Lab Assignment 4 Due March 22 (Friday)

- Lab Assignment 4
  - Due Friday, March 22
  - Branch prediction in MIPS implementation in Verilog
  - Global and hybrid branch predictors
  - All labs are individual assignments
  - No collaboration; please respect the honor code

- Extra credit: Optimize for execution time!
  - Top assignments with lowest execution times will get extra credit.
  - And, it will be fun to optimize...
Lab 3 Grade Distribution

- Average: 62/100
Lab 3 Performance Competition Results

- 1000-instruction test case
- 5 extra credit submissions resulted in the correct output

6. Albert Wang (amwang): 460350ns
3. Joseph Carlos (jcarlos): 202050ns
3. Andrew Pfeifer (apfeifer): 202050ns
3. Xiao Bo Zhao (xiaoboz): 202050ns
2. Martin Gao (yiang): 201950ns
1. Andrew Mort (amort): 201850ns
Lab 2 Extra Credit Recognition

- Microprogrammed MIPS design

1. Andrew Pfeifer (apfeifer)
Important: Lab Late Day Policy Adjustment

- Please keep submitting the labs
- Even if you have used **all** your late days
- If you have already exhausted your 5 late days and still submit a future lab late, you will still be able to get full credit

- We have adjusted the late day policy as follows
  - Everyone gets 5 additional late days for future labs (including Lab 4)
  - Each late day beyond all exhausted late days costs you 15% of the full credit of the lab
A Note on Labs

- Please talk with us:
  - if you are having difficulties with labs
  - if you would like to submit Lab 3 and get a regrade

- Attend lab sessions to get help from the TAs

- Our goal is to enable you learn the material
  - Even if late!
Homework 3 Scores
Homework 5

- Will be assigned later today
- **Due April 1**
- Topics: Virtual memory, SIMD, Caching, ...
Recitation Session this Friday

- Recitation this Friday will cover Midterm I solutions
  - As well as anything else you will ask
Readings for Next Week

- Memory Hierarchy and Caches

- Cache chapters from P&H: 5.1-5.3

- Memory/cache chapters from Hamacher+: 8.1-8.7

- An early cache paper by Maurice Wilkes
Last Lecture

- SIMD Processing
- GPU Fundamentals
Today

- Wrap up GPUs
- VLIW

If time permits
- Decoupled Access Execute
- Systolic Arrays
- Static Scheduling
Approaches to (Instruction-Level) Concurrency

- Pipelined execution
- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing
- VLIW

- Systolic Arrays
- Decoupled Access Execute
Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)
Review: High-Level View of a GPU
Review: Concept of “Thread Warps” and SIMT

- Warp: A set of threads that execute the same instruction (on different data elements) \(\rightarrow\) SIMT (Nvidia-speak)
- All threads run the same kernel
- Warp: The threads that run lengthwise in a woven fabric ...
Review: Loop Iterations as Threads

for (i=0; i < N; i++)
    C[i] = A[i] + B[i];

Process 1
    load
    load
    add
    store
    load
    load
    add
    store
    load
    load
    add
    store

Process 2
    load
    load
    add
    store
    load
    load
    add
    store

Slide credit: Krste Asanovic
Same instruction in different threads uses thread id to index and access different data elements

Let’s assume $N=16$, $\text{blockDim}=4 \rightarrow 4$ blocks
Review: Sample GPU SIMT Code (Simplified)

CPU code

```c
for (ii = 0; ii < 100; ++ii) {
}
```

CUDA code

```c
// there are 100 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```

Slide credit: Hyesoon Kim
**CPU Program**

```c
void add_matrix
( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}

int main () {
    add_matrix (a, b, c, N);
}
```

**GPU Program**

```c
__global__ add_matrix
( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index]+b[index];
}

int main() {
    dim3 dimBlock( blocksize, blocksize) ;
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```
Review: Latency Hiding with “Thread Warps”

- **Warp**: A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- No OS context switching

- Memory latency hiding
  - Graphics has millions of pixels

Slide credit: Tor Aamodt
Review: Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware
Review: SPMD

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

- Essentially, multiple instruction streams execute the same program
  - Each program/procedure can 1) execute a different control-flow path, 2) work on different data, at run-time
  - Many scientific applications programmed this way and run on MIMD computers (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD computer
SPMD Execution on SIMD Hardware

- NVIDIA calls this “Single Instruction, Multiple Thread” (“SIMT”) execution
Control Flow Problem in GPUs/SIMD

- GPU uses SIMD pipeline to save area on control logic.
  - Group scalar threads into warps

- Branch divergence occurs when threads inside warps branch to different execution paths.

