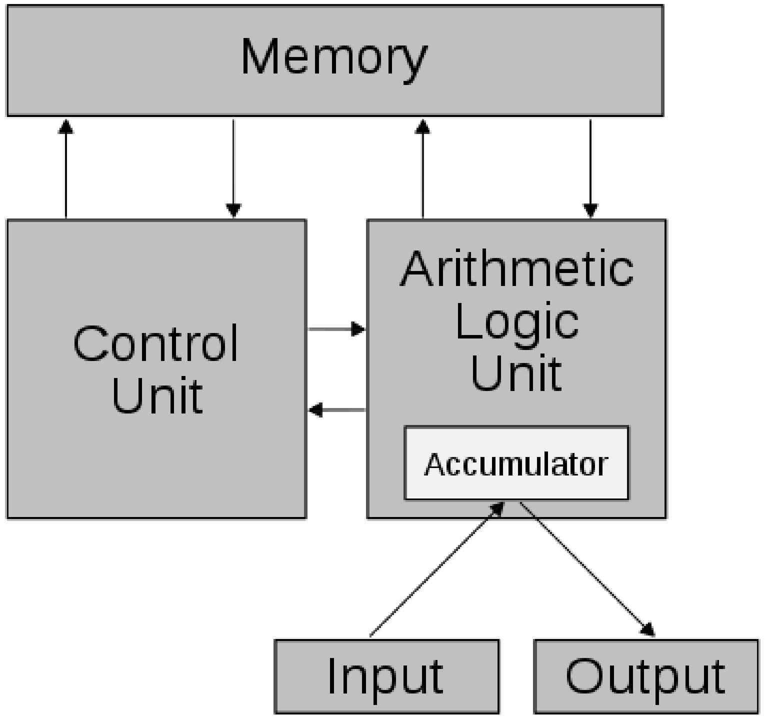
Analysis and modeling of

Computational Performance

Review of computer architectures 1: Instruction processing

von Neumann architecture

- Fundamental instructions:
  - memory accesses
    - reading
    - writing
  - arithmetic operations
  - logical operations
  - jumps
    - unconditional
    - conditional
  - input/output operations
    - often implemented as memory accesses



- Instructions can have from 0 to 3 arguments
  - registers or direct values
  - memory locations with addresses stored in registers

Assembly language

- Short review from Computer Architectures
    - with x86(64) as an example
- Fundamental instructions
    - data transfers
        - mov: memory, registers, no memory-memory
        - push, pop: stack operations x87 FPU
    - arithmetic and logic operations
        - add, sub, mul, imul, div, idiv, (fadd, fsub, fmul, fdiv) : +, -, *, /
        - inc, dec, neg: ++1, --1, *=(-1)
        - not, or, and, xor: ~, ||, &&, (a xor a == 0)
        - cmp: comparison, result in proper bits of the status register
    - control transfer (change of program counter, PC)
        - jmp: unconditional jumps
        - jge, je, jl, etc.: conditional jumps (branches)
            - based on the values in suitable bits of the status register
        - call, ret: used for function calls and returns
    - input/output instructions
        - in, out: read from, write to a port

Assembly language

- Special instructions (related to performance optimization):
  - lea – load effective address (address calculation, no memory transfer)
  - prefetchw – prefetch data into cache from memory
- Standard registers (x86-64 64-bit mode)
  - 64-bit: 16 general purpose registers %rax - %r15
    - frequent meaning: %rax – accumulator, %rbp (frame base pointer), %rsp (stack pointer), etc.
  - instruction pointer (RIP),
  - program status register (RFLAGS)
  - floating point registers: data, instruction pointer, operand pointer, etc.
  - control, debug, I/O, machine specific, segment etc. special registers
    - including: performance monitoring counters
  - compatibility with previous modes:
    - 32-bit: %eax – half of %rax, etc.
    - 16-bit: %ax – half of %eax, etc.
    - 8-bit: %ah, %al – halves of %ax – high and low, etc.

