Indian Institute of Technology Delhi

How to improve performance?

- There are **3** factors:
  - IPC, #instructions, and frequency
  - #instructions is dependent on the compiler → not on the architecture
- Let us look at IPC and frequency
- IPC

  1 if there are no stalls, otherwise < 1

  - What is the IPC of an in-order pipeline?

Methods to increase IPC

- Forwarding
- Having more not-taken branches in the code
- Faster instruction and data memories

What about frequency?

- What is frequency dependent on …
- Frequency = 1 / clock period
- Clock Period:
  - 1 pipeline stage is expected to take 1 clock cycle
  - Clock period = maximum latency of the pipeline stages
- How to reduce the clock period?
  - Make each stage of the pipeline smaller by increasing the number of pipeline stages
  - Use faster transistors

Limits to Increasing Frequency

- Assume that we have the fastest possible transistors
- Can we increase the frequency to 100 GHz?

Reasons

Limits to increasing frequency - II

- What does it mean to have a very high frequency?
- Before answering, keep these facts in mind:

1	Thumb Rule	$P \propto f^3$
2	Thermo-dynamics	$\Delta T \propto P$

P → power
f → frequency
T → Temperature

3 We need to increase the number of pipeline stages → more hazards, more forwarding paths

Pipeline Stages vs IPC

- $CPI = \frac{1}{IPC}$

$$CPI = CPI_{ideal} + stall\_rate * stall\_penalty$$

- The stall rate will remain more or less constant per instruction with the number of pipeline stages
- The **stall penalty** (in terms of cycles) will however increase
- This will lead to a net increase in CPI and **loss** in IPC

As we increase the number of stages, the IPC goes down.

What is ILP = Instruction level parallelism

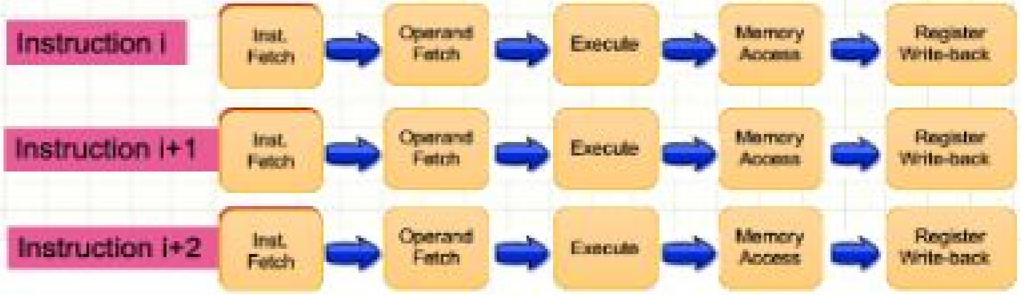
- *multiple operations (or instructions) can be executed in parallel, from a single instruction stream*
  - *so we are **not** yet talking about MIMD, multiple instruction streams*

Needed:

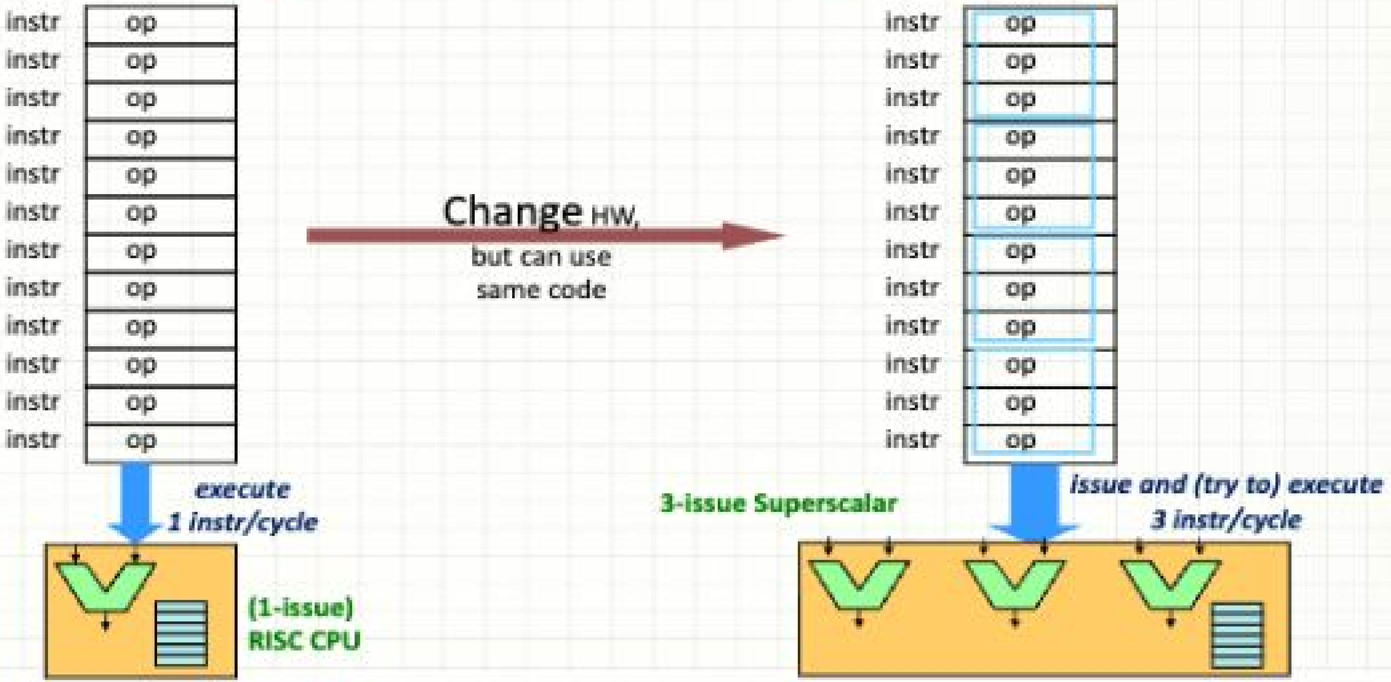
- Sufficient (HW) resources
- Parallel scheduling
  - Hardware solution
  - Software solution
- Application should contain sufficient ILP

