Lab 4: Coding on Modern Processors or Say Exploiting Instruction-Level Parallelism

Hung-Wei Tseng

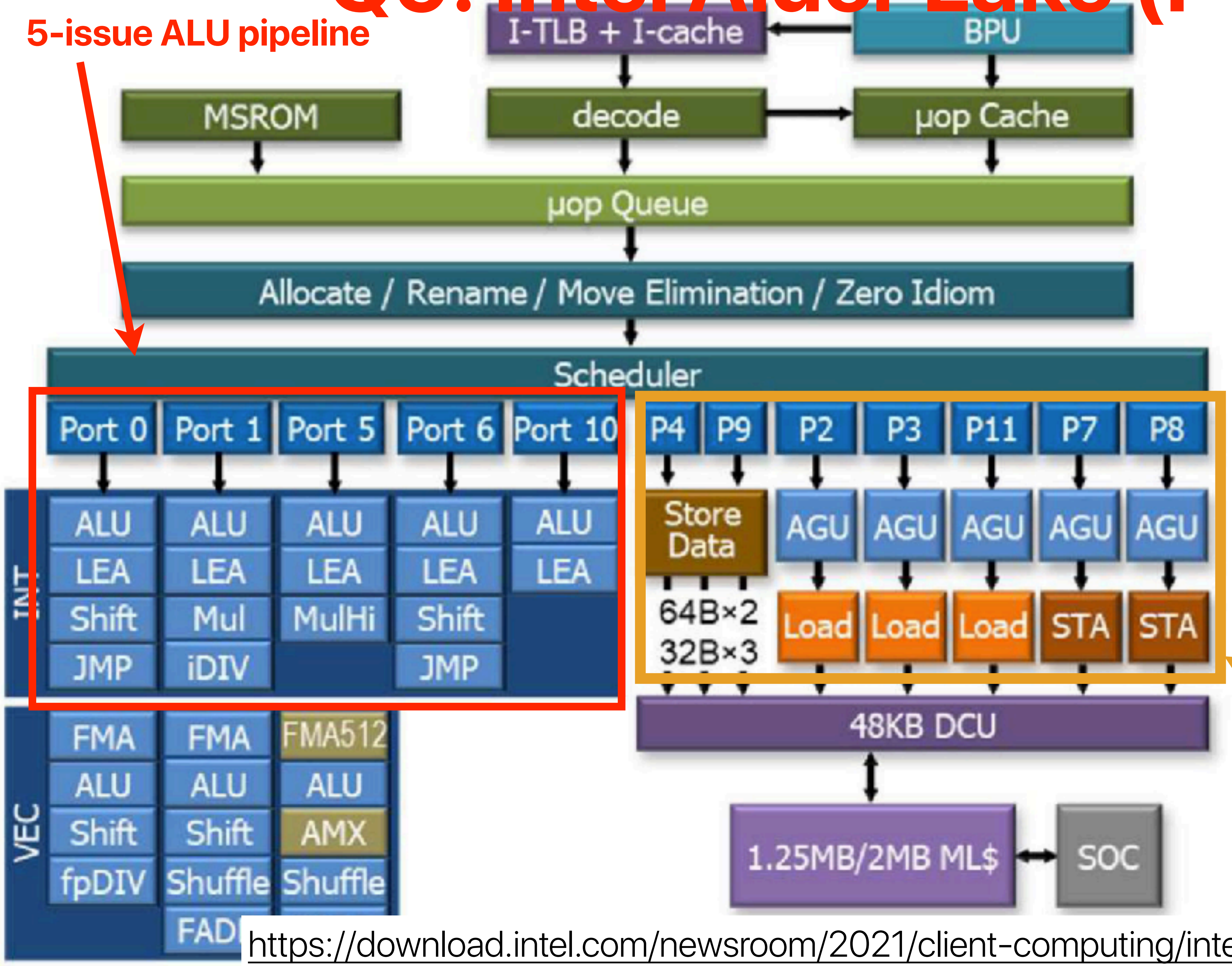
Parallelism in modern computers

- Instruction-level parallelism
  - The ability to execute multiple instructions concurrently
- Thread-level parallelism
  - The ability to execute multiple program instances concurrently
- Data-level parallelism
  - The ability to process data concurrently

Instruction-level parallelism

Q9: Intel Alder Lake (P-Core)

5-issue ALU pipeline



https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

$$MinCPI = \frac{1}{12}$$

$$MinINTInst\,.\,CPI = \frac{1}{5}$$

$$MinMEMInst\,.\,CPI = \frac{1}{7}$$

$$MinBRInst\,.\,CPI = \frac{1}{2}$$

The processor wants to maximize the throughput of memory instructions! (Q10)

7-issue memory pipeline

Compiler optimizations

Q1: Strength Reduction — Use a Simpler Operation

- Instead of multiplications and divisions, how can we compute?

`a * 43`

```
leaq   (%rdi,%rdi,4), %rax
leaq   (%rdi,%rax,4), %rax
leaq   (%rdi,%rax,2), %rax
```

`a / 43`

```
movabsq   $428994048225803526, %rdx
movq   %rdi, %rax
imulq  %rdx
sarq   $63, %rdi
movq   %rdx, %rax
subq   %rdi, %rax
```

`a % 43`

```
movabsq   $428994048225803526, %rdx
movq   %rdi, %rax
imulq  %rdx
movq   %rdi, %rcx
sarq   $63, %rcx
movq   %rdx, %rax
subq   %rcx, %rax
leaq   (%rax,%rax,4), %rdx
leaq   (%rax,%rdx,4), %rdx
leaq   (%rax,%rdx,2), %rdx
movq   %rdi, %rax
subq   %rdx, %rax
```

- form 2: `imulq s`
  - one operand is %rax
  - The other operand given in the instruction
  - product is stored in %rdx (high-order part) and %rax (low order part) → full 128-bit result

`sarq` k, Dest   Dest = Dest $\gg$ k (arithmetic)

%rdi = a
%rax = %rdi + 4*%rdi (= 5a)
%rax = %rdi + 4*%rax (=21a)
%rax = %rdi + 2*%rax (=43a)

%rdi = a
%rdx = %rdi * 428994048225803526
(= 428994048225803526a)
%rdi = %rdi >> 63 (take a's signed bit)
%rax = 428994048225803526a's higher-order part only — = 428994048225803526a >> 64 (= a/43.0000000065)
%rax = %rax - %rdi

Good reference for instruction latencies

https://uops.info/

Let's start with a program

```c
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// A function to implement bubble sort
void bubbleSort(int *arr, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
        // Last i elements are already
        // in place
        for (j = 0; j < n - 1 - i; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
// Function to print an array
void printArray(int *arr, int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d\t",arr[i]);
}
```

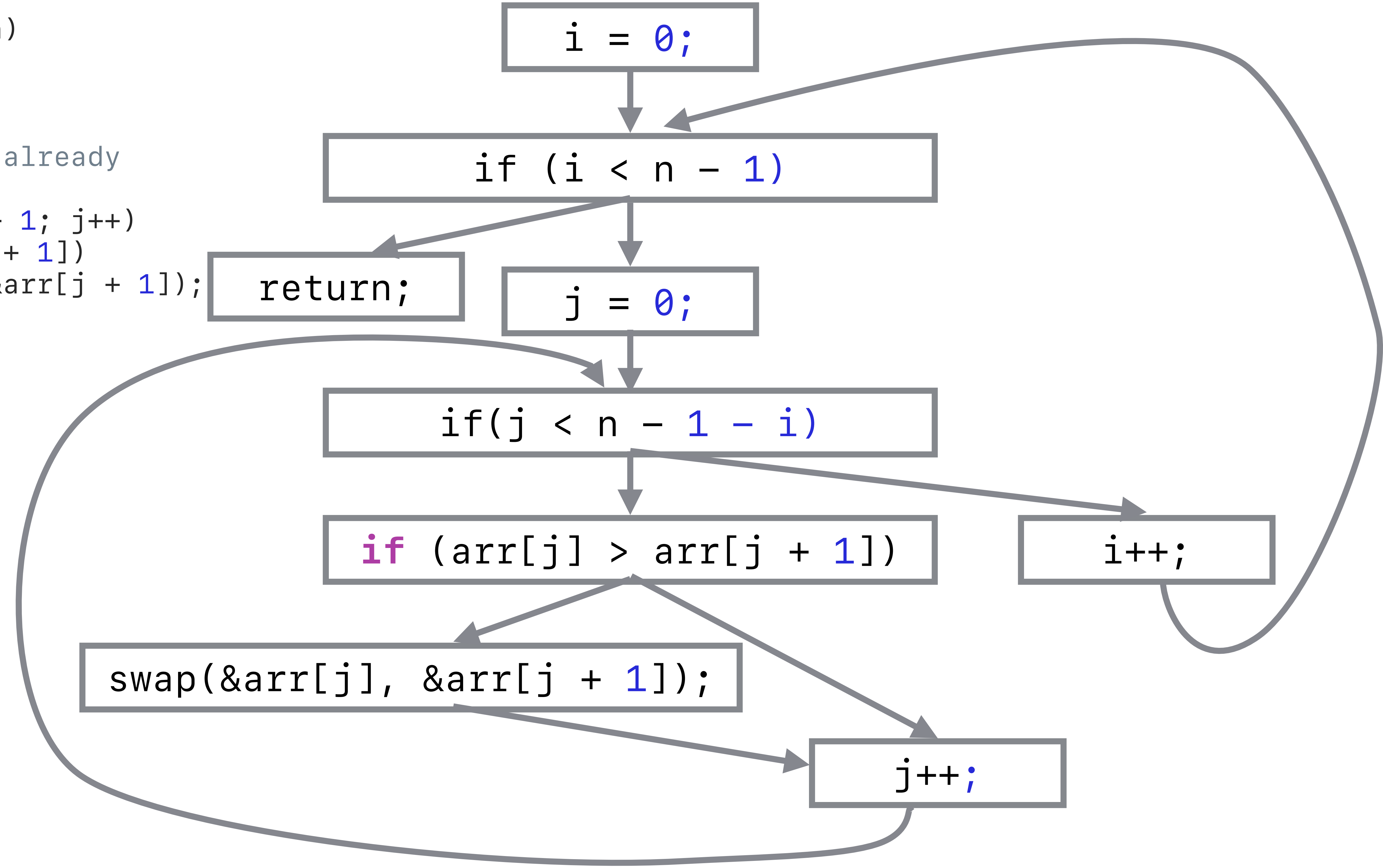
```c
int main()
{
    unsigned N = 131072;
    int *data;
    if(argc > 1)
        N = atoi(argv[1]);
    data = (int *)malloc(sizeof(int)*N);

    for (unsigned i = 0; i < N; ++i)
        data[i] = rand();
    bubbleSort(data, N);
    printArray(data, N);
    return 0;
}
```