Assembly language

- SIMD extensions (vector processing - MMX, SSE, AVX)
  - vector registers (64-bit mode)
    - 8x 64-bit MMX, 16x 128-bit XMM, 16x 256-bit YMM
  - hundreds of special instructions, including:
    - many types of *mov* instructions
    - many types of arithmetic (*add, sub, mul, div,* etc.) instructions
    - FMA (fused multiply-add) instructions
    - converting, packing and unpacking instructions
- Address calculation:
  - address = base + index*scale + disp(lacement)
  - AT&T notation: disp(base,index,scale), e.g. 32(%rsp, %rax, 8)
  - base and index stored in 32 or 64-bit registers
  - disp and scale are direct numbers (scale=1,2,4,8)
  - scale usually corresponds to the size of data in bytes

Assembly language - example

- Source code

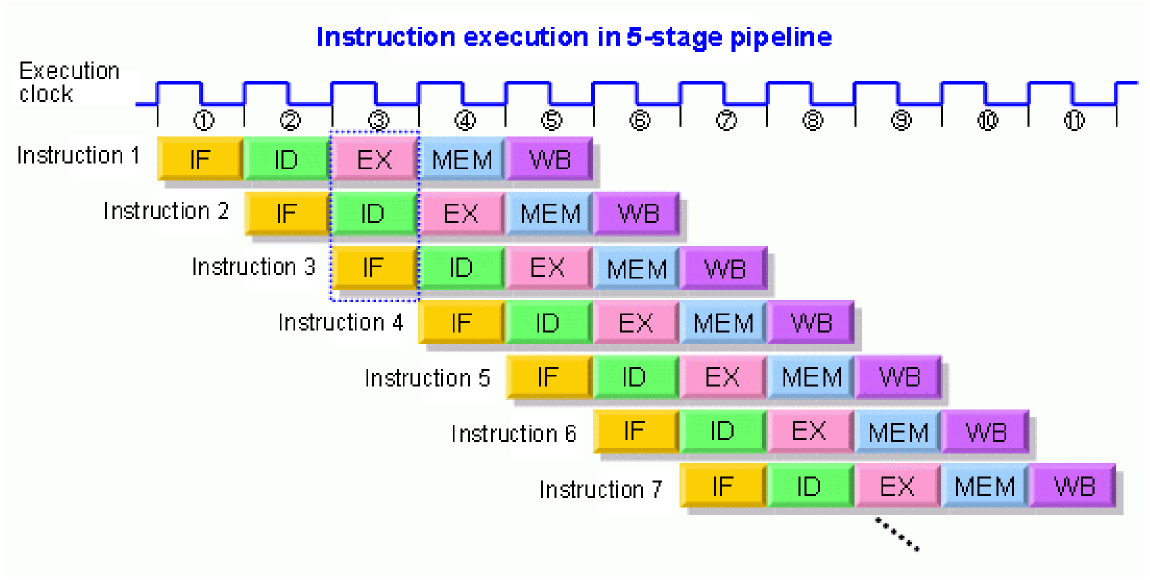
```
  for(i=0;i<num_iter;i++){
    tab_rand[i] = (double)rand()/(RAND_MAX);
  }
```

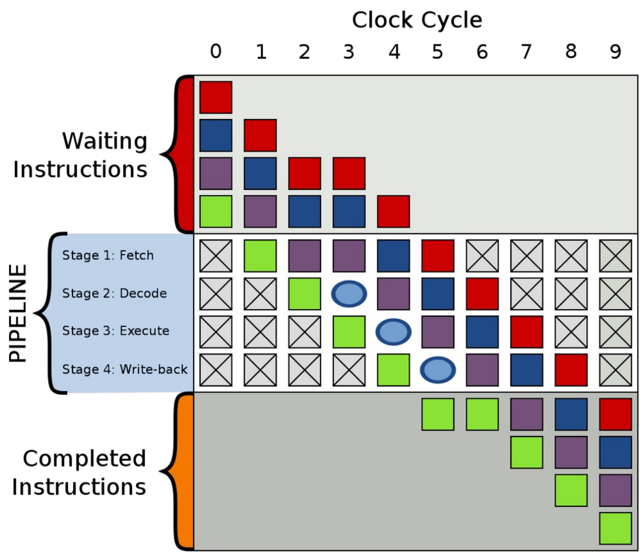
- Assembly code produced by *gcc -O3 -S*

```
.L3:
call rand
cvtsi2sd    %eax, %xmm0
divsd   .LC1(%rip), %xmm0
movsd %xmm0, 0(%rbp,%rbx)
addq    $8, %rbx
cmpq    $80000000, %rbx
jne  .L3
```

