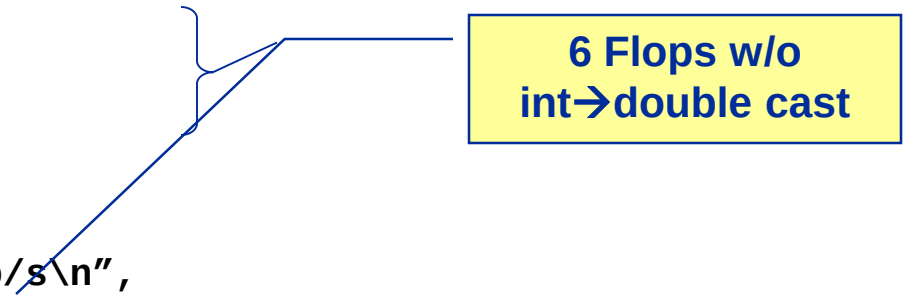
Assignment 0 – Task 1

- **Code**

```
int SLICES = 1000000000; // 1 billion
double delta_x = 1./SLICES,x,sum=0.,ct,wcs,wce,Pi;
timing(&wcs, &ct);
for (int i=0; i < SLICES; i++) {
  x = (i+0.5)*delta_x;
  sum += (4.0 / (1.0 + x * x));
}
timing(&wce, &ct);
Pi = sum * delta_x;
printf("Pi=%.15lf at %.2lf MFlop/s\n",
            Pi,       SLICES*6.0/(wce-wcs)*1e-6);
```



- `module load intel64`
- Compile with `icc -O3 –xHost ....`

Assignment 0 – Task 1

- **Results for Intel compiler (fixed 2.2 GHz clock speed)**
  - Pi version (with DIV)

    T = 3.2s → 1882 MFlop/s

  - Fast version (with MULT – just for fun ☺)

    T = 0.46s → 12940 MFlop/s

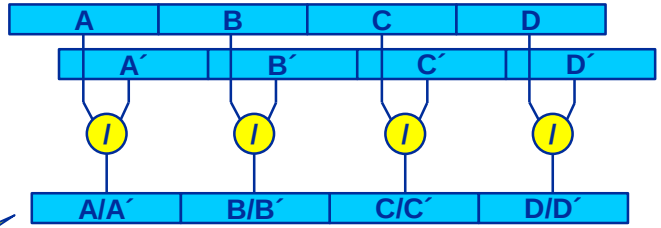
≈ 4 cycles per (AVX) iteration

Theoretical minimum (MULT+ADD throughput): 3 cycles

$2.2 \cdot 10^9$ cy/s / (SLICES/3.2s) = 7 cycles per divide

AVX instructions are used (SIMD vectorization), so one divide instruction performs four Flops

→ DIV instruction throughput ≈ 1/28 cycles^{-1}



SIMD is covered in depth later in the lecture

Assignment 0 – Task 1

- **Single precision divide**

- **Result with `-no-vec` (i.e., without SIMD vectorization):**

π = 0.671089 (obviously wrong), cycles per divide = 7

```
..B1.3
        vxorps    xmm4, xmm4, xmm4    # bitwise XOR (set to 0)
        vcvtsi2ss xmm4, xmm4, eax     # convert int to float
        inc       eax                 # increment loop counter
        vaddss    xmm5, xmm2, xmm4    # float ADD
        vmulss    xmm6, xmm10, xmm5   # float MULT
        vmulss    xmm7, xmm6, xmm6
        vaddss    xmm8, xmm0, xmm7
        vdivss    xmm9, xmm1, xmm8    # float divide, 7cy
        vaddss    xmm3, xmm3, xmm9
        cmp       eax, 1000000000     # compare loop counter
        jl        ..B1.3              # conditional jump
```

Assignment 0 – Task 1

- **Single precision divide**
- **Result without -no-vec (i.e., with SIMD vectorization)**
- **AVX Single Precision → 8-way SIMD**

π = 0.5142 (also wrong, but different ☺)

cycles per divide = 1.1
(i.e., 8.8 cy per SIMD divide!?!?)

- Compiler uses (inexact, 11-bit) reciprocal plus Newton-Raphson step
- Reciprocal instruction is vectorized and pipelined (1-cy throughput)
- No actual divide instruction involved

```
..B1.3:
  add       eax, 16
  vinsertf128 ymm10, ymm0, xmm1, 1
  vpaddd    xmm0, xmm0, xmm6
  vcvtdq2ps ymm11, ymm10
  vpaddd    xmm1, xmm1, xmm6
  vaddps    ymm12, ymm3, ymm11
  vmulps    ymm13, ymm4, ymm12
  vmulps    ymm14, ymm13, ymm13
  vaddps    ymm15, ymm2, ymm14
  vrcpps    ymm10, ymm15
  vmulps    ymm15, ymm10, ymm15
  vaddps    ymm11, ymm10, ymm10
  vmulps    ymm12, ymm15, ymm10
  vsubps    ymm13, ymm11, ymm12
  vmulps    ymm14, ymm7, ymm13
  vaddps    ymm5, ymm5, ymm14
  vinsertf128 ymm10, ymm0, xmm1, 1
  vpaddd    xmm0, xmm0, xmm6
  vcvtdq2ps ymm11, ymm10
  vpaddd    xmm1, xmm1, xmm6
  vaddps    ymm12, ymm3, ymm11
  vmulps    ymm13, ymm4, ymm12
  vmulps    ymm14, ymm13, ymm13
  vaddps    ymm15, ymm2, ymm14
  vrcpps    ymm10, ymm15
  vmulps    ymm15, ymm10, ymm15
  vaddps    ymm11, ymm10, ymm10
  vmulps    ymm12, ymm15, ymm10
  vsubps    ymm13, ymm11, ymm12
  vmulps    ymm14, ymm7, ymm13
  vaddps    ymm8, ymm8, ymm14
  cmp       eax, 1000000000
  jb        ..B1.3
```

RCPPS + Newton-Raphson step

- **Adapted from the Intel64/IA-32 Optimization Manual, Chapter 11:**

  To compute **`Z[i]=A[i]/B[i]`**
  on a large vector of single-precision numbers, **`Z[i]`** can be calculated by a divide operation, or by multiplying **`1/B[i]`** by **`A[i]`**.

  Denoting **`B[i]`** by a, it is possible to estimate $1/a$ using the (V)RCPPS instruction, achieving approximately 11-bit precision.

  For better accuracy you can use one Newton-Raphson iteration:
  We search the zero of $f(x) = a - 1/x$. The initial approximation is $x_0 = rcp(a) \approx 1/a$.

  Then, by the Newton-Raphson scheme, we get

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 2x_0 - ax_0^2$$

  x_1 is an approximation of $1/a$ with approximately 22-bit precision.

- **The compiler applies this trick whenever it is deemed “safe”**
  - heuristics apply