Control flow graph

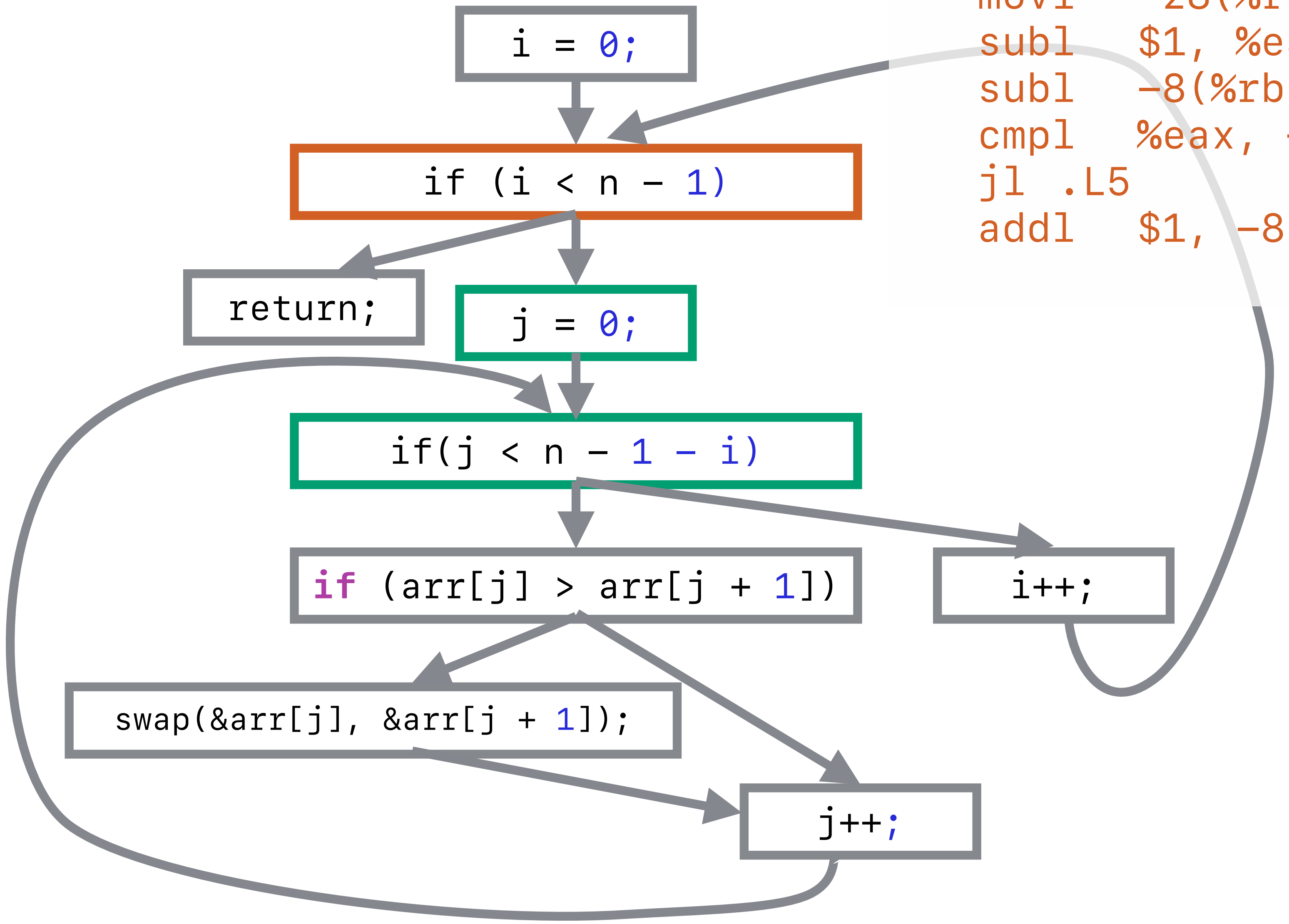
- A graph shows all possible route of executing a piece code
- A “directed” graph consists of **basic blocks**
- A **basic block** is a sequence of instructions that will always execute together
  - A sequence of instructions until we reach a “branch”
  - Compiler typically can agreesively reorder instructions within a basic block

Control flow graph

```
void bubbleSort(int *arr, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
        // Last i elements are already
        // in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
```



Control flow graph and code




```
.L3:
    movl   -28(%rbp), %eax
    subl   $1, %eax          // -28(%rbp) is n
    subl   -8(%rbp), %eax    // -8(%rbp) is n
    cmpl   %eax, -4(%rbp)
    jl  .L5
    addl   $1, -8(%rbp)
```

```
.L5:
    movl   -4(%rbp), %eax
    cltq
    leaq   0(,%rax,4), %rdx
    movq   -24(%rbp), %rax
    addq   %rdx, %rax
    movl   (%rax), %edx
    movl   -4(%rbp), %eax
    cltq
    addq   $1, %rax
    leaq   0(,%rax,4), %rcx
    movq   -24(%rbp), %rax
    addq   %rcx, %rax
    movl   (%rax), %eax
    cmpl   %eax, %edx
    jle .L4
```

Where do you see opportunities of compiler optimizations?

```
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// A function to implement bubble sort
void bubbleSort(int *arr, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
        // Last i elements are already
        // in place
        for (j = 0; j < n - 1 - i; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
// Function to print an array
void printArray(int *arr, int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d\t",arr[i]);
}
```

Should become register variables

Common sub-expression elimination (CSE)

```
int main()
{
    unsigned N = 131072;
    int *data;
    if(argc > 1)
        N = atoi(argv[1]);
    data = (int *)malloc(sizeof(int)*N);

    for (unsigned i = 0; i < N; ++i)
        data[i] = rand();
    bubbleSort(data, N);
    printArray(data, N);
    return 0;
}
```

The unoptimized swap function and its caller stub

```
movq    -24(%rbp), %rax
addq    %rcx, %rax
movq    %rdx, %rsi
movq    %rax, %rdi
call    swap
```

put base address of a & b

in `%rdi` and `%rsi`

RSP (stack pointer) register points to top of stack

Instruction	Effective Operations
pushq src	subq $8, %rsp movq src, (%rsp)
popq dest	movq (%rsp), dest addq $8, %rsp

```
swap:
.LFB6:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -24(%rbp)
    movq    %rsi, -32(%rbp)
    movq    -24(%rbp), %rax
    movl    (%rax), %eax
    movl    %eax, -4(%rbp)
    movq    -32(%rbp), %rax
    movl    (%rax), %edx
    movq    -24(%rbp), %rax
    movl    %edx, (%rax)
    movq    -32(%rbp), %rax
    movl    -4(%rbp), %edx
    movl    %edx, (%rax)
    nop
    popq    %rbp
    ret
```

save %rbp in the stack and obtain the current stack pointer

move base addresses into the stack

swap values completely on memory operations...