Pipelining

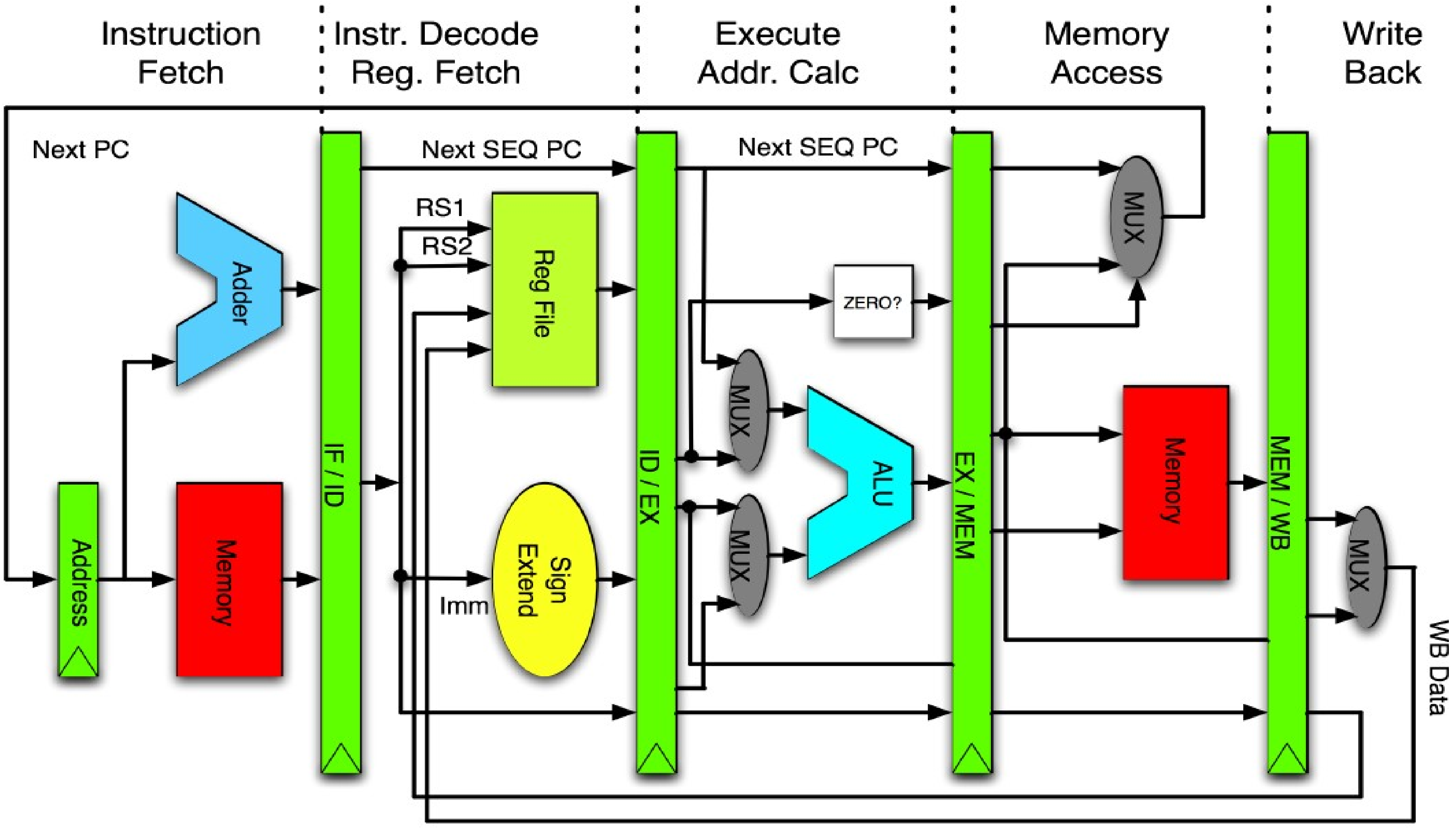
- Instruction processing
  - Fetch
  - Decode
  - Execute
  - (Memory access)
  - Write-back
  - (Interrupts checking)





