Processors & Pipelines

These slides adapted from materials provided by the textbook authors.

What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

```
404663:   mov    $0x0,%eax
404668:   cmp    (%rdi),%rsi
40466b:   jge    404685
40466d:   mov    0x8(%rdi),%rax

  . . .

404685:   repz retq
```

Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

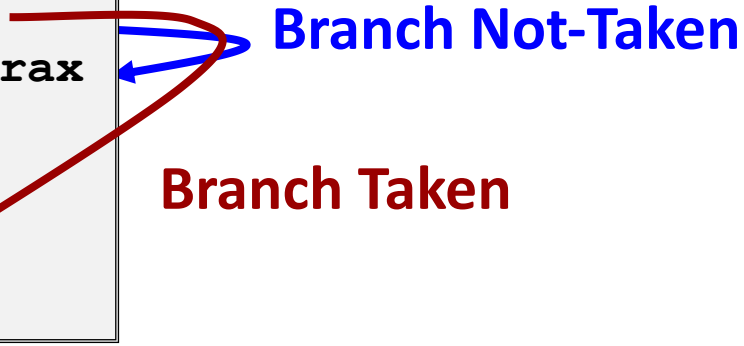
Branch Outcomes

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **Cannot resolve until outcome determined by branch/integer unit**

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```



Branch Prediction

- **Idea**
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don't actually modify register or memory data

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```

Predict Taken

Begin Execution

Branch Prediction Through Loop

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

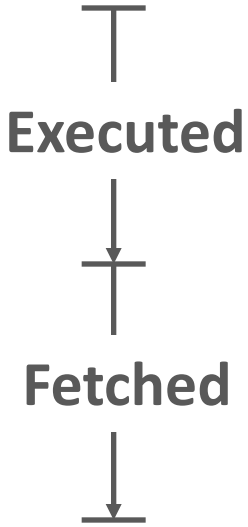
i = 100

Read invalid location

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Assume
vector length = 100



Branch Misprediction Invalidation

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Assume
vector length = 100

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 100

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Invalidate

Branch Misprediction Recovery

```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
401036:   jmp    401040
 . . .
401040:   vmovsd %xmm0,(%r12)
```

i = 99

Definitely not taken

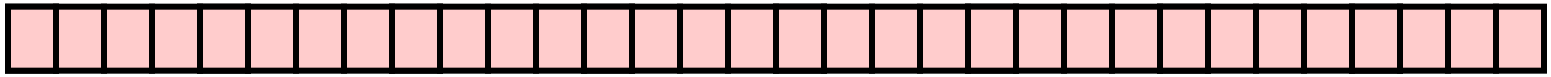
Reload Pipeline

- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

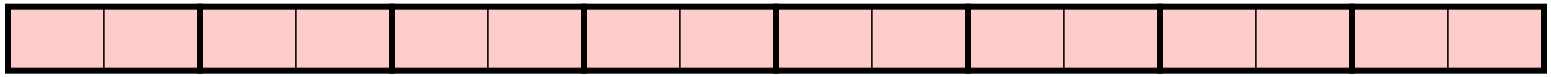
Programming with AVX2

YMM Registers

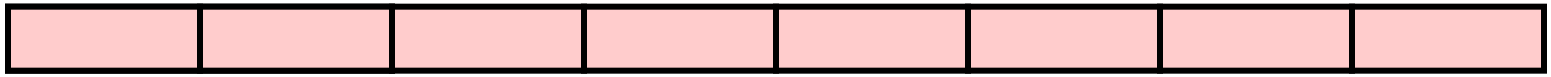
- 16 total, each 32 bytes
- 32 single-byte integers



