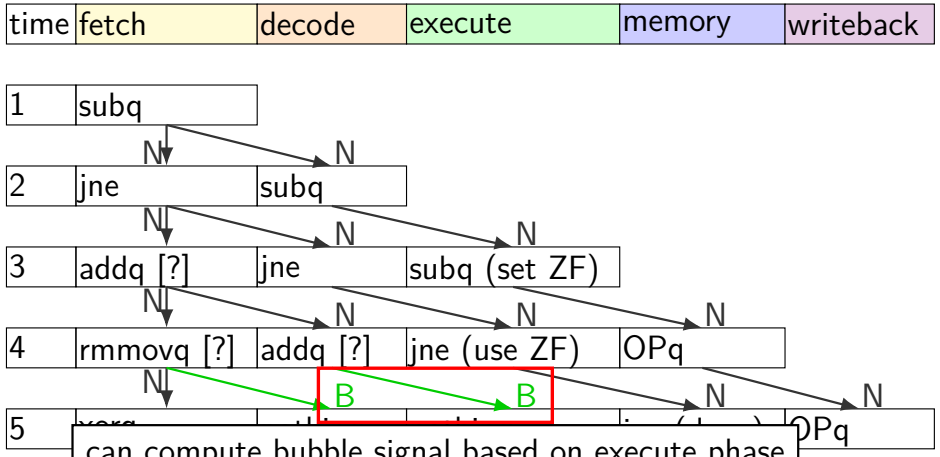
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble



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

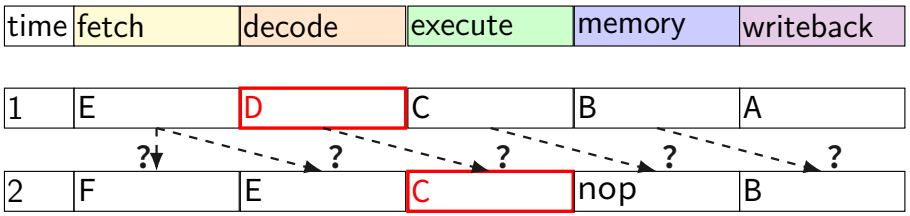
squashing with stall/bubble



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

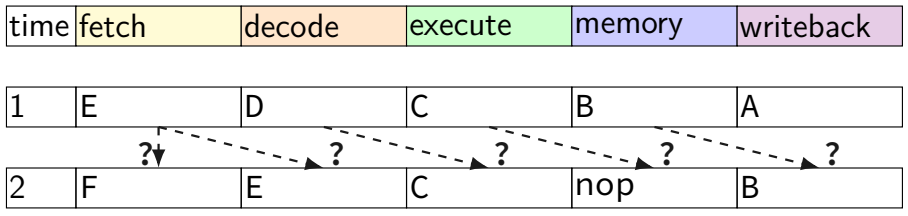
stal ew value
bubble (B) = use default (no-op);

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)

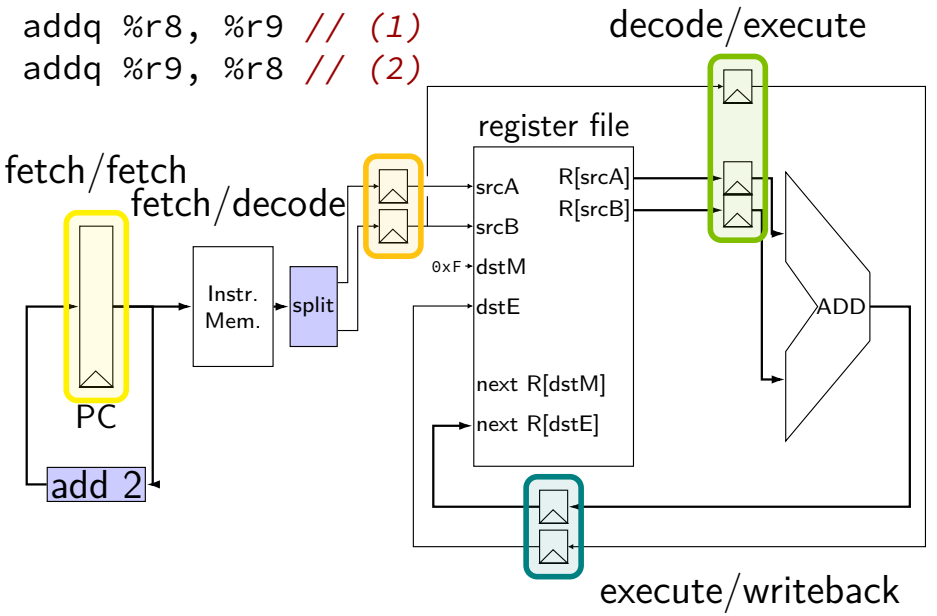
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B

