Announcements

- **Program #6**
  - is due done week from today
  - small change to getCitySchedule() return value
- **Final Exam**
  - **Saturday, May 12**
  - **4:00-6:00 pm**
  - in **PHY 1412**

Optimization Wrap-up from Last Time

- **Optimizations you should do regardless of processor/compiler**
  - Code Motion (moving out of the loop)
  - Reducing procedure calls
  - Unneeded memory usage
  - Share Common sub-expressions
- **Machine-Dependent Optimization**
  - Pointer Code
  - Unrolling
  - Enabling instruction level parallelism
Instruction Control Unit

- **Retirement Unit**
- **Register File**
- **Fetch Control**
- **Instruction Decode**
- **Instruction Cache**

Execution Unit

- **Instruction Control Unit**
- **Operations**
- **Integer/Branch**
- **General Integer**
- **FP add**
- **FP mult/div**
- **Load**
- **Store**

Operations Results

- **Data Cache**

Address & Data
### A simple 5-stage pipeline

- **Instruction pipeline:**
  - IF: instruction fetch (from memory)
  - ID: instruction decode and register fetch
  - X: ALU execution and branch condition evaluation
  - M: memory reference, either load or store
  - WB: results written back (to registers)

- **Notes:**
  - each stage ideally takes one cycle.
  - so a single instruction takes 5 cycles, but....
  - 100 instructions take 104 cycles
    - approach 1 cycle/instruction
    - only in an ideal world
      - memory stages (IF, M) can take more than one cycle
      - mis-predicted branches cause stalls
  - Consider Both Latency and Issue Time

### Pipelining assuming no branching and no dependencies

- **Simple 5-stage pipeline**

| Time | 1      | 2      | 3      | 4      | 5      | 6      | 7      | 8      | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td>WB</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td>WB</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td>WB</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelining, and Branching Penalties

- Simple 5-stage pipeline

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bra</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>tossed</td>
<td>tossed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>tossed</td>
<td>tossed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch target known here

pipeline stalls (not a good thing)

from branch target

Enhancing Parallelism

- Dependences hurt pipeline performance

```c
int sum = 0, arr[], i;

for (i = 0; i < 100; i++) {
    sum += arr[i];
}
```

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>add R2 R3 R2</td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td>WB</td>
<td>ID</td>
<td>X</td>
</tr>
<tr>
<td>add R2 R4 R2</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>ID</td>
<td>X</td>
</tr>
</tbody>
</table>
Enhancing Parallelism

- Dependences hurt pipeline performance,
  - so use loop unrolling (or loop unwinding)
  - Often done by compiler not programmer

```c
int sum = 0, arr[100], i;

for (i = 0; i < 99; i+=3) {
    suma += arr[i];
    sumb += arr[i+1];
    sumc += arr[i+2];
}

sum = suma + sumb + sumc + arr[99];
```

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>X</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>R2 R6</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>R3 R7</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>R4 R8</td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>R2 R9</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>R3 R10 R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Spilling

- register spilling is when not enough registers
  - assume 2 general-purpose registers what happens with:
    ```c
    for (i = 0; i < ...; i++) {
        arr[i] += a + b + c;
    }
    ```
  - Some values are temporarily stored on the stack
    - Causes a significant performance drop
  - rewrite as so that the array element stays in the register
    ```c
    for (i = 0; i < ...; i++) {
        arr[i] += a;
        arr[i] += b;
        arr[i] += c;
    }
    ```
Memory Cache

- Try to keep frequently used things close at hand
  - Can make some small amount of memory really fast
- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: Recently referenced items are likely to be referenced in the near future.
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.

Locality Example

- Data
  - Reference array elements in succession (stride-1 reference pattern): Spatial locality
  - Reference sum each iteration: Temporal locality
- Instructions
  - Reference instructions in sequence: Spatial locality
  - Cycle through loop repeatedly: Temporal locality

```plaintext
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```
Memory Hierarchy

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Cost to access (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>1</td>
</tr>
<tr>
<td>L1/L2 cache</td>
<td>2-10</td>
</tr>
<tr>
<td>L3 cache</td>
<td>10-100</td>
</tr>
<tr>
<td>memory</td>
<td>50-300</td>
</tr>
<tr>
<td>remote memory (BUS)</td>
<td>100-500</td>
</tr>
</tbody>
</table>

More Detailed Memory Hierarchy

- **Registers**: CPU registers hold words retrieved from cache memory.
- **On-chip L1 cache (SRAM)**: L1 cache holds cache lines retrieved from the L2 cache.
- **Off-chip L2 cache (SRAM)**: L2 cache holds cache lines retrieved from memory.
- **Main memory (DRAM)**: Main memory holds disk blocks retrieved from local disks.
- **Remote Main memory (DRAM)**
- **Local secondary storage (local disks)**: Local disks hold files retrieved from disks on remote network servers.
- **Remote secondary storage (distributed file systems, Web servers)**: Larger, slower, and cheaper (per byte) storage devices.
- **Smaller, faster, and costlier (per byte) storage devices**
Time

- On a computer there are many ways measure time
  - Wall time
    - always running
  - Process time
    - time your process was running
    - doesn't count:
      - time when your process was stopped for others
      - time when your program stopped to wait for I/O
      - time spent running in OS kernel (system calls)
- What time to use depends on what you are measuring

Timing a Program

- To Time an entire program
  - time <program name> <program arguments>
  - prints out time in the form:
    - 2.23u 0.26s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
    - first two numbers are user and system time
    - third number is wall time
    - fourth is percentage of wall time (user+system)/wall
    - remainder are paging and I/O statistics
- Provides Some idea about timing
  - hard to know what to do about it
  - what functions are taking most of the time?
Representing Time Of Day

- How to deal with
  - timezones
  - daylight savings times
  - computers moving timezones
  - leap seconds?
- Answer:
  - keep time internally a seconds + fractions of a second
    - 2 32 bits values work nicely for 1ns accuracy for > 100 years
  - use a reference starting point
    - called epoch - midnight 1/1/1970
  - keep all time in UTC form until printed
    - no time zones or daylight savings time to deal with

Adding Timing Calls to your program

- Wall Time
  - int gettimeofday(struct timeval *tv, struct timezone *tz);
  - tv is a struct of time tv_sec and tv_usec (10^-6 seconds)
  - tz is no longer used (pass null)
- Process Time
  - int getrusage(int who, struct rusage *usage);
  - who is RUSAGE_SELF or RUSAGE_CHILDREN
    - children is all terminated children
  - rusage contains fields for
    - struct timeval ru_utime; /* user time used */
    - struct timeval ru_stime; /* system time used */
    - various other OS stats
Example Measuring Time

```c
main() {
    struct rusage startRU, endRU;
    struct timeval startWall, endWall;

    gettimeofday(&startWall, NULL);
    getrusage(&startRU);
    /* code to time */

    gettimeofday(&startWall, NULL);
    getrusage(&startRU);
    /* compute difference */
}
```