Since we cannot increase frequency …

- Increase IPC
  - Issue more instructions per cycle
    - 2, 4, or 8 instructions
- Make it a superscalar processor → A processor that can execute multiple instructions per cycle
  - Have multiple in-order pipelines

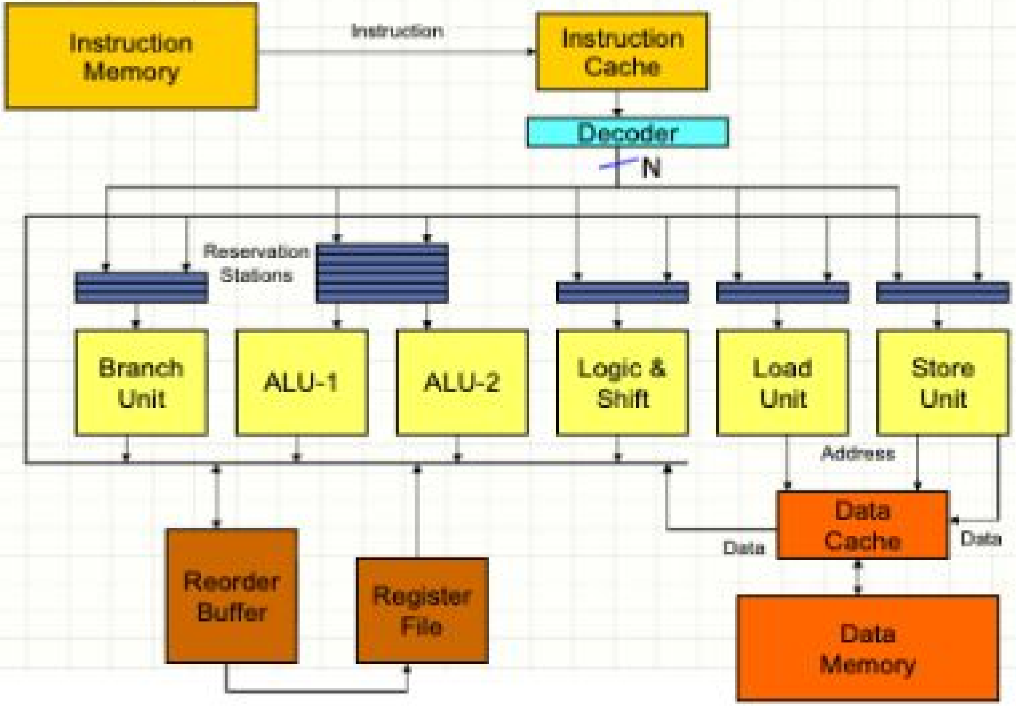


Single Issue RISC vs Superscalar



Superscalar: General Architecture Concept

N = issue rate



Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX				
SUB.D	F8,F2,F6		IF	EX				
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF				
MUL.D	F12,F2,F4							

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX			
SUB.D	F8,F2,F6		IF	EX	EX			
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF				
MUL.D	F12,F2,F4							

stall because of data dep.

cannot be fetched because window full

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX		
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF		EX		
MUL.D	F12,F2,F4					IF		

Example of Superscalar Processor Execution

- **Superscalar processor organization:**
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX	EX	
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF		EX	EX	
MUL.D	F12,F2,F4					IF		

cannot execute
structural hazard

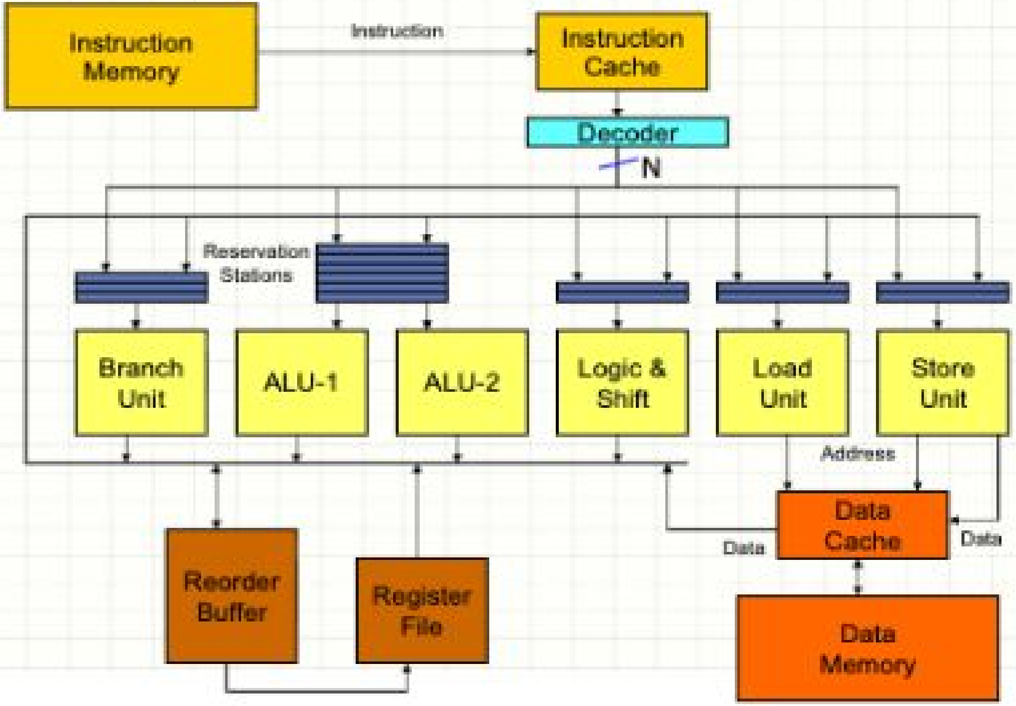
Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX	EX	WB
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				EX
ADD.D	F6,F8,F2			IF		EX	EX	WB
MUL.D	F12,F2,F4					IF		?