N N S B N

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

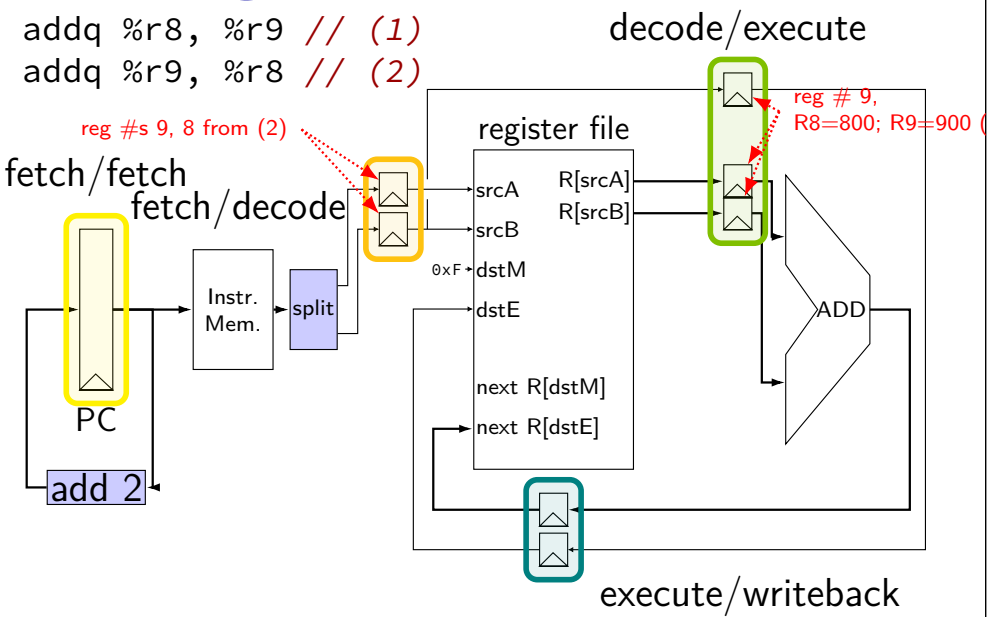
forwarding

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



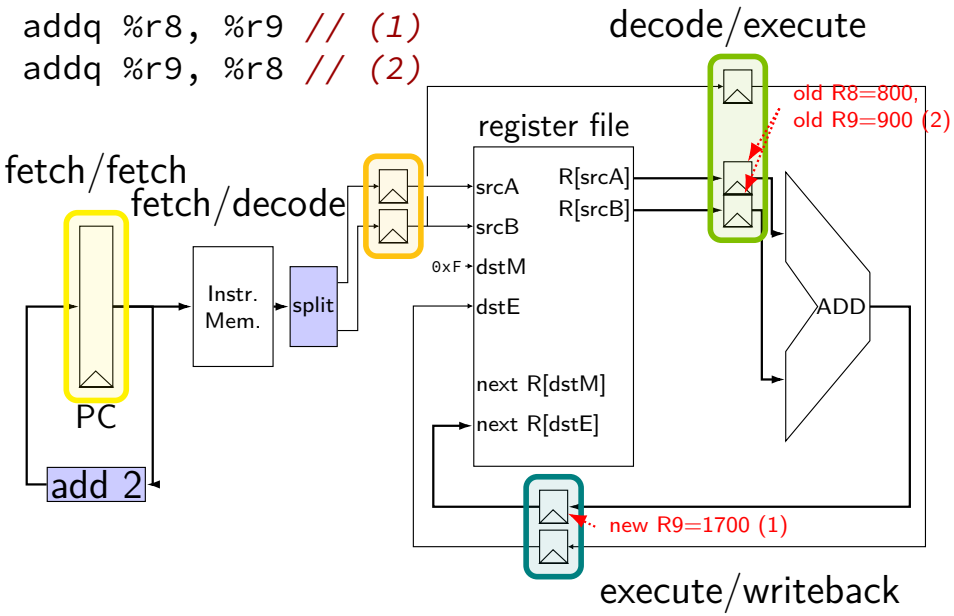
forwarding

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



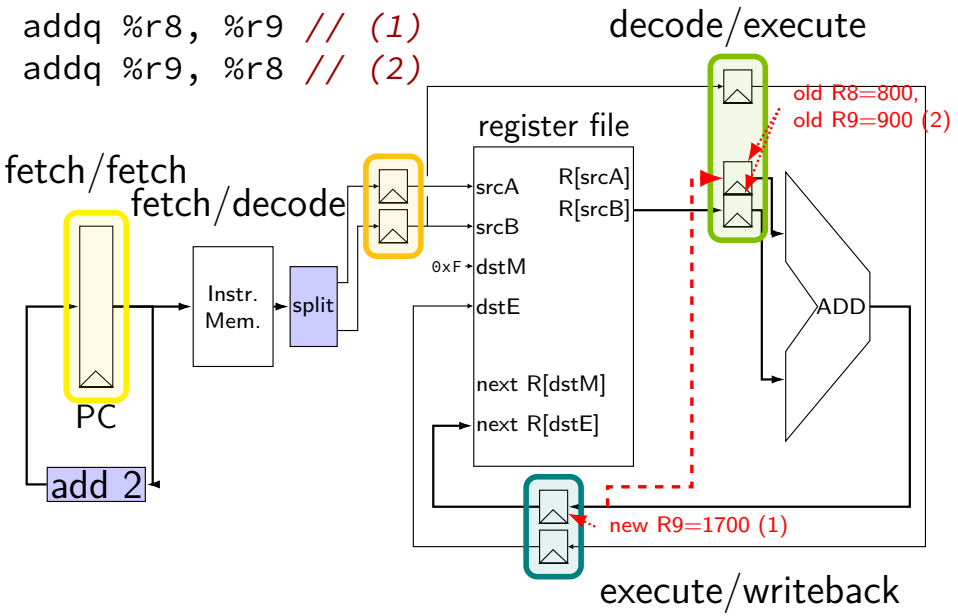
forwarding

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