Where do you see opportunities of compiler optimizations?

function inlining

```
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Should become register variables

```
// A function to implement bubble sort
void bubbleSort(int *arr, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
        // Last i elements are already
        // in place
        for (j = 0; j < n - 1 - i; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
// Function to print an array
void printArray(int *arr, int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d\t",arr[i]);
}
```

Common sub-expression elimination (CSE)

```
int main()
{
    unsigned N = 131072;
    int *data;
    if(argc > 1)
        N = atoi(argv[1]);
    data = (int *)malloc(sizeof(int)*N);

    for (unsigned i = 0; i < N; ++i)
        data[i] = rand();
    bubbleSort(data, N);
    printArray(data, N);
    return 0;
}
```

Optimization with -O1

- Some variables are now in registers
- Leading to fewer memory accesses
- Load Effective Address (lea) — is not a memory instruction, but an arithmetic instruction compute the “address” only
  - leal -1(%rsi), %eax ==
    xorl %eax, %eax
    addl %rsi, %eax
    sub $1, %eax
- Strength Reduction — use a simpler operation

Q6, Q7, Q8: function call overhead

Before

```
movq   %rbx, %rsi
call   swap
jmp .L5
```

put base address of a in `%rsi`

RSP (stack pointer) register points to top of stack

Instruction	Effective Operations
`pushq src`	`subq $8, %rsp` `movq src, (%rsp)`
`popq dest`	`movq (%rsp), dest` `addq $8, %rsp`

```
swap:
    movl    (%rdi), %eax
    movl    (%rsi), %edx
    movl    %edx, (%rdi)
    movl    %eax, (%rsi)
    ret
```

optimized swap operations

2 instructions in caller, 5 instructions in callee = 7

After

```
.L4:
    addq   $4, %rax
    cmpq   %r8, %rax
    je  .L10
.L5:
    movl   (%rax), %edx
    movl   4(%rax), %ecx
    cmpl   %ecx, %edx
    jle .L4
    movl   %ecx, (%rax)
    movl   %edx, 4(%rax)
    jmp .L4
```

Only two lines optimized swap operations

Only 2!

The compiler can leverage the fact that they're a[j] and a[j+1]

-O2

- Loop invariant code motion
- Constant propagation
- Function Inlining
- More you can find when you do your assignment 4!

Q3 and Q4 — similar to this demo!

```
for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)  // Branch X
            sum ++;
    }
}
```

	If data is not sorted?	If data is sorted?
The prediction accuracy of X before threshold	50%	100%
The prediction accuracy of X after threshold	50%	100%

Q5: Loop unrolling — how does that reduce “branch” instructions

```
for (i=0;i<size;i++)
     sum += a[i];
```

```
movslq  (%rdi), %rdx
addq  $4, %rdi
addq  %rdx, %rax
cmpq  %rcx, %rdi
jne   .L21
movslq  (%rdi), %rdx
addq  $4, %rdi
addq  %rdx, %rax
cmpq  %rcx, %rdi
jne   .L21
movslq  (%rdi), %rdx
addq  $4, %rdi
addq  %rdx, %rax
cmpq  %rcx, %rdi
jne   .L21
movslq  (%rdi), %rdx
addq  $4, %rdi
addq  %rdx, %rax
cmpq  %rcx, %rdi
jne   .L21
```

20 instructions and 4 branches, 4 iterations

If we have n iterations?

$5 \times n$ instructions

n branches

```
new_size = size >> 3
for (i=0;i<new_size*8;i++)
     sum += a[i];
```

```
.L26: movslq  (%rdi), %r9
      movslq  8(%rdi), %r10
      addq  $32, %rdi
      movslq  -20(%rdi), %r11
      movslq  -16(%rdi), %rax
      addq  %rcx, %r9
      movslq  -28(%rdi), %rcx
      movslq  -12(%rdi), %rsi
      movslq  -8(%rdi), %r8
      addq  %rcx, %r9
      movslq  -4(%rdi), %rcx
      addq  %r10, %r9
      addq  %r11, %r9
      addq  %rax, %r9
      addq  %rsi, %r9
      addq  %r8, %r9
      addq  %r9, %rcx
      cmpq  %rdx, %rdi
      jne   .L26
.L26: movslq  (%rdi), %r9
      movslq  8(%rdi), %r10
      addq  $32, %rdi
      movslq  -20(%rdi), %r11
      movslq  -16(%rdi), %rax
```

If we have n iterations?

$19 \times \frac{n}{8}$ instructions

$\frac{n}{8}$ branches

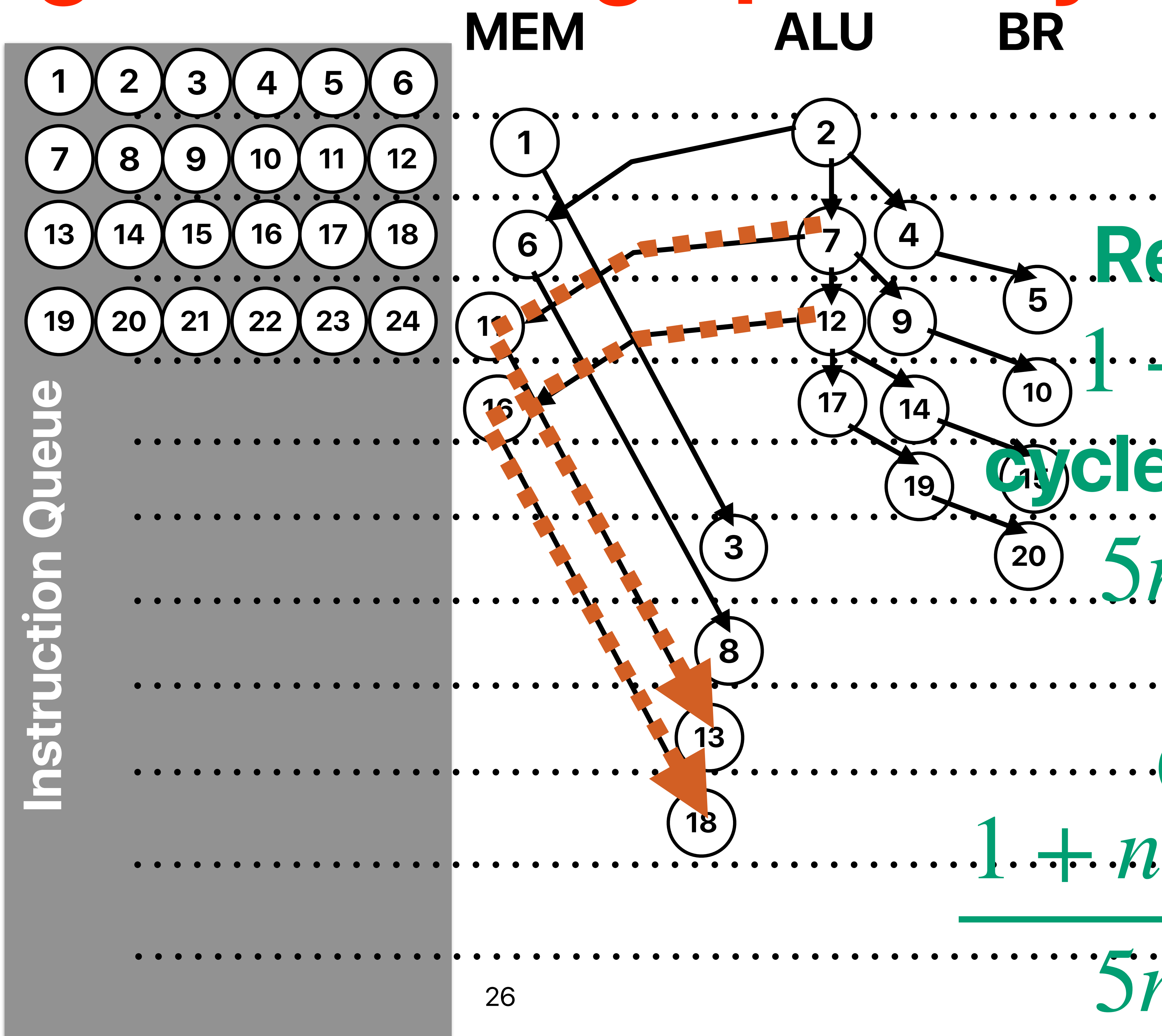
19 instructions and 1 branch for 8 iterations,

Q11—Q15, Q17: critical path of programs

- Critical path: the sequence of instructions determines the execution time
  - If we remove this instruction from the program and the execution time will change, it’s on the critical path
  - Otherwise, it’s not on the critical path
- How to find critical path
  - Draw the dataflow/data dependency graph!
  - The length of each edge should represent the latency of the operation
  - Identify the path that defines the total latency

Through data flow graph analysis