Superscalar: General Architecture Concept

N = issue rate



Hazards

- Three types of hazards
  - Structural
    - Multiple instructions need access to the same hardware at the same time
  - Data dependence
    - There is a dependence between operands (in register or memory) of successive instructions
  - Control dependence
    - Determines the order of the execution of basic blocks
    - When jumping/branching to another address the pipeline has to be (partly) squashed and refilled
- Hazards cause scheduling problems and delay the pipeline

Data dependences

- **RaW** read after write
  - real or flow dependence
  - can only be avoided by value prediction (i.e. speculating on the outcome of a previous operation)
- **WaR** write after read
- **WaW** write after write
  - WaR and WaW are false or name dependencies
  - Could be avoided by renaming (if sufficient registers are available); see later slide

Notes:

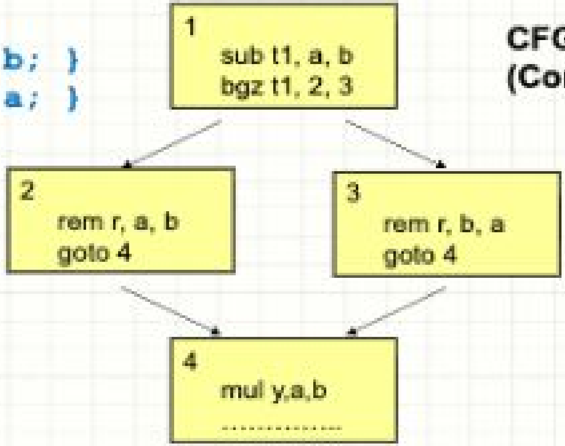
1. data dependences can be **both** between **registers** and **memory** data operations
2. data dependencies are shown in de DDG: Data Dependence Graph

Control Dependences: CFG

C input code:

```
if (a > b)  { r = a % b; }
    else    { r = b % a; }
y = a*b;
```

CFG (Control Flow Graph):



Questions:

- *How real are control dependences?*
- *Can '`mul y,a,b`' be moved to block 2, 3 or even block 1?*
- *Can 'rem r, a, b' be moved to block 1 and executed speculatively?*

Avoiding pipeline stalls due to Hazards

- **Structural**
  - Buy more hardware
    - Extra units, pipelined units, more ports on RF and data memory (or banked memories), etc.
  - Note: more HW means bigger chip => could increase cycle time t_{cycle}
- **Data dependence**
  - Real (RaW) dependences: add Forwarding (aka Bypassing) logic
    - Compiler optimizations
  - False (WaR & WaW) dependences: use renaming (either in HW or in SW)
- **Control dependence**
  - Adding extra pipeline HW to reduce the number of Branch delay slots
  - Branch prediction
  - Avoiding Branches

FP Loop: Where are the Hazards?

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in cycles	stalls between in cycles
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

```
Loop: L.D    F0,0(R1) ;F0=vector element
      ADD.D  F4,F0,F2 ;add scalar from F2
      S.D    0(R1),F4 ;store result
      DADDUI R1,R1,-8 ;decrement pointer 8B
      BNEZ   R1,Loop  ;branch R1!=zero
```

```
1 Loop: L.D    F0,0(R1)   ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2   ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4   ;store result
7       DADDUI R1,R1,-8   ;decrement pointer 8B (DW)
8       stall             ;assumes can't forward to branch
9       BNEZ   R1,Loop    ;branch R1!=zero
```

```
1 Loop: L.D    F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4
7       BNEZ   R1,Loop
```

```
1  Loop: L.D    F0,0(R1)
2        L.D    F6,-8(R1)
3        L.D    F10,-16(R1)
4        L.D    F14,-24(R1)
5        ADD.D  F4,F0,F2
6        ADD.D  F8,F6,F2
7        ADD.D  F12,F10,F2
8        ADD.D  F16,F14,F2
9        S.D    0(R1),F4
10       S.D    -8(R1),F8
11       S.D    -16(R1),F12
12       DSUBUI R1,R1,#32
13       S.D    8(R1),F16 ; 8-32 = -24
14       BNEZ   R1,LOOP
```

14 clock cycles, or 3.5 per iteration

```
1  Loop: L.D    F0,0(R1)
3        ADD.D  F4,F0,F2
6        S.D    0(R1),F4        ;drop DSUBUI & BNEZ
7        L.D    F6,-8(R1)
9        ADD.D  F8,F6,F2
12       S.D    -8(R1),F8       ;drop DSUBUI & BNEZ
13       L.D    F10,-16(R1)
15       ADD.D  F12,F10,F2
18       S.D    -16(R1),F12     ;drop DSUBUI & BNEZ
19       L.D    F14,-24(R1)
21       ADD.D  F16,F14,F2
24       S.D    -24(R1),F16
25       DADDUI R1,R1,#-32      ;alter to 4*8
26       BNEZ   R1,LOOP
```

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)

Dynamic Scheduling Principle

- What we examined so far is *static scheduling*
  - Compiler reorders instructions so as to avoid hazards and reduce stalls
- *Dynamic scheduling*:
  hardware rearranges instruction execution to reduce stalls
- Example:

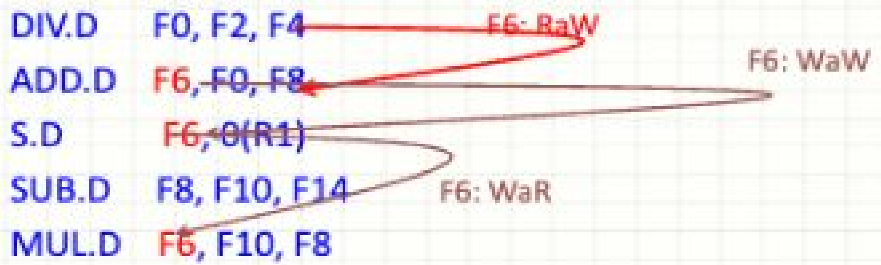
```
    DIV.D  F0,F2,F4       ; takes 24 cycles and
RaW: real dependence      ; is not pipelined
    ADD.D  F10,F0,F8

    SUB.D  F12,F8,F14
```

This instruction cannot continue even though it does not depend on previous Div and Add

- **Key idea:** Allow instructions behind stall to proceed

Register Renaming: General Idea

- Example, look at F6:



```
DIV.D    F0, F2, F4
ADD.D    F6, F0, F8
S.D      F6, 0(R1)
SUB.D    F8, F10, F14
MUL.D    F6, F10, F8
```

- False (aka "name") dependences with F6
  - anti: WaR and
  - output: WaW in this example

Question: how can this code (optimally) be executed?