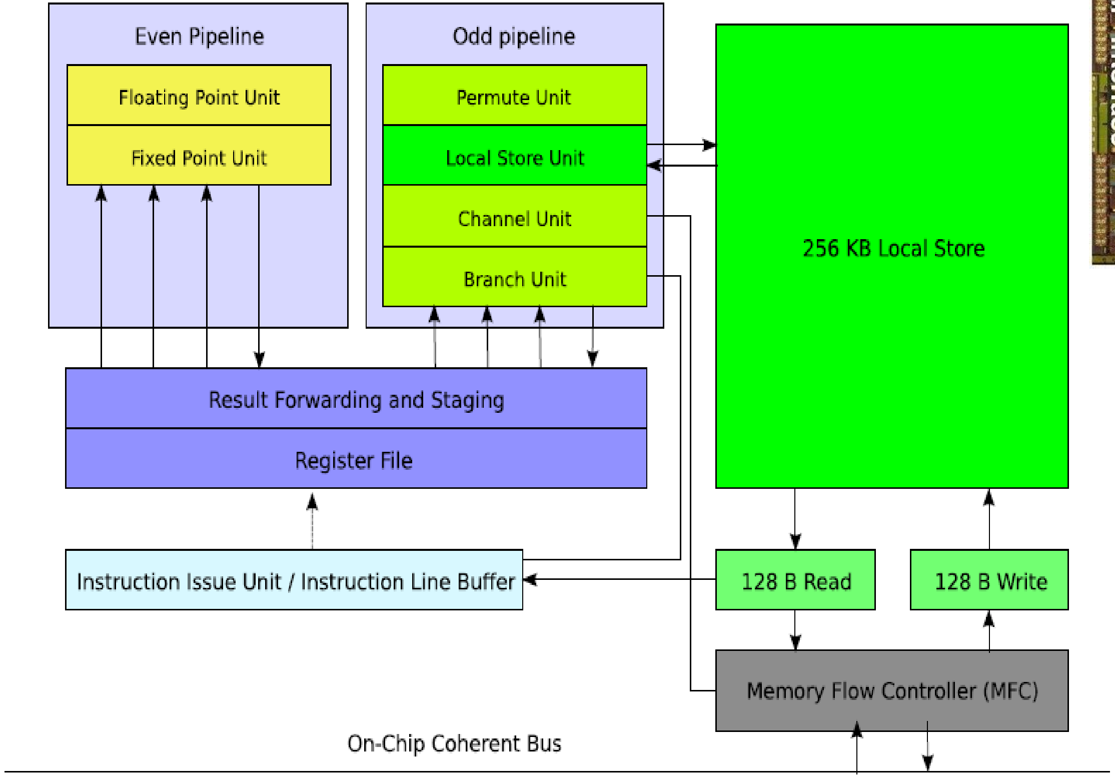
- Pipelining
  - k-stages
  - theoretically k-times faster
  - problems:
    - bubbles, stalls due to hazards
      - from 1 up to hundreds of cycles