```
for (i=0;i<size;i++)
     sum += a[i];
①  movl (%rdi), %ecx
②  addq $8, %rdi
③  addq %rcx, %rax
④  cmpq %rdx, %rdi
⑤  jne  .L3
⑥  movl (%rdi), %ecx
⑦  addq $8, %rdi
⑧  addq %rcx, %rax
⑨  cmpq %rdx, %rdi
⑩  jne  .L3
⑪  movl (%rdi), %ecx
⑫  addq $8, %rdi
⑬  addq %rcx, %rax
⑭  cmpq %rdx, %rdi
⑮  jne  .L3
⑯  movl (%rdi), %ecx
⑰  addq $8, %rdi
⑱  addq %rcx, %rax
⑲  cmpq %rdx, %rdi
⑳  jne  .L3
```

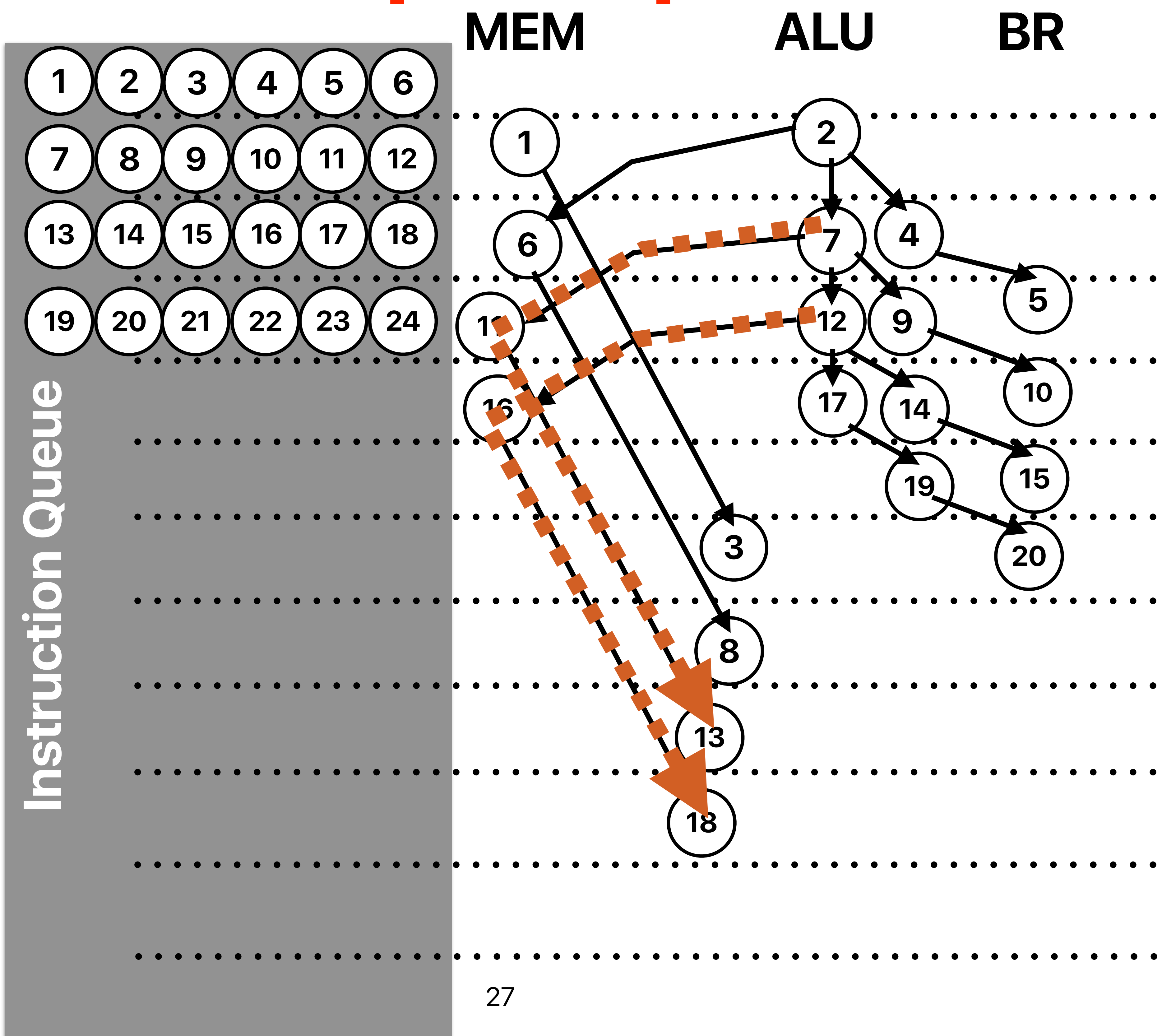


Requires $1+n+5$ cycles to finish $5n$ insts.

CPI = $\frac{1+n+5}{5n} \approx 0.2$

How can we improve performance?