Slide credit: Tor Aamodt
Branch Divergence Handling (I)

Slide credit: Tor Aamodt
A;
if (some condition) {
    B;
} else {
    C;
}
D;

Control Flow Stack

<table>
<thead>
<tr>
<th>Next PC</th>
<th>Recv PC</th>
<th>Amask</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>--</td>
<td>1111</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>1110</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>0001</td>
</tr>
</tbody>
</table>

Execution Sequence

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Time

Slide credit: Tor Aamodt
Dynamic Warp Formation

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warp at divergence
  - Enough threads branching to each path to create full new warps
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)

Dynamic Warp Formation Example

A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Legend

- Execution of Warp x at Basic Block A
- Execution of Warp y at Basic Block A

A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Baseline

Dynamic Warp Formation

Slide credit: Tor Aamodt
What About Memory Divergence?

- Modern GPUs have caches
- Ideally: Want all threads in the warp to hit (without conflicting with each other)
- Problem: One thread in a warp can stall the entire warp if it misses in the cache.

- Need techniques to
  - Tolerate memory divergence
  - Integrate solutions to branch and memory divergence
NVIDIA GeForce GTX 285

- **NVIDIA-speak:**
  - 240 stream processors
  - “SIMT execution”

- **Generic speak:**
  - 30 cores
  - 8 SIMD functional units per core
NVIDIA GeForce GTX 285 “core”

- 34 instruction stream decode
- SIMD functional unit, control shared across 8 units
- Multiply-Add
- Multiply
- 64 KB of storage for fragment contexts (registers)

Slide credit: Kayvon Fatahalian
Groups of 32 threads share instruction stream (each group is a Warp)
Up to 32 warps are simultaneously interleaved
Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian
NVIDIA GeForce GTX 285

30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian
VLIW and DAE
Remember: SIMD/MIMD Classification of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor

- **MISD?** Multiple instructions operate on single data element
  - Closest form: systolic array processor?

- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
SISD Parallelism Extraction Techniques

- We have already seen
  - Superscalar execution
  - Out-of-order execution

- Are there simpler ways of extracting SISD parallelism?
  - VLIW (Very Long Instruction Word)
  - Decoupled Access/Execute
VLIW
VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated (contrast with SIMD)

- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction

- Traditional Characteristics
  - Multiple functional units
  - Each instruction in a bundle executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units

- ELI: Enormously longword instructions (512 bits)
SIMD Array Processing vs. VLIW

- Array processor

![Diagram showing SIMD array processing and VLIW execution]

Program Counter → add VR, VR, 1

Instruction Execution:
- add VR[0], VR[0], 1
- add VR[1], VR[1], 1
- add VR[2], VR[2], 1
- add VR[3], VR[3], 1

VLEN = 4
VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
  - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- VLIW (Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design
VLIW Philosophy (II)

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.
Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
  - Most successful commercially

- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones
VLIW Tradeoffs

Advantages

- No need for dynamic scheduling hardware → simple hardware
- No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
- No need for instruction alignment/distribution after fetch to different functional units → simple hardware

Disadvantages

- Compiler needs to find N independent operations
  - If it cannot, inserts NOPs in a VLIW instruction
  - Parallelism loss AND code size increase
- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
- Lockstep execution causes independent operations to stall
  - No instruction can progress until the longest-latency instruction completes
VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
  - Too many NOPs (not enough parallelism discovered)
  - Static schedule intimately tied to microarchitecture
    - Code optimized for one generation performs poorly for next
  - No tolerance for variable or long-latency operations (lock step)

++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
  - Enable code optimizations
++ VLIW successful in embedded markets, e.g. DSP
DAE
Decoupled Access/Execute

- Motivation: Tomasulo’s algorithm too complex to implement
  - 1980s before HPS, Pentium Pro

- Idea: Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.

Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
  - Synchronizes the two upon control flow instructions (using branch queues)

\[
q = 0.0 \\
\text{Do } 1 \quad k = 1, 400 \\
x(k) = q + y(k) \times (r \times z(k+10) + t \times z(k+11))
\]

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCERPT)

```
A7 + -400  . negative loop count
A2 + 0    . initialize index
A3 + 1    . index increment
X2 + r    . load loop invariants
X5 + t    . into registers

loop: X3 + z + 10, A2 . load z(k+10)
X7 + z + 11, A2 . load z(k+11)
X4 + X2  *f X3 . r*z(k+10)-flt. mult.
X3 + X5  *f X7 . t * z(k+11)
X7 + y, A2 . load y(k)
X6 + X3  +f X4 . r*z(x+10)+t*z(k+11))
X4 + X7  *f X6 . y(k) * (above)
A7 + A7 + 1 . increment loop counter
x, A2 + X4 . store into x(k)
A2 + A2 + A3 . increment index

JAM loop  . Branch if A7 < 0
```

Access | Execute
---|---
AEQ + z + 10, A2 | X4 + X2  *f AEQ
AEQ + z + 11, A2 | X3 + X5  *f AEQ
AEQ + y, A2 | X6 + X3  +f X4
A7 + A7 + 1 | EAQ + AEQ  *f X6
x, A2 + EAQ | *
A2 + A2 + A3 | *