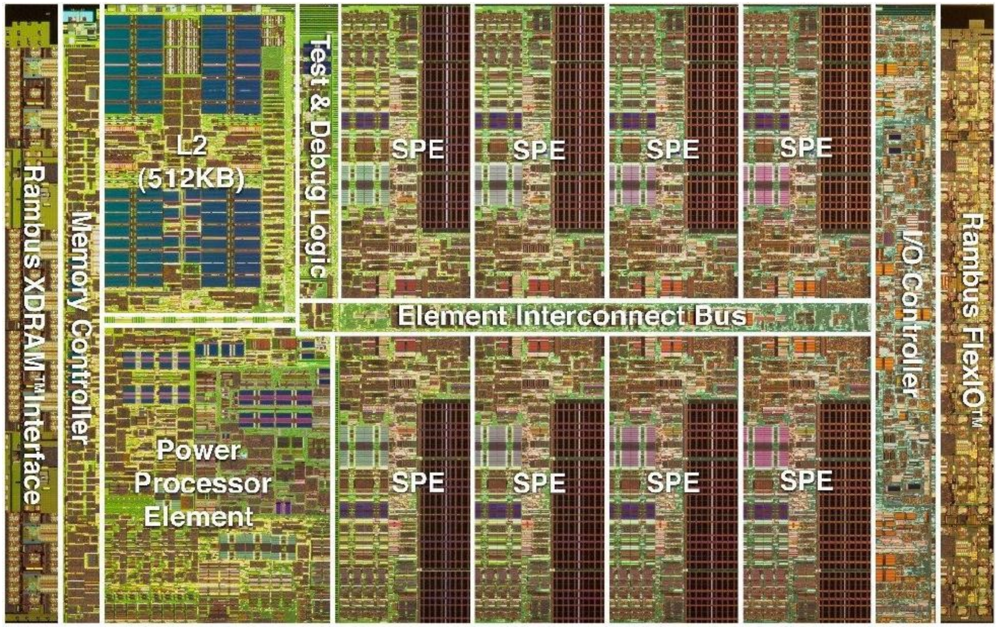
Pipelining – simple RISC design

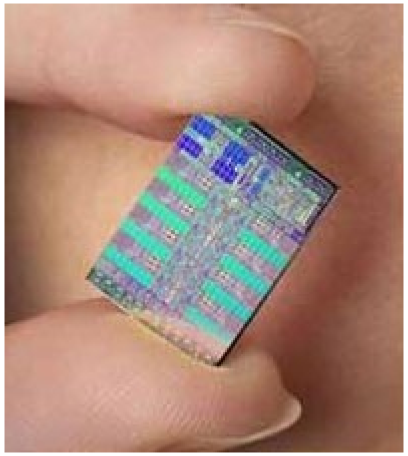


Pipelining – coding example

- PowerXCell processor
  - Cell microarchitecture (PlayStation3)
  - two execution pipelines







Pipelining – coding example

- Efficient use of both (0D and 1D) pipelines
  - no stalls
  - maximal performance – two operations completed at each cycle

```
007681 0D  123456           fma       $118,$124,$67,$118
007681 1D  123456           lqd       $79,160($53)
007682 0D   234567          fma       $120,$124,$65,$120
007682 1D   234567          lqd       $123,176($53)
007683 0D    345678         fma       $67,$124,$67,$68
007683 1D    345678         lqd       $121,192($53)
007684 0D     456789        fma       $68,$124,$69,$70
007684 1D     456789        lqd       $119,208($53)
007685 0D      567890       fma       $65,$124,$65,$66
007685 1D      567890       lqd       $117,224($53)
007686 0D       678901      fma       $69,$124,$69,$72
007686 1D       678901      lqd       $66,240($53)
```