```
for (i=0;i<size;i++)
     sum += a[i];
① movl (%rdi), %ecx
② addq $8, %rdi
③ addq %rcx, %rax
④ cmpq %rdx, %rdi
⑤ jne  .L3
⑥ movl (%rdi), %ecx
⑦ addq $8, %rdi
⑧ addq %rcx, %rax
⑨ cmpq %rdx, %rdi
⑩ jne  .L3
⑪ movl (%rdi), %ecx
⑫ addq $8, %rdi
⑬ addq %rcx, %rax
⑭ cmpq %rdx, %rdi
⑮ jne  .L3
⑯ movl (%rdi), %ecx
⑰ addq $8, %rdi
⑱ addq %rcx, %rax
⑲ cmpq %rdx, %rdi
⑳ jne  .L3
```



Q9: Intel Alder Lake (P-Core)

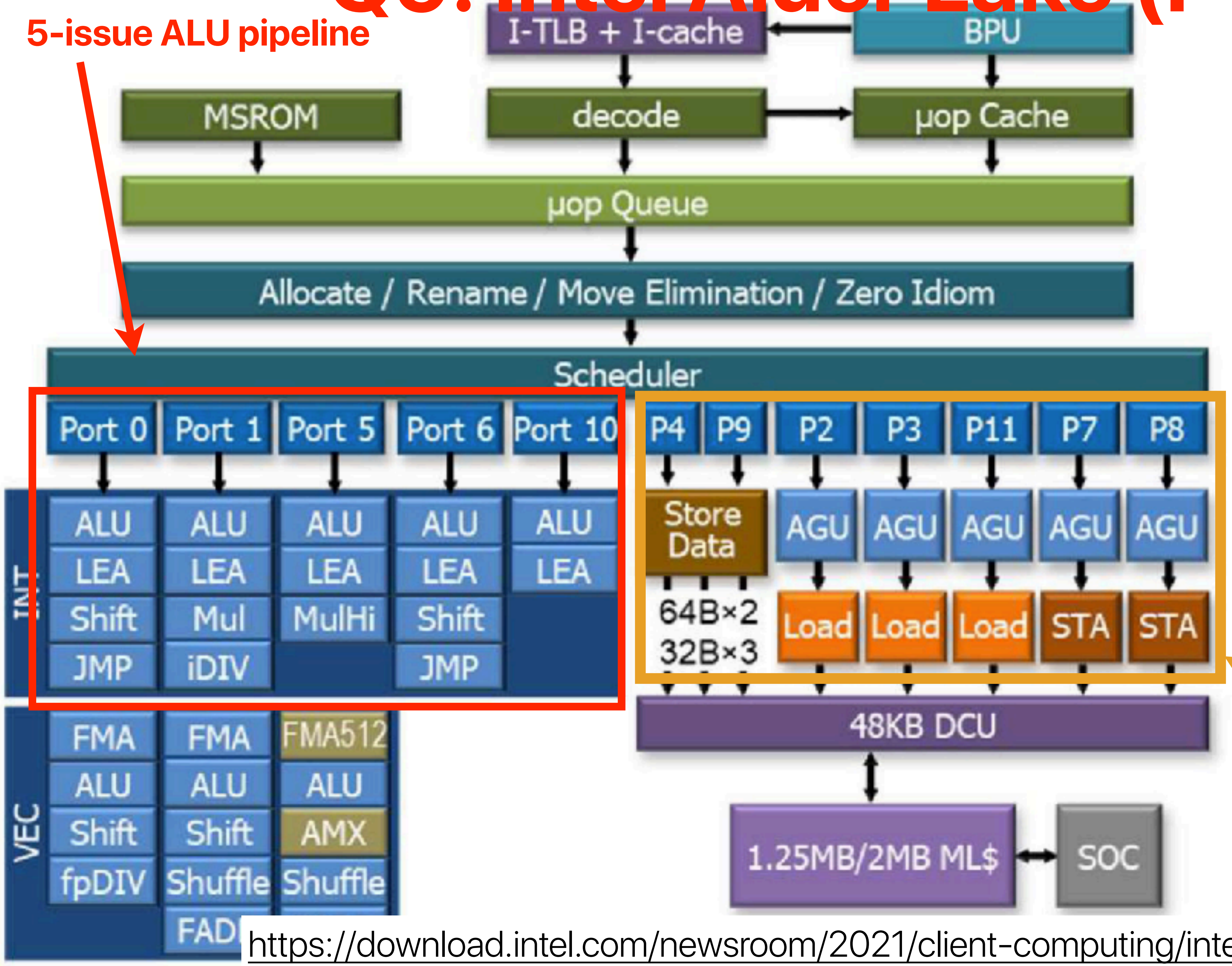
$$MinCPI = \frac{1}{12}$$

$$MinINTInst.CPI = \frac{1}{5}$$

$$MinMEMInst.CPI = \frac{1}{7}$$

$$MinBRInst.CPI = \frac{1}{2}$$

5-issue ALU pipeline



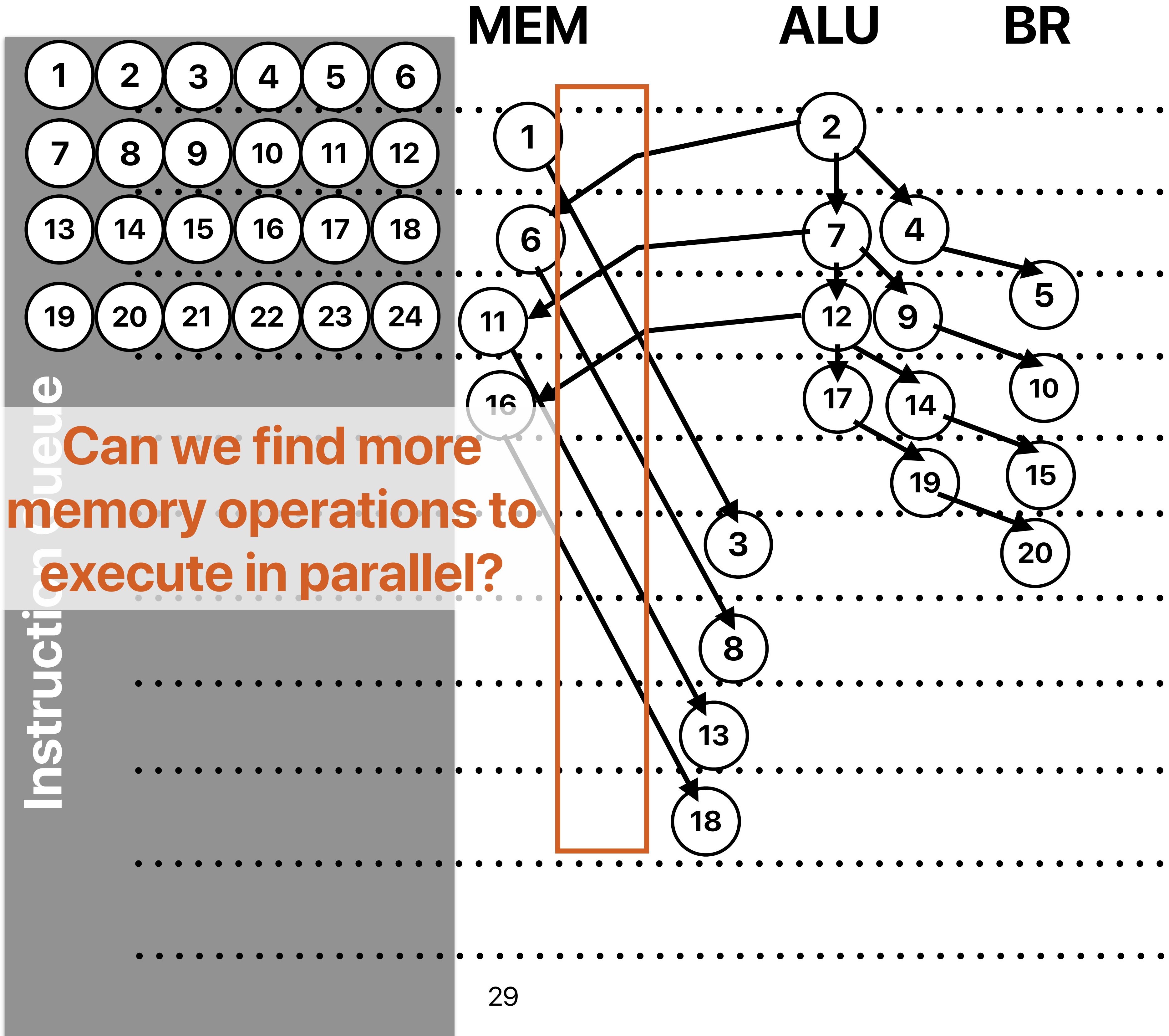