Register Renaming Technique

- Eliminate name/false dependencies:
  - anti- (WaR) and output (WaW)) dependencies
- Can be implemented
  - by the **compiler**
    - advantage: low cost
    - disadvantage: "old" codes perform poorly
  - by **hardware**
    - advantage: binary compatibility
    - disadvantage: extra hardware needed

Register Renaming by HW

- Same example after renaming:

```
DIV.D    R, F2, F4
ADD.D    S, R, F8
S.D      S, 0(R1)
SUB.D    T, F10, F14
MUL.D    U, F10, T
```

Original code:

```
DIV.D  F0, F2, F4
ADD.D  F6, F0, F8
S.D    F6, 0(R1)
SUB.D  F8, F10, F14
MUL.D  F6, F10, F8
```

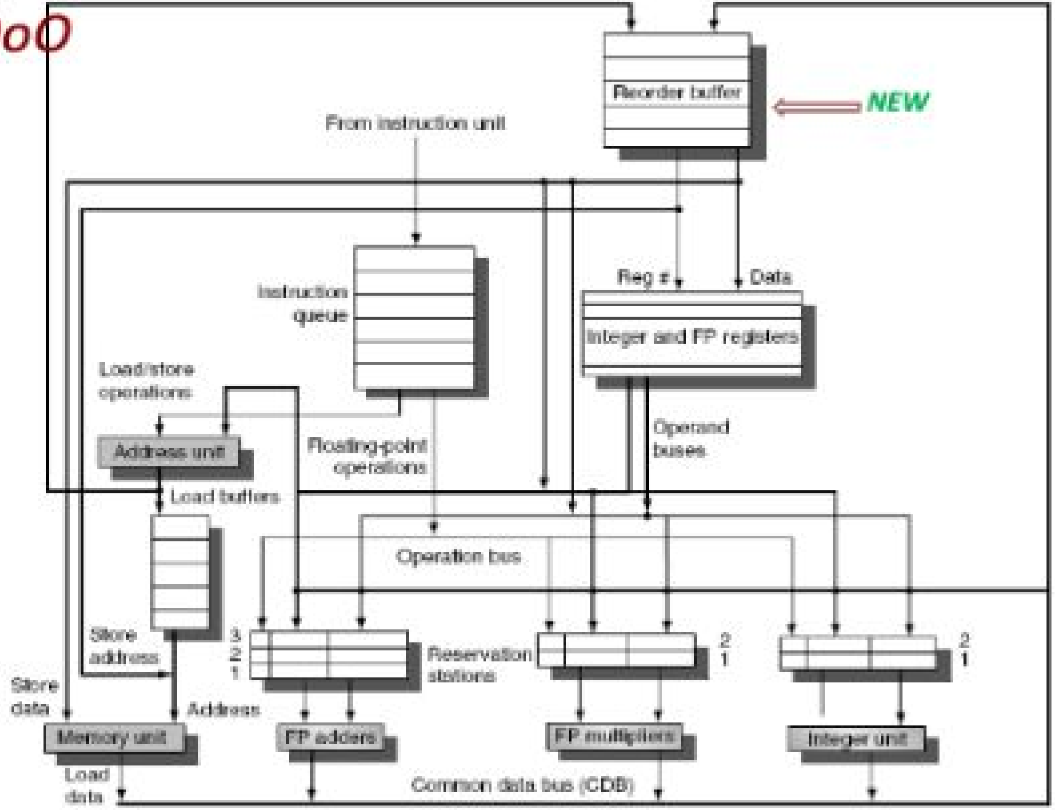
- Each destination gets a new (physical) register assigned
- Now only RaW hazards remain, which can be strictly ordered
- We will see several HW implementations of Register Renaming
  1. use ReOrder Buffer (ROB) & Reservation Stations, or
  2. use large register file with mapping table

Speculation (Hardware based)

- Execute instructions along predicted execution paths but ***only commit the results if the prediction was correct***
- Instruction commit: *allowing an instruction to only **update** the register file when instruction is no longer speculative*
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits:
  - **Reorder buffer, or Large renaming register file**
  - why? think about it?

Speculative OoO

execution with speculation using RoB



Indian Institute of Technology Delhi

Reorder Buffer (RoB)

- Register values and memory values are not written until an instruction commits

- RoB effectively renames the destination registers
  - every destination gets a new entry in the RoB
- On misprediction:
  - Speculated entries in RoB are cleared

- Exceptions:
  - Not recognized/taken until it is ready to commit
  - Precise exceptions require that 'later' entries in RoB are cleared

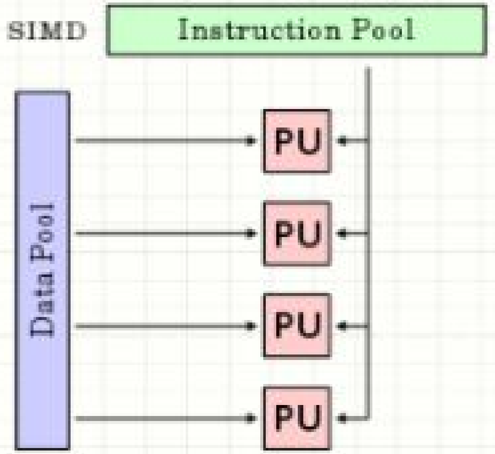
Flynn* Taxonomy, 1966

- In 2013, SIMD and MIMD most common parallelism in architectures – usually both in same system!
- Most common parallel processing programming style: Single Program Multiple Data ("SPMD")
  - Single program that runs on all processors of a MIMD
  - Cross-processor execution coordination using synchronization primitives
- SIMD (aka hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

*Prof. Michael Flyn
Stanford

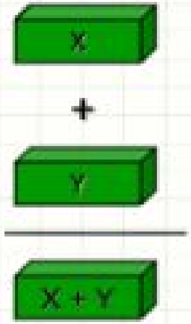
Single-Instruction/Multiple-Data Stream (SIMD or "sim-dee")

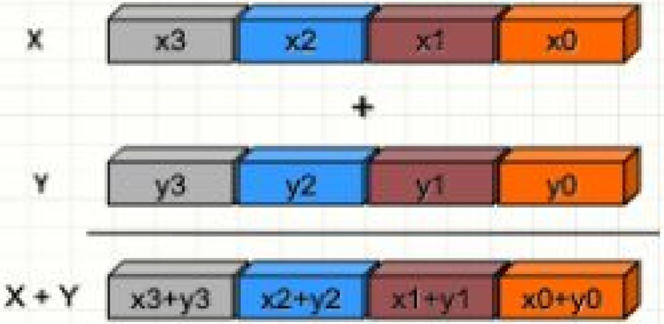


- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

SIMD: Single Instruction, Multiple Data

- Scalar processing
  - traditional mode
  - one operation **produces** one result
- SIMD processing
  - With Intel SSE / SSE2
  - SSE = streaming SIMD extensions
  - one operation **produces** multiple results





Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

What does this mean to you?