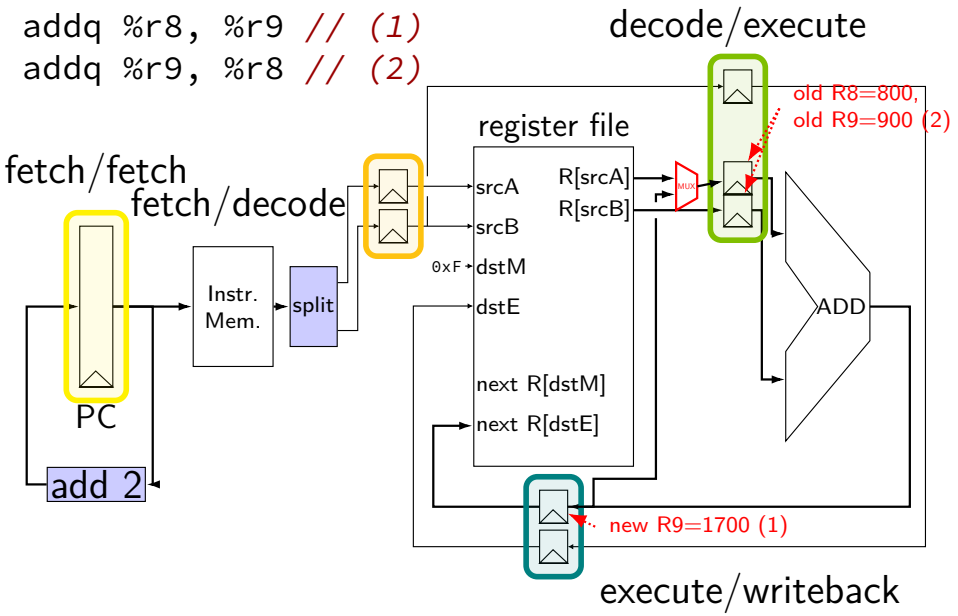
forwarding

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



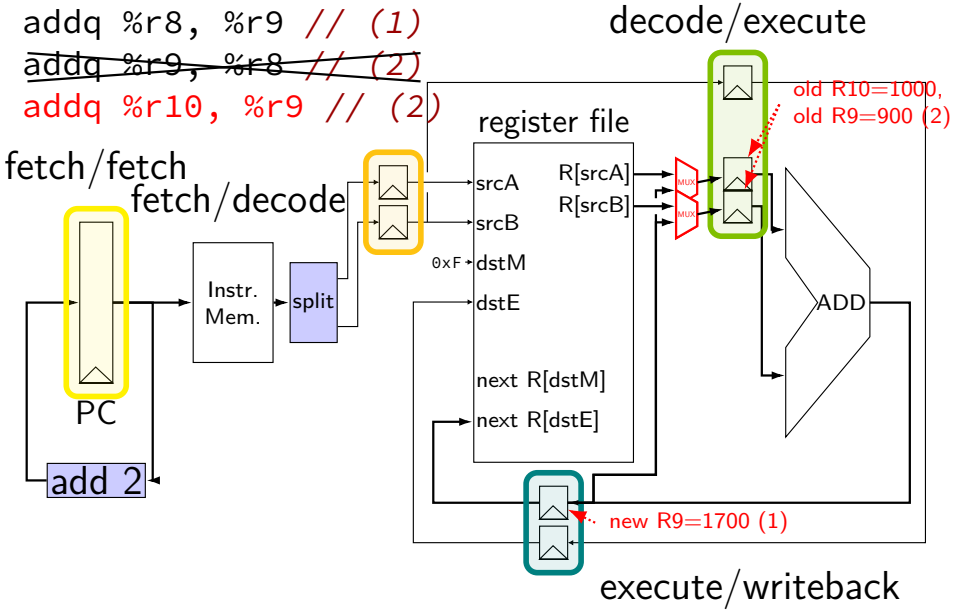
forwarding

```
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

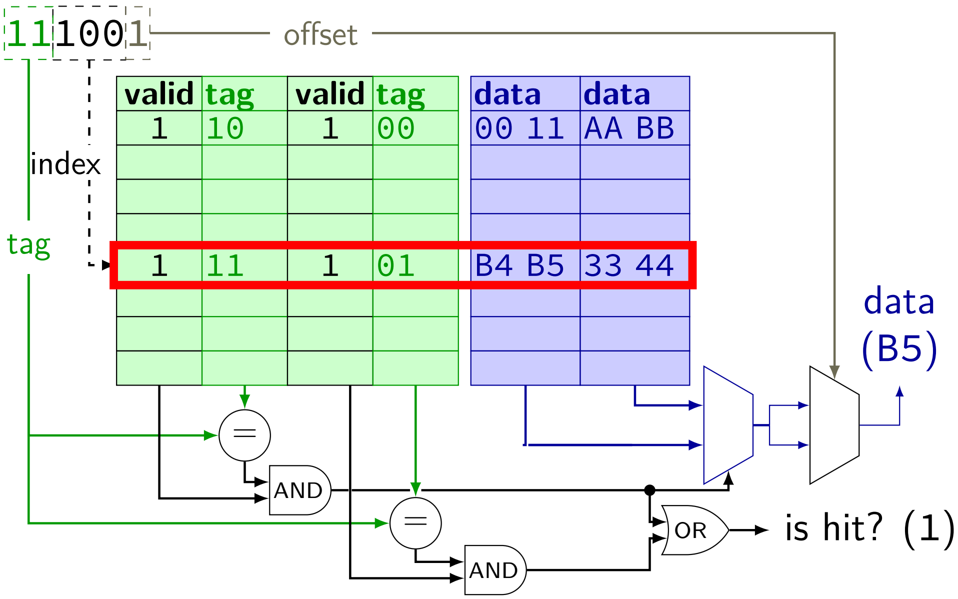
cycle #	0	1	2	3	4	5	6	7	8
`mrmovq 0(%r10), %r8`	F	D	E	M	W				
`rmmovq %r8, 0(%r10)`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