Pipelining – coding example

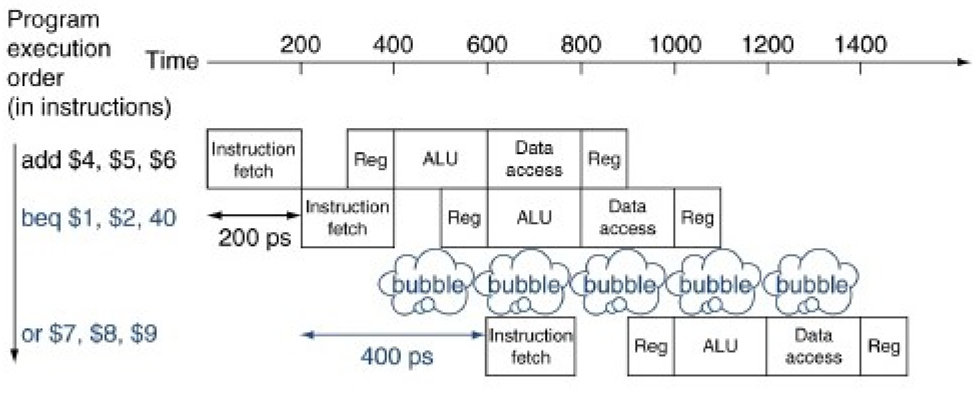
- Inefficient use of pipelines
  - many stalls
  - complex operations

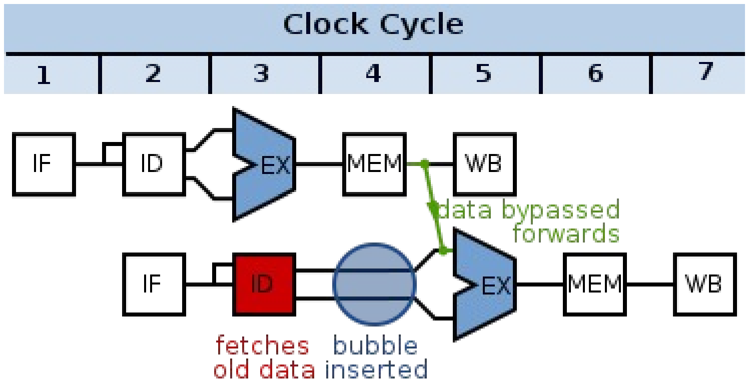
```
005616 1   678901                          lqd          $54,5568($1)
005622 0    -----2345                      rotmi        $42,$54,-31
005626 0          ---67                    a            $42,$54,$42
005628 0              -8901                rotmai       $52,$42,-1
005632 0                ---23              cgti         $42,$52,0
005634 1                    -4567          brz          $42,.LC__88
005635 1                      5678         brz          $27,.LC__88
005636 0D                      67          a            $54,$127,$7

006244 0  ----456789                          dfm     $115,$115,$113
006257 0  -------7890123456789                dfm     $115,$115,$44
006264 0          ------4567890123456         fscrrd  $114
006277 0                 -------------78      selb    $113,$114,$45,$46
```

Pipeline processing

- Pipeline hazards and stalls
  - structural hazards
    - not enough resources
  - control hazards
    - branches, change of PC
      - unconditional jumps
        - prefetching
      - conditional jumps
        - branch prediction
  - data hazards
    - dependencies
      - bypassing data
      - code reorganization