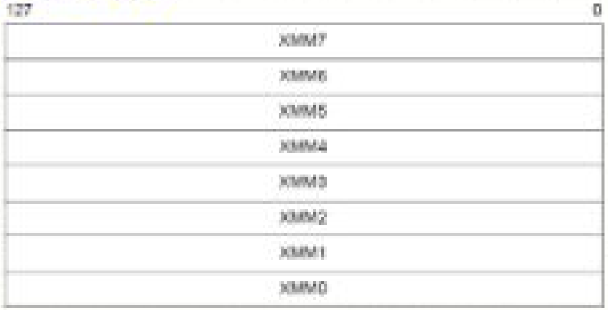
- In addition to SIMD extensions, the processor may have other special instructions
  - Fused Multiply-Add (FMA) instructions:

    x = y + c * z

    is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or * alone
- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Using special functions ("intrinsics") or write in assembly ☹

Intel SIMD Extensions

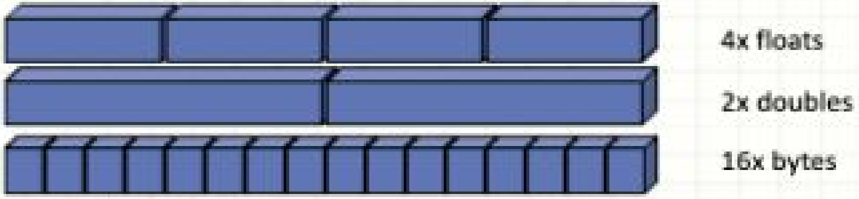
- MMX 64-bit registers, reusing floating-point registers [1992]
- SSE2/3/4, new 8 128-bit registers [1999]



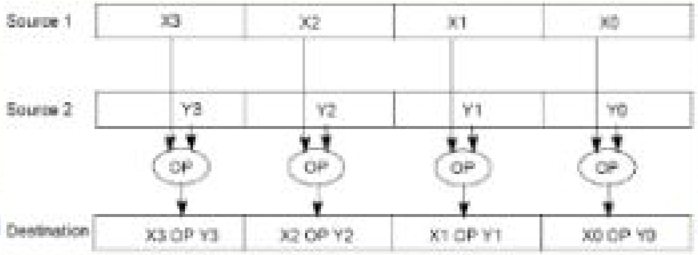
- AVX, new 256-bit registers [2011]
  – Space for expansion to 1024-bit registers

SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



- Instructions perform add, multiply etc. on all the data in parallel

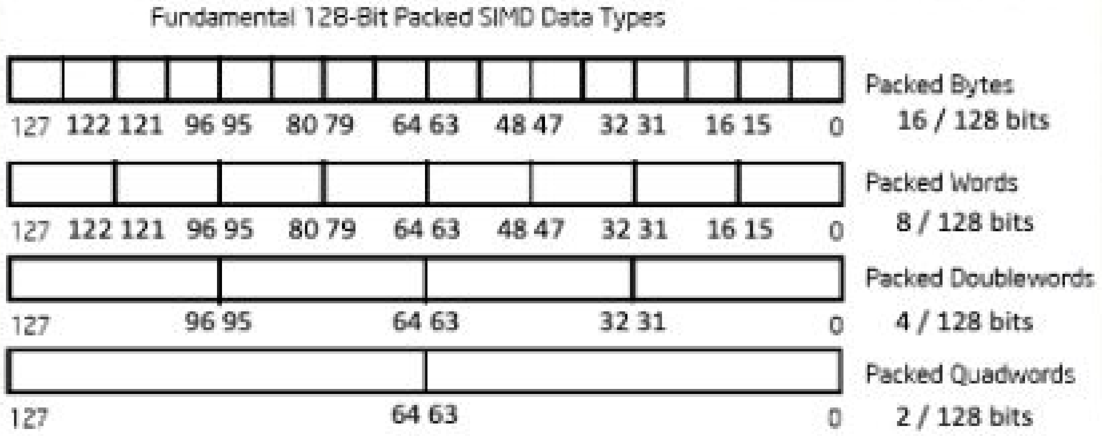


- Similar on GPUs, vector p erations)

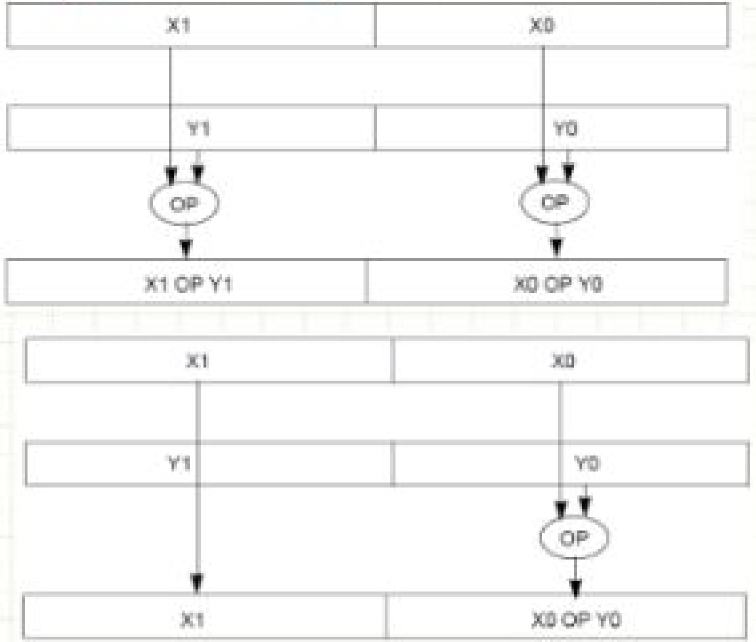
Indian Institute of Technology Delhi

Intel Architecture SSE2+ 128-Bit SIMD Data Types

- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
  - Single-precision FP: Double word (32 bits)
  - Double-precision FP: Quad word (64 bits)