exercise: stalls and forwarding (3)

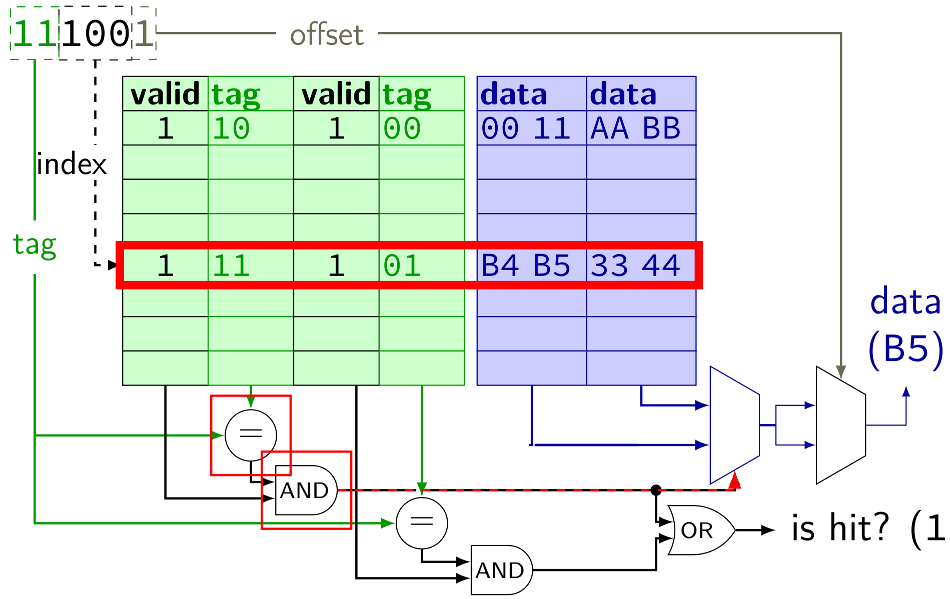
```
addq %rax, %rax
jne foo // taken
foo: mrmovq 0(%rax), %rbx
addq %rbx, %rcx
mrmovq 0(%rbx), %rcx
```

Are there stalls? Where does forwarding happen?

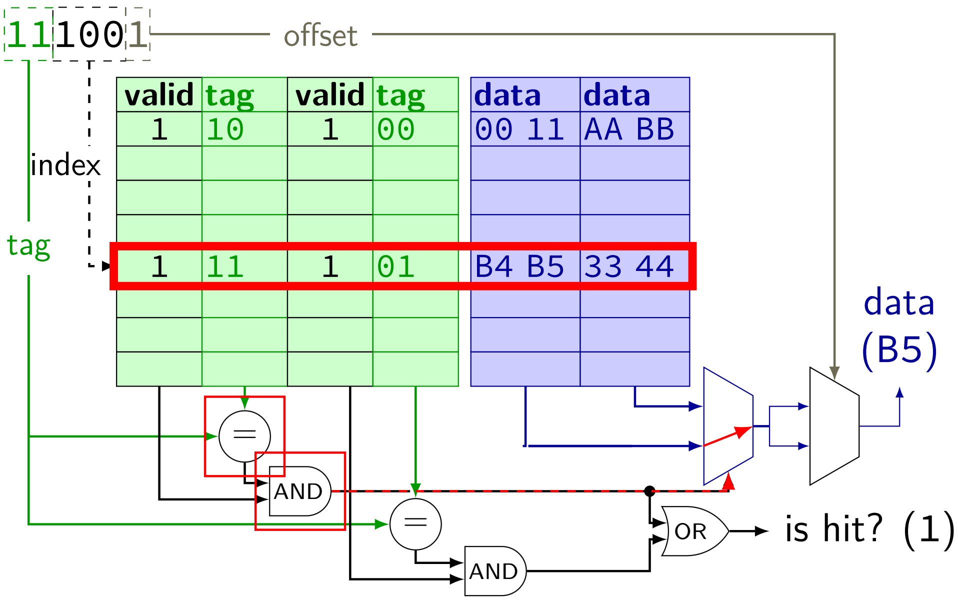
cache operation (associative)



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];
```