- 16 16-bit integers

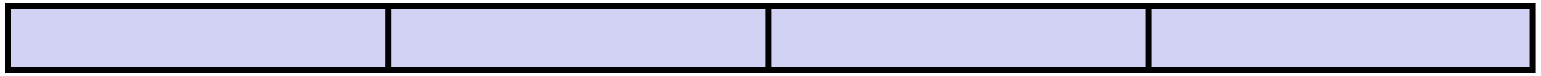


- 8 32-bit integers



- 8 single-precision floats

- 4 double-precision floats

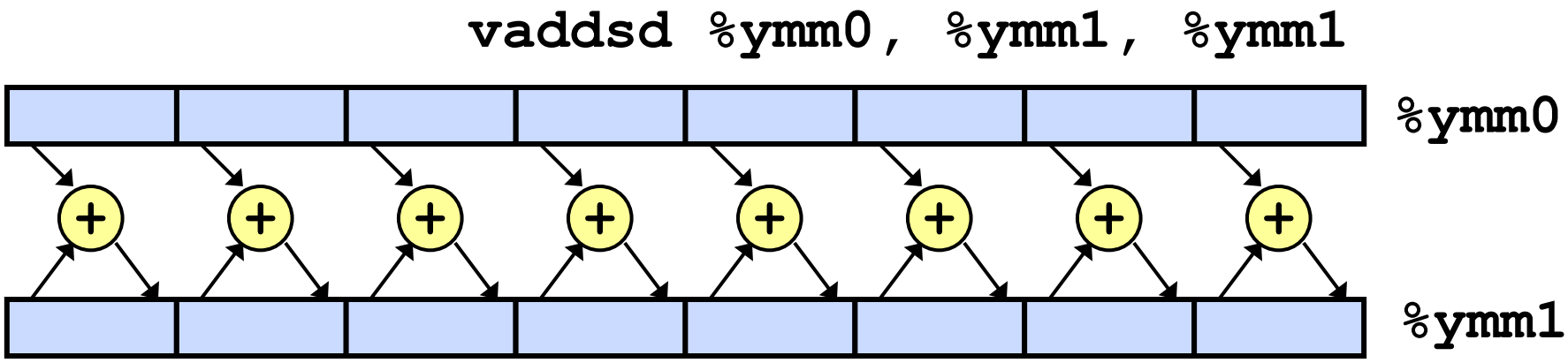


- 1 single-precision float

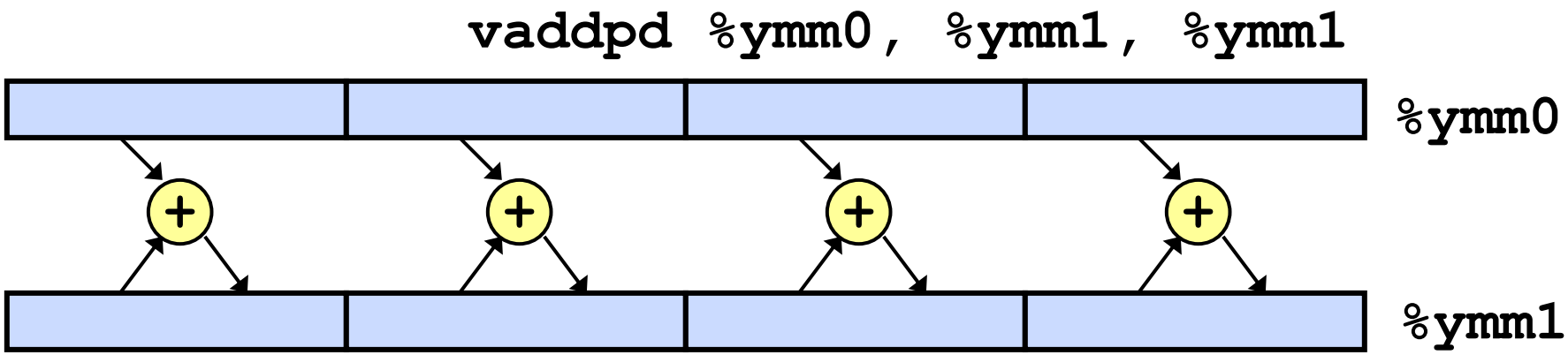
- 1 double-precision float

SIMD Operations

- SIMD Operations: Single Precision



- SIMD Operations: Double Precision



Can you use avx2?

- On linux, file /proc/cpuinfo shows you CPU extensions
- Command “cat /proc/cpuinfo” will output that info

```
jovyan@jupyter-grunwald:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU @ 2.20GHz
stepping        : 0
microcode       : 0x1
cpu MHz         : 2200.000
cache size      : 56320 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxs
all nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid pni pclmulqdq ssse3 fma c
2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single pti f
i1 hle avx2 smep bmi2 erms invpcid rtm rdseed adx smap xsaveopt
bugs            : cpu_meltdown spectre_v1 spectre_v2
bogomips        : 4400.00
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

GCC vectors using “hints” / pragmas

```c
#include <stdio.h>

typedef int data_t;

/* Number of bytes in a vector */
#define VBYTES 32

/* Number of elements in a vector */
#define VSIZE VBYTES/sizeof(data_t)

typedef data_t vec_t __attribute__
      ((vector_size(VBYTES)));

/* Compute inner product of SSE vector */
data_t innerv(vec_t av, vec_t bv) {
    long int i;
    vec_t pv = av * bv;
    data_t result = 0;
    for (i = 0; i < VSIZE; i++)
        result += pv[i];
    return result;
}
```

```
innerv:
.LFB23:
    .cfi_startproc
    leaq    8(%rsp), %r10
    .cfi_def_cfa 10, 0
    andq    $-32, %rsp
    pushq   -8(%r10)
    pushq   %rbp
    .cfi_escape 0x10,0x6,0x2,0x76,0
    movq    %rsp, %rbp
    pushq   %r10
    .cfi_escape 0xf,0x3,0x76,0x78,0x6
    subq    $72, %rsp
    movq    %fs:40, %rax
    movq    %rax, -24(%rbp)
    xorl    %eax, %eax
    vpmulld %ymm1, %ymm0, %ymm1
    vmovdqa %ymm1, -80(%rbp)
    leaq    -80(%rbp), %rdx
    leaq    32(%rdx), %rcx
```

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec Throughput Bound	0.06	0.12	0.25	0.12

- **Make use of AVX Instructions**
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page http://csapp.cs.cmu.edu/3e/waside.html

Getting High Performance

- **Good compiler and flags**
- **Watch out for hidden algorithmic inefficiencies**
- **Write compiler-friendly code**
  - Watch out for optimization blockers:
    procedure calls & memory references
- **Look carefully at innermost loops (where most work is done)**

- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)