Packed and Scalar Double-Precision Floating-Point Operations



Packed

Scalar

Example: SIMD Array Processing

```
for each f in array
    f = sqrt(f)
```

```
for each f in array
{
    load f to floating-point register
    calculate the square root
    write the result from the
register to memory
}
```

```
for each 4 members in array
{
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    store the 4 results from the register to memory
}
```

SIMD style

Data-Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

```
    for(i=1000; i>0; i=i-1)
        x[i] = x[i] + s;
```

- How can reveal more data-level parallelism than available in a single iteration of a loop?
- *Unroll loop* and adjust iteration rate

Loop Unrolling in C

- Instead of compiler doing loop unrolling, could do it yourself in C

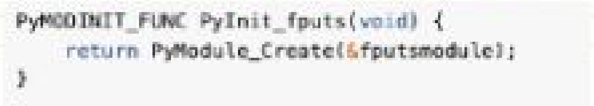
```
for(i=1000; i>0; i=i-1)
     x[i] = x[i] + s;
```

- Could be rewritten

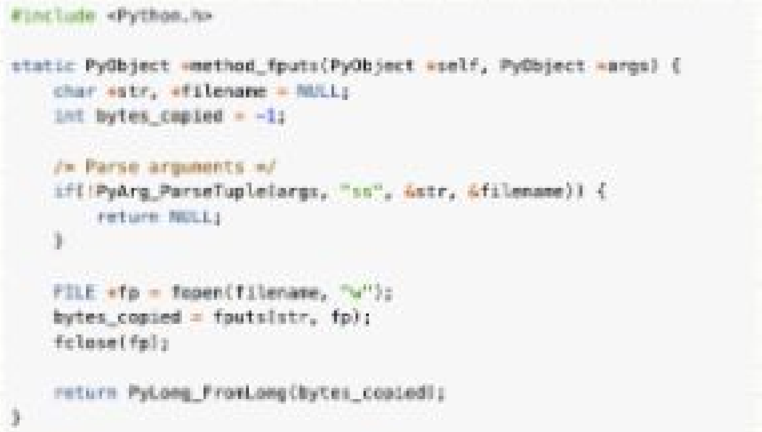
```
for(i=1000; i>0; i=i-4) {
        x[i]   = x[i] + s;
        x[i-1] = x[i-1] + s;
        x[i-2] = x[i-2] + s;
        x[i-3] = x[i-3] + s;
   }
```

Building a Python-C extension

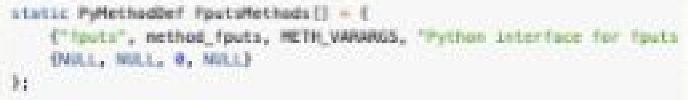
- Write the C function
  - PyObject
  - PyArg_ParseTuple()
  - PyLong_FromLong()
- Write the init function
  - PyMethodDef
  - PyModuleDef
- PyMODINIT_FUNC
  - First import calls this



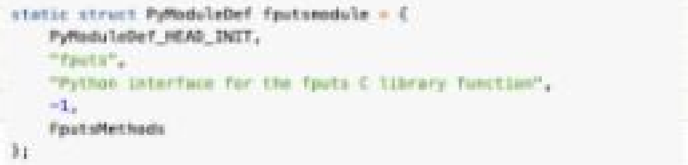
```c
PyMODINIT_FUNC PyInit_fputs(void) {
    return PyModule_Create(&fputsmodule);
}
```



```c
#include <Python.h>

static PyObject *method_fputs(PyObject *self, PyObject *args) {
    char *str, *filename = NULL;
    int bytes_copied = -1;

    /* Parse arguments */
    if(!PyArg_ParseTuple(args, "ss", &str, &filename)) {
        return NULL;
    }

    FILE *fp = fopen(filename, "w");
    bytes_copied = fputs(str, fp);
    fclose(fp);

    return PyLong_FromLong(bytes_copied);
}
```



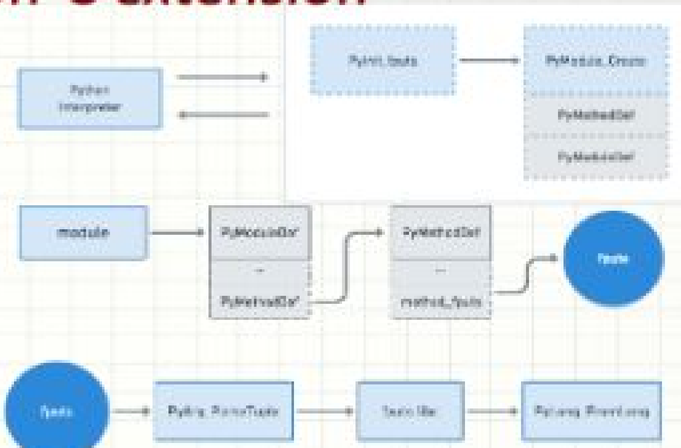
```c
static PyMethodDef FputsMethods[] = {
    {"fputs", method_fputs, METH_VARARGS, "Python interface for fputs
    {NULL, NULL, 0, NULL}
};
```

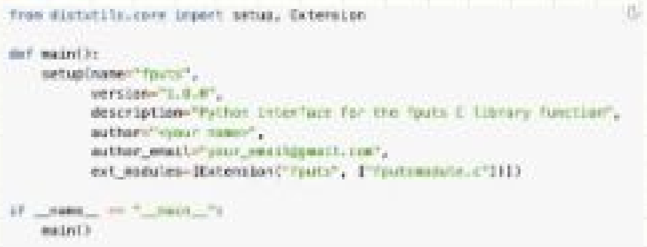


```c
static struct PyModuleDef fputsmodule = {
    PyModuleDef_HEAD_INIT,
    "fputs",
    "Python interface for the fputs C library function",
    -1,
    FputsMethods
};
```