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 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, j, k in I by J by K block:
        B[i * N + j] += A[i * N + k]
                      * A[k * N + j];
    }
  }
}
```


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 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, j, k in I by J by K block:
        B[i * N + j] += A[i * N + k]
                      * A[k * N + j];
    }
  }
}
```

B_{ij} used K times for one miss

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 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, j, k in I by J by K block:
        B[i * N + j] += A[i * N + k]
                      * A[k * N + j];
    }
  }
}
```

A_{ik} used $> J$ times for one miss

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 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, j, k in I by J by K block:
        B[i * N + j] += A[i * N + k]
                      * A[k * N + j];
    }
  }
}
```

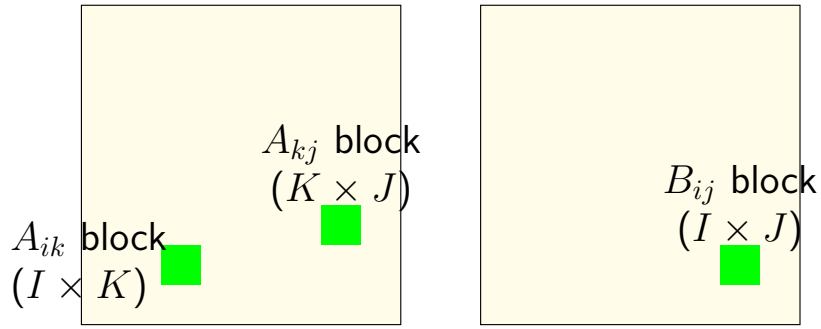
A_{kj} used I times for one miss

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 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, j, k in I by J by K block:
        B[i * N + j] += A[i * N + k]