The processor wants to maximize the throughput of memory instructions! (Q10)

7-issue memory pipeline

https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

How can we improve performance?

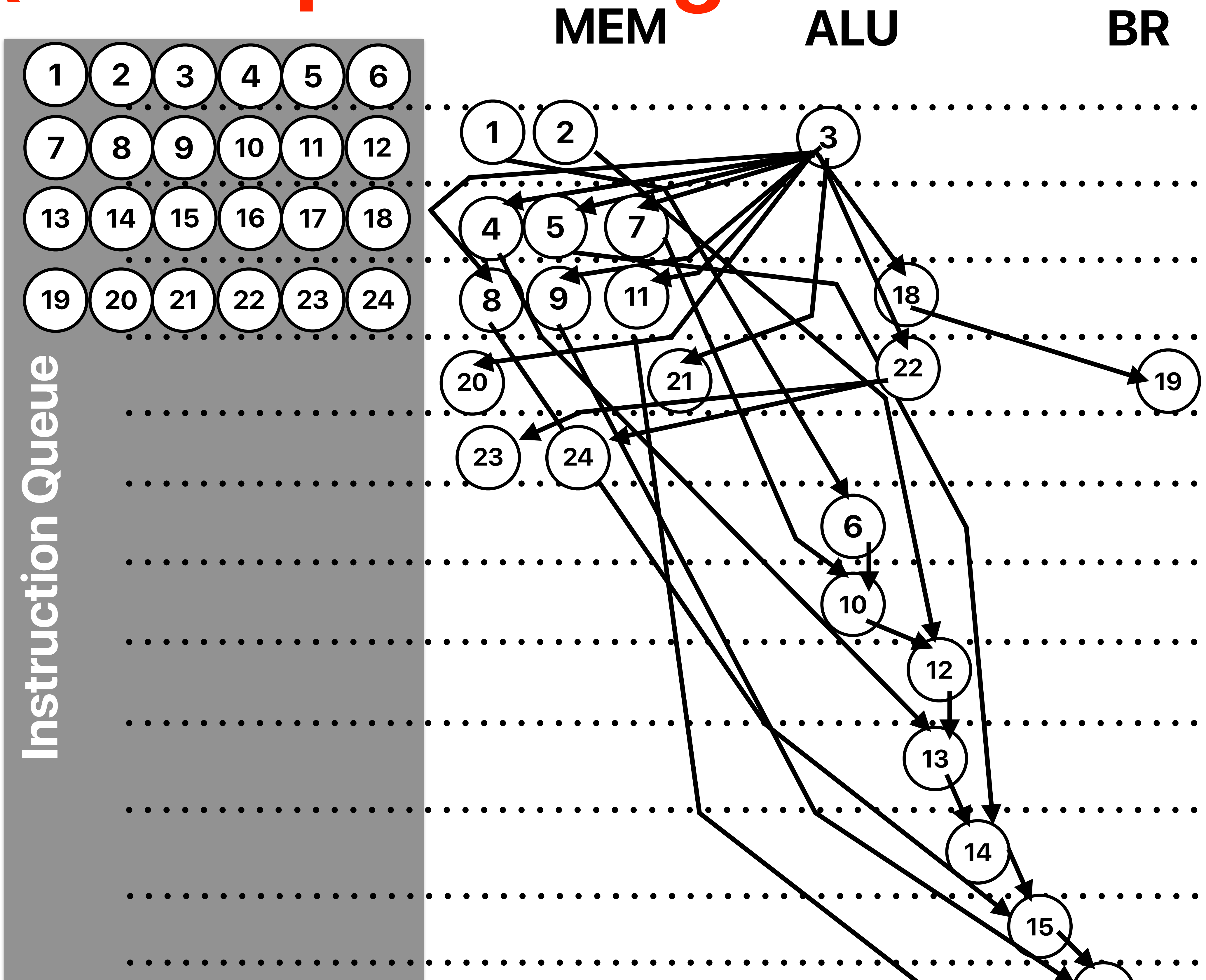
```
for (i=0;i<size;i++)
     sum += a[i];
①  movl (%rdi), %ecx
②  addq $8, %rdi
③  addq %rcx, %rax
④  cmpq %rdx, %rdi
⑤  jne  .L3
⑥  movl (%rdi), %ecx
⑦  addq $8, %rdi
⑧  addq %rcx, %rax
⑨  cmpq %rdx, %rdi
⑩  jne  .L3
⑪  movl (%rdi), %ecx
⑫  addq $8, %rdi
⑬  addq %rcx, %rax
⑭  cmpq %rdx, %rdi
⑮  jne  .L3
⑯  movl (%rdi), %ecx
⑰  addq $8, %rdi
⑱  addq %rcx, %rax
⑲  cmpq %rdx, %rdi
⑳  jne  .L3
```



Q13, Q14: loop unrolling!