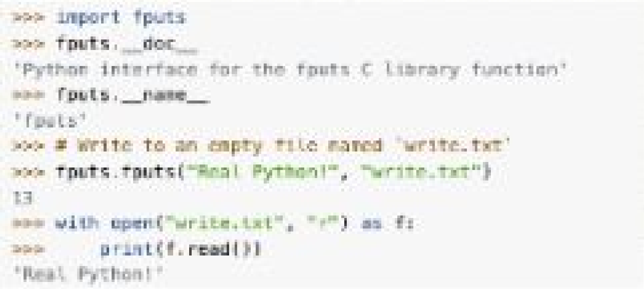
Building a Python-C extension

- This is how it appears
- Create setup.py
- Compile and Install
  - python3_setup.py_install
- Use in your code





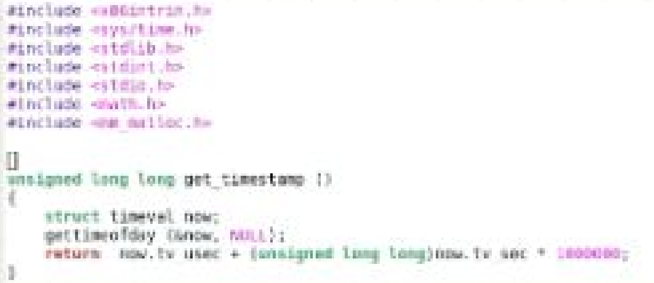
```
from distutils.core import setup, Extension

def main():
    setup(name="fputs",
          version="1.0.0",
          description="Python interface for the fputs C library function",
          author="<your name>",
          author_email="your_email@gmail.com",
          ext_modules=[Extension("fputs", ["fputsmodule.c"])])

if __name__ == "__main__":
    main()
```



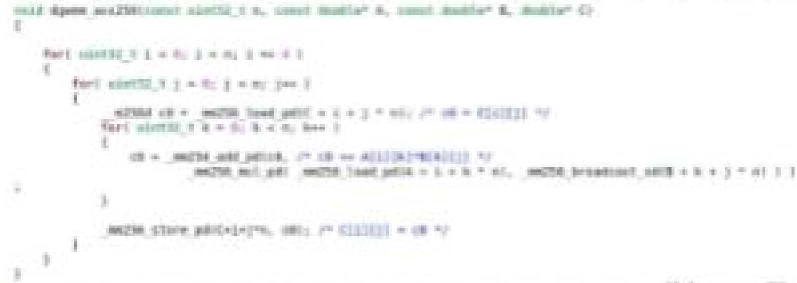
```
>>> import fputs
>>> fputs.__doc__
'Python interface for the fputs C library function'
>>> fputs.__name__
'fputs'
>>> # Write to an empty file named 'write.txt'
>>> fputs.fputs("Real Python!", "write.txt")
13
>>> with open("write.txt", "r") as f:
>>>     print(f.read())
'Real Python!'
```

https://realpython.com/build-python-c-extension-module/

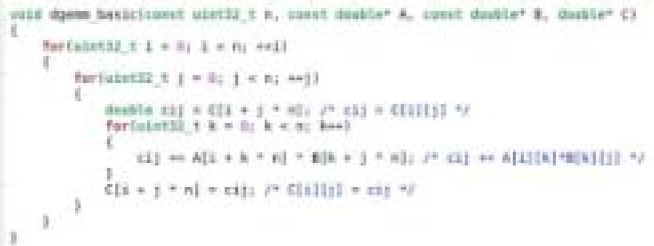
Accelerate matrixMultiply



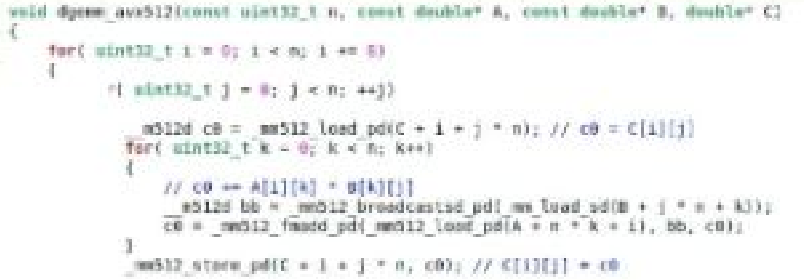
```c
#include <x86intrin.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include <mm_malloc.h>

[]
unsigned long long get_timestamp ()
{
    struct timeval now;
    gettimeofday (&now, NULL);
    return  now.tv_usec + (unsigned long long)now.tv_sec * 1000000;
}
```



```c
void dgemm_avx256(const uint32_t n, const double* A, const double* B, double* C)
{

    for( uint32_t i = 0; i < n; i += 4 )
    {
        for( uint32_t j = 0; j < n; j++ )
        {
            __m256d c0 = _mm256_load_pd(C + i + j * n); /* c0 = C[i][j] */
            for( uint32_t k = 0; k < n; k++ )
            {
                c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
                        _mm256_mul_pd( _mm256_load_pd(A + i + k * n), _mm256_broadcast_sd(B + k + j * n) ) )
;
            }

            _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
        }
    }
}
```



```c
void dgemm_basic(const uint32_t n, const double* A, const double* B, double* C)
{
    for(uint32_t i = 0; i < n; ++i)
    {
        for(uint32_t j = 0; j < n; ++j)
        {
            double cij = C[i + j * n]; /* cij = C[i][j] */
            for(uint32_t k = 0; k < n; k++)
            {
                cij += A[i + k * n] * B[k + j * n]; /* cij += A[i][k]*B[k][j] */
            }
            C[i + j * n] = cij; /* C[i][j] = cij */
        }
    }
}
```



```c
void dgemm_avx512(const uint32_t n, const double* A, const double* B, double* C)
{
    for( uint32_t i = 0; i < n; i += 8)
    {
            for( uint32_t j = 0; j < n; ++j)

              __m512d c0 = _mm512_load_pd(C + i + j * n); // c0 = C[i][j]
             for( uint32_t k = 0; k < n; k++)
             {
                 // c0 += A[i][k] * B[k][j]
                 __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B + j * n + k));
                 c0 = _mm512_fmadd_pd(_mm512_load_pd(A + n * k + i), bb, c0);
             }
             _mm512_store_pd(C + i + j * n, c0); // C[i][j] = c0
```