                      * A[k * N + j];
    }
  }
}
```

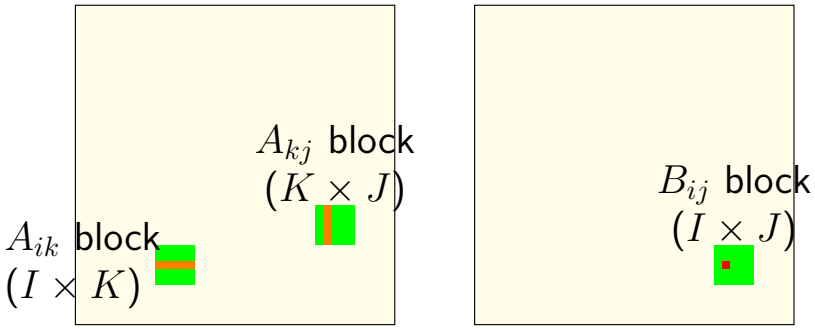
catch: $IK + KJ + IJ$ elements must fit in cache

array usage: block



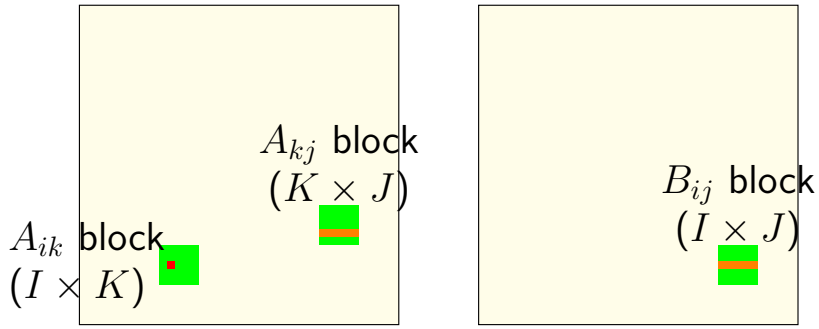
inner loop keeps "blocks" from A, B in cache

array usage: block



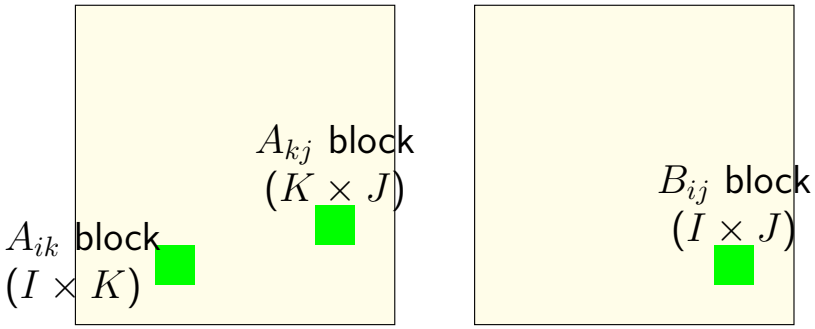
B_{ij} calculation uses strips from A
K calculations for one load (cache miss)

array usage: block



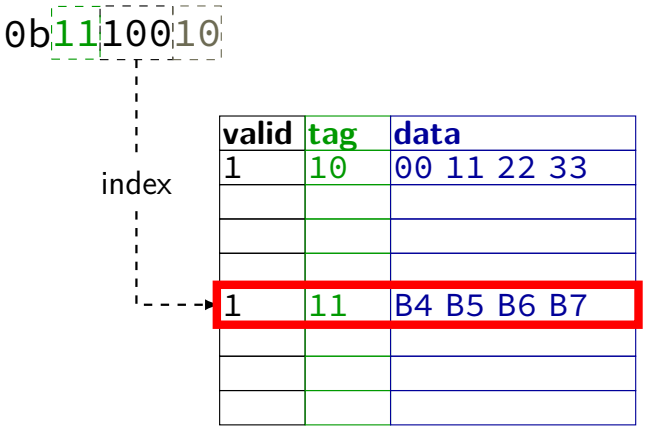
A_{ik} calculation uses strips from A, B
J calculations for one load (cache miss)

array usage: block

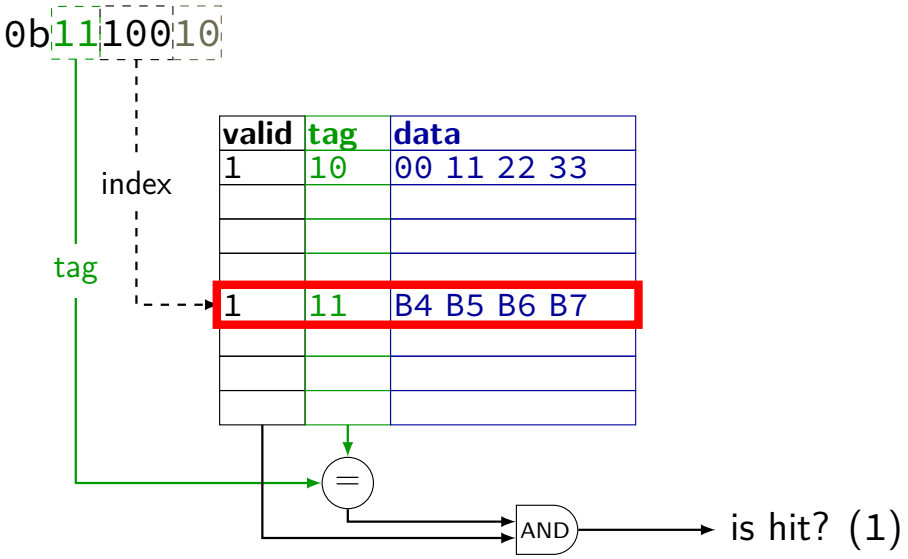


(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ loads
(assuming everything stays in cache)

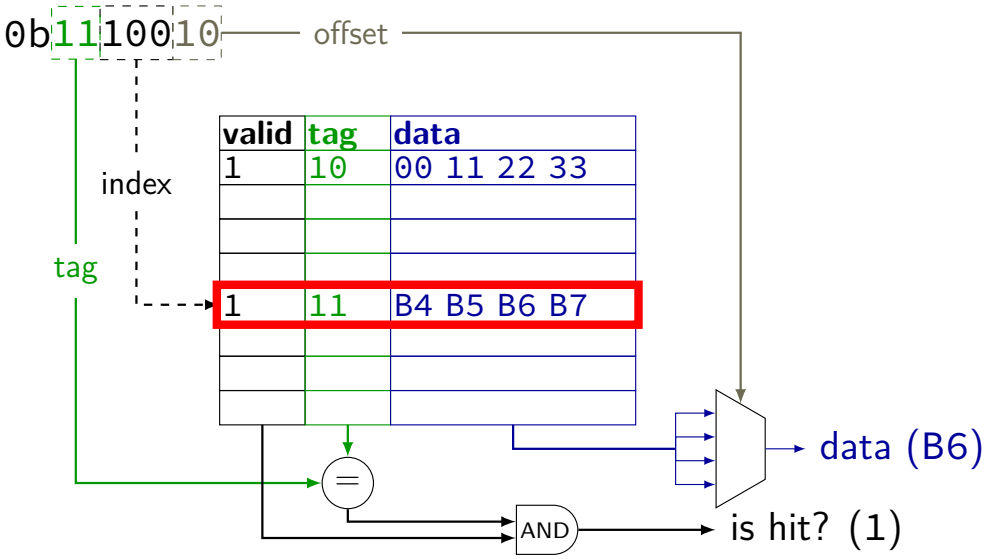
cache operation (read)



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

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

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		

$B = 2 = 2^b$ 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

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

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		

$B = 2 = 2^b$ 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)	

tag index offset