```
new_size = size >> 3
for (i=0;i<new_size*8;i++)
    sum += a[i];
```

```
①  .L26: movslq   (%rdi), %r9
②        movslq   8(%rdi), %r10
③        addq  $32, %rdi
④        movslq   -20(%rdi), %r11
⑤        movslq   -16(%rdi), %rax
⑥        addq  %rcx, %r9
⑦        movslq   -28(%rdi), %rcx
⑧        movslq   -12(%rdi), %rsi
⑨        movslq   -8(%rdi), %r8
⑩        addq  %rcx, %r9
⑪        movslq   -4(%rdi), %rcx
⑫        addq  %r10, %r9
⑬        addq  %r11, %r9
⑭        addq  %rax, %r9
⑮        addq  %rsi, %r9
⑯        addq  %r8, %r9
⑰        addq  %r9, %rcx
⑱        cmpq  %rdx, %rdi
⑲        jne   .L26
⑳  .L26: movslq   (%rdi), %r9
㉑        movslq   8(%rdi), %r10
㉒        addq  $32, %rdi
㉓        movslq   -20(%rdi), %r11
㉔        movslq   -16(%rdi), %rax
```



Q16: How do we implement "switch"?

```
void teams_switch(uint64_t *pid, uint64_t *team, uint64_t total_number_of_students)
{
    uint64_t i;
    do {
        switch((pid[i] & 0x7)) {
        // V: taken if false
            case 0:
                team[i]=YELLOW;
                break;
            case 1:
                team[i]=RED;
                break;
            case 2:
                team[i]=BLUE;
                break;
            case 3:
                team[i]=ROCKET;
                break;
            case 4:
                team[i]=YELLOW;
                break;
            case 5:
                team[i]=RED;
                break;
            case 6:
                team[i]=BLUE;
                break;
            default:
                team[i]=ROCKET;
                break;
        }
    // Z: i < total_number_of_students means taken
    } while(++i<total_number_of_students);
    return;
}
```

Store the address of L12 to %r8

Yellow: 0; Red: 1; Blue: 2; Rocket: 3

```
_Z12teams_switchPmS_m:
.LFB3:
    endbr64
    pushq  %rbp
    pushq  %rbx
    movq   %rdi, %r9
    movl   $3, %ebp
    leaq   .L12(%rip), %r8
    movl   $2, %ebx
    movl   $1, %r11d
    movl   $0, %r10d
    jmp .L16
.L14:
    movq   %r10, %rax
.L15:
    movq   %rax, (%rsi,%rcx,8)
    addq   $1, %rcx
    cmpq   %rdx, %rcx
    jnb .L19
.L16:
    movq   (%r9,%rcx,8), %rax
    andl   $7, %eax
    cmpq   $6, %rax
    ja  .L10
    movslq (%r8,%rax,4), %rdi
    addq   %r8, %rdi
    notrack jmp   *%rdi
```

- **cmpq** a, b sets flags based on b−a, but doesn't store

The "jump" table

```
.L12:
    .long  .L14-.L12
    .long  .L15-.L12
    .long  .L15-.L12
    .long  .L15-.L12
    .long  .L14-.L12
    .long  .L13-.L12
    .long  .L11-.L12
    .text
.L13:
    movq   %r11, %rax
    jmp .L15
.L11:
    movq   %rbx, %rax
    jmp .L15
.L10:
    movq   %rbp, %rax
    jmp .L15
.L19:
    popq   %rbx
    popq   %rbp
    ret
```

Retrieving the offset in the jump table

Calculating the address of the instruction

jump to the instruction where %rdi points to

Announcement

- Lab report 4 due this Saturday
- Programming assignment 3 is up and due 9/7/2024
  - 24x is the minimum requirement
- Hall of fame of programming assignment 2
  - Aggressive loop unrolling
  - Matrix transpose
    - Tiled transpose
  - Matrix tiling

Rank	Submission Name	markov_solution_c 8192 128 sp
1	Nicholas Droppa	24.37
2	Andrew	23.13
3	J	21.1
4	steven	20.66
5	Arnav Dandu	20.63

Computer
Science &
Engineering

142L

つづく