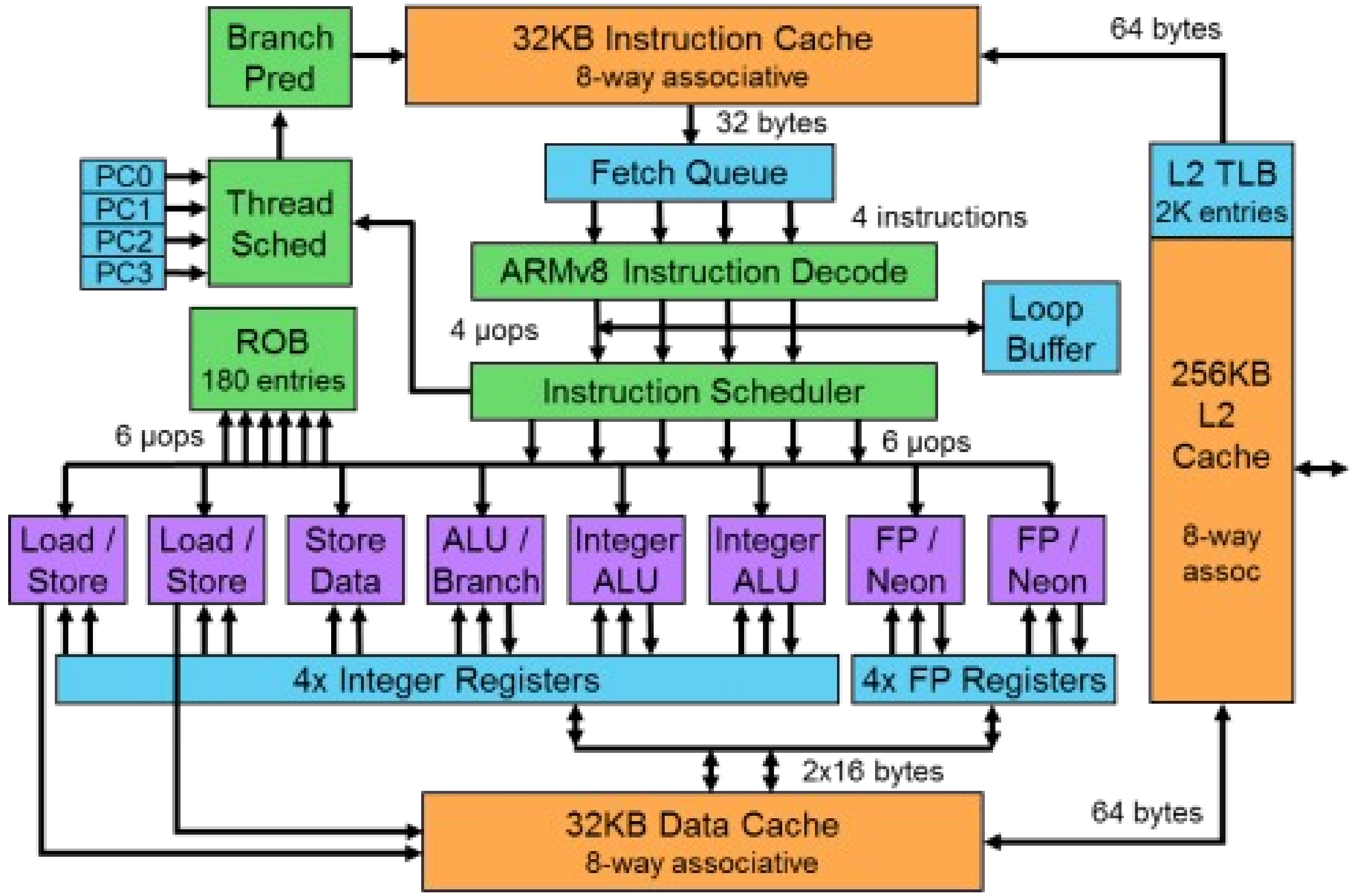




Branches

- Measuring impact of branches on performance
  - Branches can make up to 20% of all executed instructions
    - mainly due to loops
  - The negative effect of unconditional branches can be mitigated by prefetching and out of order execution
  - The negative effect of conditional branches can be mitigated by branch prediction and speculative execution with the example strategies:
    - static prediction (often with branch delay slot)
      - branch always taken
      - branch never taken
      - backward branch taken, forward branch not taken
    - dynamic prediction
      - based on the history of jumps

ARMv8 architecture


Branch Pred
32KB Instruction Cache
8-way associative
64 bytes
32 bytes
PC0
PC1
PC2
PC3
Thread Sched
Fetch Queue
4 instructions
L2 TLB
2K entries
ARMv8 Instruction Decode
Loop Buffer
ROB
180 entries
4 µops
Instruction Scheduler
256KB L2 Cache
8-way assoc
6 µops
6 µops
Load / Store
Load / Store
Store Data
ALU / Branch
Integer ALU
Integer ALU
FP / Neon
FP / Neon
4x Integer Registers
4x FP Registers
2x16 bytes
32KB Data Cache
8-way associative
64 bytes