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

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		

$B = 2 = 2^b$ 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

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

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		

$B = 2 = 2^b$ 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)	

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)	

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)	

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

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 byte blocks, 2 sets

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

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 byte blocks, 2 sets

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

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

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 byte blocks, 2 sets

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

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	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

$2^3 = 8$ 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	--	-- --
011	0	--	-- --
100	0	--	-- --
101	1	00	AA BB
110	0	--	-- --
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

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

2004 CPU

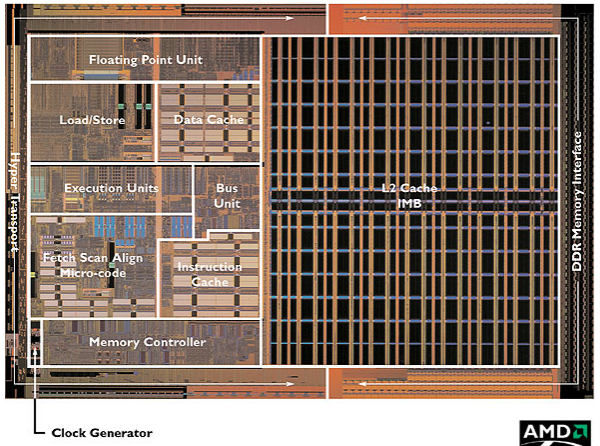


Image: approx 2004 AMD press image of Opteron die

2004 CPU

Registers

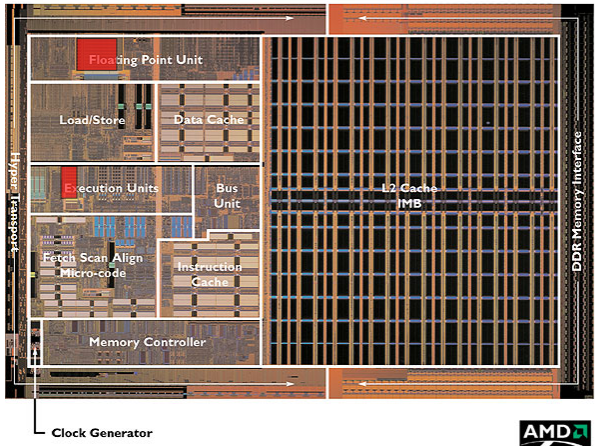


Image: approx 2004 AMD press image of Opteron die

23

2004 CPU

Registers
L1 cache

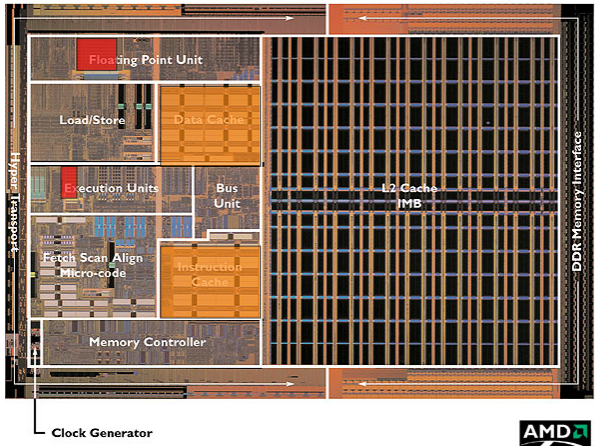


Image: approx 2004 AMD press image of Opteron die

23

2004 CPU

Registers
L1 cache
L2 cache

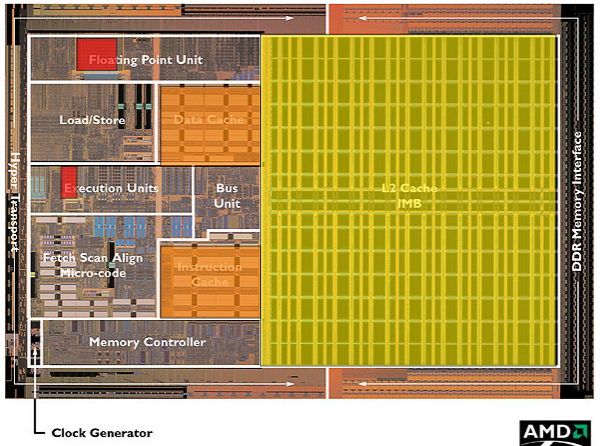
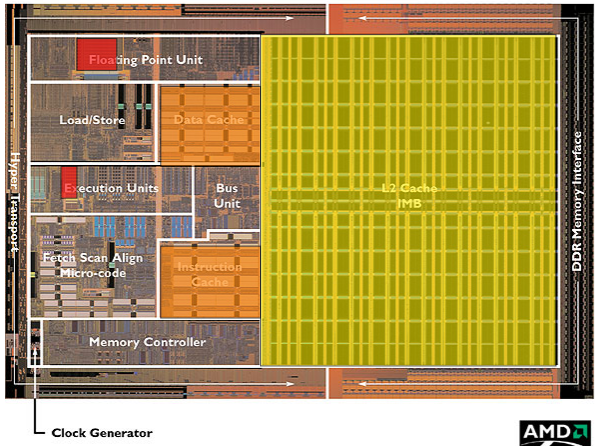


Image: approx 2004 AMD press image of Opteron die

23

2004 CPU



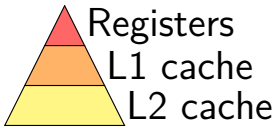
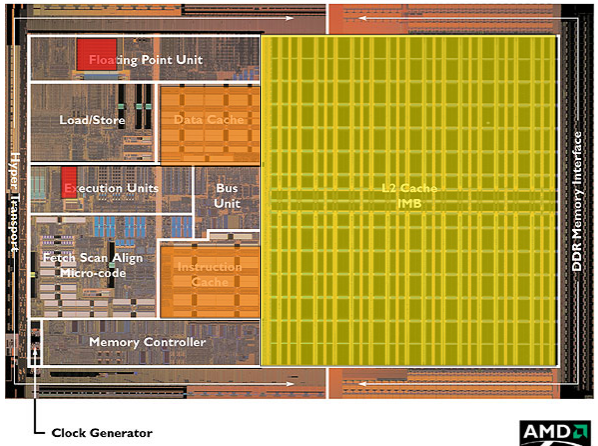


Image: approx 2004 AMD press image of Opteron die

2004 CPU



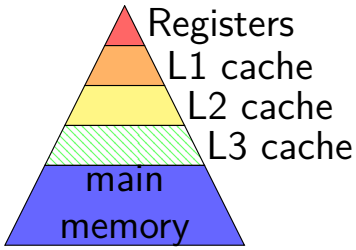
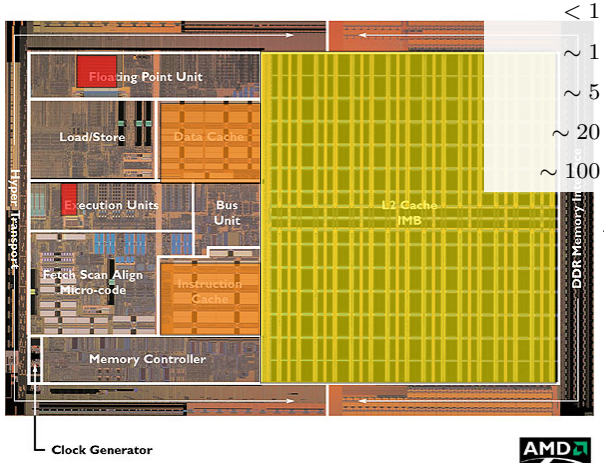


Image: approx 2004 AMD press image of Opteron die

2004 CPU



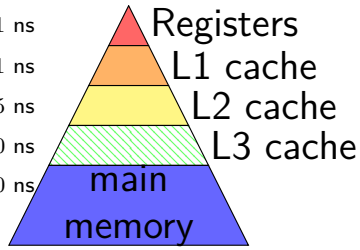
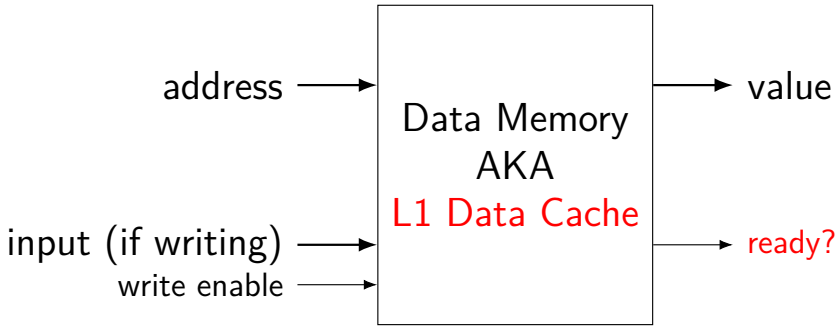
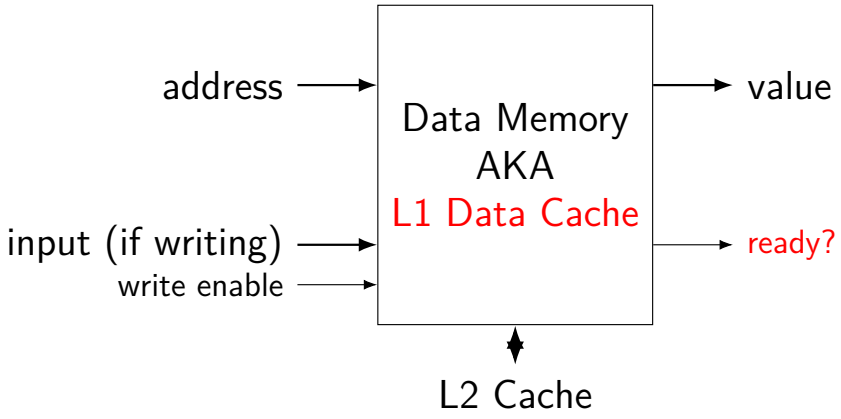


Image: approx 2004 AMD press image of Opteron die

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?

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