Accelerate matrixMultiply

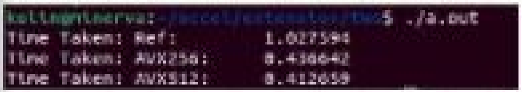
```c
int main(){
        uint32_t trial_no = 11;
        uint32_t n = 32 * 20;

        double *a;
        double *b;
        double *c;
        unsigned long long t0;
        unsigned long long t1;
        unsigned long long t;
struct timeval  tv1, tv2;
```

```c
double result;

a = (double*) _mm_malloc (n * n * sizeof(double), 64);
b = (double*) _mm_malloc (n * n * sizeof(double), 64);
c = (double*) _mm_malloc (n * n * sizeof(double), 64);
```

```c
    for(uint32_t i = 0; i < n * n; ++i)
    {
            a[i] = rand();
    }
```

```c
for(uint32_t i = 0; i < trial_no; i++)
{

        gettimeofday(&tv1, NULL);
        dgemm_basic(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
}
result =  result / (double)(trial_no);
printf("Time Taken: Ref:\t%f\n",result);
```

```c
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{

        gettimeofday(&tv1, NULL);
        dgemm_avx256(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
 }
result =  result / (double)(trial_no);
printf("Time Taken: AVX256:\t%f\n",result);
```



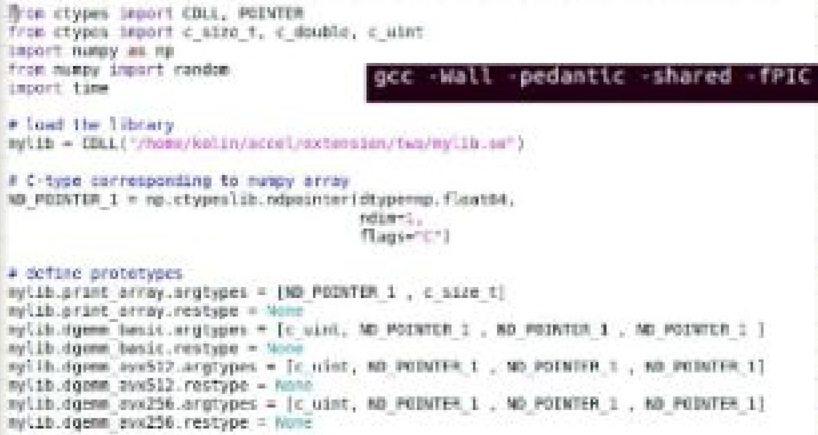
```
kolli@minerva:[illegible]$ ./a.out
Time Taken: Ref:          1.027594
Time Taken: AVX256:       0.436642
Time Taken: AVX512:       0.412659
```

```c
result=0;
for(uint32_t i = 0; i < trial_no; i++)
{
        gettimeofday(&tv1, NULL);
        dgemm_avx512(n, a, b, c);
        gettimeofday(&tv2, NULL);
        result += (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec - tv1.tv_sec);
}
result =  result / (double)(trial_no);
printf("Time Taken: AVX512:\t%f\n",result);
```

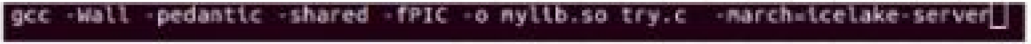
```
gcc -Wall -pedantic matmul-3.c  -march=icelake-server
```

Using shared Objects

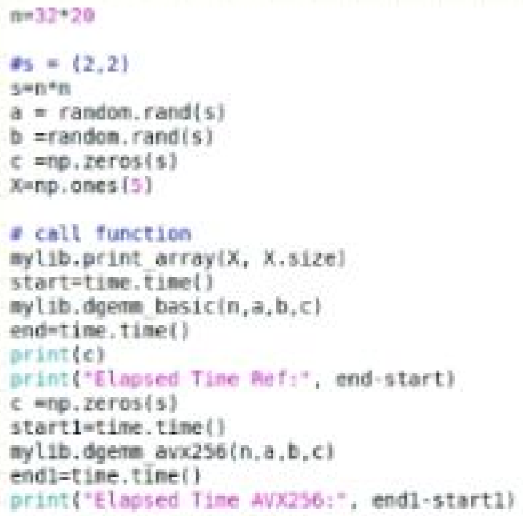
Indian Institute of Technology Delhi



```
from ctypes import CDLL, POINTER
from ctypes import c_size_t, c_double, c_uint
import numpy as np
from numpy import random
import time

# load the library
mylib = CDLL("/home/kolin/accel/extension/two/mylib.so")

# C-type corresponding to numpy array
ND_POINTER_1 = np.ctypeslib.ndpointer(dtype=np.float64,
                                      ndim=1,
                                      flags="C")

# define prototypes
mylib.print_array.argtypes = [ND_POINTER_1 , c_size_t]
mylib.print_array.restype = None
mylib.dgemm_basic.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1 ]
mylib.dgemm_basic.restype = None
mylib.dgemm_avx512.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1]
mylib.dgemm_avx512.restype = None
mylib.dgemm_avx256.argtypes = [c_uint, ND_POINTER_1 , ND_POINTER_1 , ND_POINTER_1]
mylib.dgemm_avx256.restype = None
```



```
gcc -Wall -pedantic -shared -fPIC -o mylib.so try.c  -march=icelake-server
```



```
n=32*20

#s = (2,2)
s=n*n
a = random.rand(s)
b =random.rand(s)
c =np.zeros(s)
X=np.ones(5)

# call function
mylib.print_array(X, X.size)
start=time.time()
mylib.dgemm_basic(n,a,b,c)
end=time.time()
print(c)
print("Elapsed Time Ref:", end-start)
c =np.zeros(s)
start1=time.time()
mylib.dgemm_avx256(n,a,b,c)
end1=time.time()
print("Elapsed Time AVX256:", end1-start1)
```