Fig. 2b. Compilation onto CRAY-1-like architecture

Fig. 2c. Access and execute programs for straight-line section of loop
Decoupled Access/Execute (III)

- Advantages:
  + Execute stream can run ahead of the access stream and vice versa
    + If A takes a cache miss, E can perform useful work
    + If A hits in cache, it supplies data to lagging E
    + Queues reduce the number of required registers
  + Limited out-of-order execution without wakeup/select complexity

- Disadvantages:
  -- Compiler support to partition the program and manage queues
    -- Determines the amount of decoupling
  -- Branch instructions require synchronization between A and E
  -- Multiple instruction streams (can be done with a single one, though)
Astronautics ZS-1

- Single stream steered into A and X pipelines
- Each pipeline in-order

Astronautics ZS-1 Instruction Scheduling

- **Dynamic scheduling**
  - A and X streams are issued/executed independently
  - Loads can bypass stores in the memory unit (if no conflict)
  - Branches executed early in the pipeline
    - To reduce synchronization penalty of A/X streams
    - Works only if the register a branch sources is available

- **Static scheduling**
  - Move compare instructions as early as possible before a branch
    - So that branch source register is available when branch is decoded
  - Reorder code to expose parallelism in each stream
  - Loop unrolling:
    - Reduces branch count + exposes code reordering opportunities
Loop Unrolling

- **Idea:** Replicate loop body multiple times within an iteration
  + Reduces loop maintenance overhead
    - Induction variable increment or loop condition test
  + Enlarges basic block (and analysis scope)
    - Enables code optimization and scheduling opportunities

-- What if iteration count not a multiple of unroll factor? (need extra code to detect this)
-- Increases code size
Systolic Arrays
Why Systolic Architectures?

- **Idea:** Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory.

- Similar to an assembly line
  - Different people work on the same car
  - Many cars are assembled simultaneously
  - Can be two-dimensional

- **Why?** Special purpose accelerators/architectures need
  - Simple, regular designs (keep # unique parts small and regular)
  - High concurrency $\rightarrow$ high performance
  - Balanced computation and I/O (memory access)
Systolic Architectures


![Diagram of systolic architecture]

Figure 1. Basic principle of a systolic system.

- Memory: heart
- PEs: cells
- Memory pulses data through cells
Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs → achieve high throughput w/o increasing memory bandwidth requirements

- Differences from pipelining:
  - Array structure can be non-linear and multi-dimensional
  - PE connections can be multidirectional (and different speed)
  - PEs can have local memory and execute kernels (rather than a piece of the instruction)
Systolic Computation Example

- Convolution
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
  - Many image processing tasks

*Given* the sequence of weights \( \{w_1, w_2, \ldots, w_k\} \) and the input sequence \( \{x_1, x_2, \ldots, x_n\} \),

*compute* the result sequence \( \{y_1, y_2, \ldots, y_{n+1-k}\} \) defined by

\[
y_i = w_1x_i + w_2x_{i+1} + \cdots + w_kx_{i+k-1}
\]
Systolic Computation Example: Convolution

- \( y_1 = w_1x_1 + w_2x_2 + w_3x_3 \)
- \( y_2 = w_1x_2 + w_2x_3 + w_3x_4 \)
- \( y_3 = w_1x_3 + w_2x_4 + w_3x_5 \)

Figure 8. Design W1: systolic convolution array (a) and cell (b) where \( w_i \)'s stay and \( x_i \)'s and \( y_i \)'s move systolically in opposite directions.
Systolic Computation Example: Convolution

- Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions
More Programmability

- Each PE in a systolic array
  - Can store multiple “weights”
  - Weights can be selected on the fly
  - Eases implementation of, e.g., adaptive filtering

- Taken further
  - Each PE can have its own data and instruction memory
  - Data memory → to store partial/temporary results, constants
  - Leads to stream processing, pipeline parallelism
    - More generally, staged execution
Pipeline Parallelism

Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.
File Compression Example

Figure 3. File compression algorithm executed using pipeline parallelism
Systolic Array

- **Advantages**
  - Makes multiple uses of each data item $\rightarrow$ reduced need for fetching/refetching
  - High concurrency
  - Regular design (both data and control flow)

- **Disadvantages**
  - Not good at exploiting irregular parallelism
  - Relatively special purpose $\rightarrow$ need software, programmer support to be a general purpose model
The WARP Computer

- HT Kung, CMU, 1984-1988

- Linear array of 10 cells, each cell a 10 Mflop programmable processor
- Attached to a general purpose host machine
- HLL and optimizing compiler to program the systolic array
- Used extensively to accelerate vision and robotics tasks

The WARP Computer

Figure 1: Warp system overview
The WARP Computer

Figure 2: Warp cell data path
Systolic Arrays vs. SIMD

- Food for thought...
Some More Recommended Readings

- Recommended: