Objective and Outline

Objective: …

Outline:
1) Acknowledgements
2) …

Acknowledgement

This lecture note has been obtained from similar courses all over the world. I wish to thank all the professors who created such good works on those lecture notes. Without them, these slides could have not been presented to you.
Software Design for Low-Power

- Software dictates much of hardware activity (see next slide).
- Must optimize software at several levels of abstraction.
- Ultimately involves hardware/software trade-off.
- Need software power estimation method.

Sources of Software Power Dissipation

1. Memory system uses power (1/10 to 1/4) in portable computers
   - Dominates in video processing
2. System bus switching activity
   - Is controlled by software
3. ALU and FPU data paths
   - Use good scheduling to avoid being activated at all times.
4. Control logic and clock
   - Use shortest possible program to do the computation

→ In addition, Software control of hardware:
   1. Reduces power of idle components
   2. Controls power saving modes
Memory Accesses Are Expensive!

- **Reason:** data bus (especially word data lines with high fan-out), address bus, and row and column decoders are highly capacitive

Reducing power

1. Mapping of data structures to memory banks provides the possibility of parallel loading of the multiple words (which are more energy efficient)
2. Memory access patterns greatly affect cache performance
   - Cache more power efficient than main memory.
   - Closer to CPU and smaller than main memory.

Methods of Software Power Estimation for Code

1. Lower level: Use gate-level simulation & power estimation on gate-level description of the hardware running the code – more accurate
2. Higher level: Look at the frequency of each type of instruction/sequence (execution profile).
   - Features:
     - Use lookup table.
     - Model ALUs, register files, etc.
     - Use bus switching activity exclusively for the instructions.
     - Must know bus architecture,_OPCODEs, representative input data to program, mapping of code/data to address space.
       - Characterizing instruction power **empirically** on real hardware.
Instruction Level Power Analysis

- For each instruction, need to measure:
  1. Base power cost (independent of prior state)
  2. Prior state cost – needs large-scale analysis/simulation:
    - Circuit state effects – due to localized processor state change from the execution of one instruction to the another.
      - Example: ADD A ← B, C
      - Includes:
        1. Change of instruction bus state.
        2. Switching of control lines (enable multiplier and disable adder).
        3. ALU mode changes.
        4. Routing costs to/from register file (same instructions with different operands)
  3. Pipeline stalls, buffer stalls, cache misses (stalls) also should be considered.
    - A stall (freeze) is the delay in cycles caused due to any of the hazards.

Pipeline Stall, Cache Miss/Stall

One of the tradeoffs with making a deep pipelining is what is called a stall. What happens if you have a really deep pipe, and one of the stages is dependent on another instructions data, and that instruction hasn't been completed yet. This is what a stall could look like. Instruction 3 cannot start until something it needs from Instruction 2 is finished (written). So there is this big fat bubble of nothing going on, while instruction 3 is waiting for Instruction 2 to complete (it is blocking up the queue/pipe).

Another type of stall, is called a cache stall. Each instruction has to be loaded, and have all its data loaded. If the data that an instruction needs (or the instruction itself) is not loaded, then we have to wait for the "fetch" part of the processor to go and get it. The fetch logic tries to think ahead, and tells the cache what data it is going to need in a few instructions, and get the cache to "pre-etch" it -- but sometimes it misses.

If the data (or instruction) that we need is in the L1 cache we load the instruction/data from the L1 cache and go on (takes a single cycle). But there are cases where the L1 cache hasn't preloaded what we need -- then we have an (L1) cache stall. Then we have to go to L2 cache (which takes more time). The L2 cache is bigger (but slower) yet it's likely to have what we need -- if it does, we get what we need and go on (which takes a few cycles, say 8 - 14 cycles). But if it isn't in the L2 cache then we have to go to main memory -- this is an L2 cache stall (or a complete cache stall). Since processors are so much faster than main memory, this can take another 10 or 20 cycles (or more) -- which is a really big stall. That is like 30+ instructions total that we could have completed all just wasted waiting for memory.
Instruction Analysis Example

\[ E_P = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \]  

- \( E_P \) = Overall energy cost of program
- \( B_i \) = Base cost of instruction type \( i \)
- \( O_{i,j} \) = Cost of instruction type \( i \) followed by type \( j \)
- \( O_{i,j} = O_{j,i} \)
- \( E_k \) = Costs of pipeline stalls and cache misses

A Detailed Example

- Simple DSP with 4 registers A, B, C, D

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Base Cost (pJ)</th>
<th>LOAD</th>
<th>DLOAD</th>
<th>ADD</th>
<th>MULT</th>
<th>LOAD ADD</th>
<th>LOAD MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>1.98</td>
<td>0.13</td>
<td>0.15</td>
<td>1.19</td>
<td>0.92</td>
<td>1.25</td>
<td>1.06</td>
</tr>
<tr>
<td>DLOAD</td>
<td>2.37</td>
<td>0.17</td>
<td>1.19</td>
<td>0.92</td>
<td>1.32</td>
<td>1.16</td>
<td>1.06</td>
</tr>
<tr>
<td>ADD</td>
<td>0.99</td>
<td>0.26</td>
<td>0.53</td>
<td>0.86</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td>1.19</td>
<td>0.66</td>
<td>0.79</td>
<td>0.96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD; ADD</td>
<td>2.10</td>
<td>0.40</td>
<td>0.53</td>
<td>0.79</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD; MULT</td>
<td>2.25</td>
<td>0.79</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( T = 25 \text{ ns} \)

DLOAD: Dual load
Example Concluded

Evaluate \((x \times y) + z\)

\[
\begin{align*}
DLOAD & \leftarrow x, \ B \leftarrow y & \# \text{ prior instruction was ADD} \\
LOAD & \leftarrow z; \ MULT & \leftarrow A, B & \# \text{ LOAD & MULT packed together} \\
ADD & \leftarrow C, D & \# A \text{ now contains } (x \times y) + z
\end{align*}
\]

The energy and power estimate can be tabulated as follows:

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Base Cost (pJ)</th>
<th>Circuit State (pJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLOAD</td>
<td>2.37</td>
<td>1.19</td>
</tr>
<tr>
<td>LOAD;MULT</td>
<td>2.25</td>
<td>1.06</td>
</tr>
<tr>
<td>ADD</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>Totals</td>
<td>5.61</td>
<td>3.24</td>
</tr>
</tbody>
</table>

Total energy = 8.85 pJ  
Average power = \(8.85 \text{ pJ/75 ns} = 118 \text{ \(\mu\)W}\)

---

Software Power Minimization Techniques

1. Select least expensive instructions/instruction sequences
2. Minimize frequency or cost of memory access
3. Use hardware power minimization features
4. Select best algorithm:
   - Must map well to available hardware
   - Be efficient for problem being solved
   - Must maximize performance & parallel processing

➢ (If) Constraints, (Then) Do:
   - Battery-powered system – minimize total energy dissipated
   - When heat dissipation or reliability is important – minimize instantaneous/average power
Algorithm Transformations

One adder is available

Note: Reduction operations

- Use parallel hardware, but run slowly.

\[
\begin{align*}
\text{sum} & = 0 \\
\text{for } i & = 0 \text{ to } n \\
\text{sum} & = \text{sum} + A(i)
\end{align*}
\]

Two Adders Available

Can be performed at the same time

\[
\begin{align*}
\text{sum} & = 0 \\
\text{for } i & = 0 \text{ to } n/2 - 1 \\
\text{tmp} & = A(2i) + A(2i+1) \\
\text{sum} & = \text{sum} + \text{tmp}
\end{align*}
\]

It is the tmp from the previous iteration (registered).
Low-Power Integrated Circuit Design
Section 8: Software Design for Low Power

Four Adders Available

\[
\begin{align*}
\text{A(0)} & \quad \text{A(1)} & \quad \text{A(2)} & \quad \text{A(3)} & \quad \text{A(4)} & \quad \text{A(5)} & \quad \text{A(6)} & \quad \text{A(7)} & \quad \text{A(8)} & \quad \text{A(9)} & \quad \text{A(10)} & \quad \text{A(11)} \\
\sum & = 0 \\
\text{for } i = 0 \text{ to } n/4 - 1 \\
\text{tmp1} & = \text{A}(4i) + \text{A}(4i+1) \\
\text{tmp2} & = \text{A}(4i+2) + \text{A}(4i+3) \\
\text{tmp} & = \text{tmp1} + \text{tmp2} \\
\sum & = \sum + \text{tmp} \\
\end{align*}
\]

Can be performed at the same time

Minimizing Memory Access Costs

- Memory causes power and performance bottleneck
- Minimize # of accesses needed by algorithm
- Minimize total memory size needed by algorithm
- Put memory accesses as close as possible to CPU
  - Register, then cache, external RAM last
- Efficiently use memory bandwidth:
  - Use multiple-word parallel loads, not single word loads
**Improve Loop Nesting and Operation Order**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
</table>
| FOR \(i := 1\) TO \(N\) DO  
  \(B[i] := f(A[i])\);  
FOR \(i := 1\) TO \(N\) DO  
  \(C[i] := g(B[i])\); |
| FOR \(i := 1\) TO \(N\) DO BEGIN  
  \(B[i] := f(A[i])\);  
  \(C[i] := g(B[i])\);  
END; |

- Vector \(B\) too large to store in registers \(\rightarrow\) used memory transfers, instead.
- Loop rearrangement allowed intermediate \(B\) to stay in general register.
  - Got rid of \(2N\) memory accesses for \(B\) and \(N\) memory locations not needed.

---

**Reduce Space Requirements**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
</table>
| FOR \(i := 1\) TO \(N\) DO  
  \(B[i] := f(A[i])\);  
FOR \(j := 1\) TO \(N\) DO  
  \(D := g(C[j], D)\); |
| FOR \(j := 1\) TO \(N\) DO  
  \(D := g(C[j], D)\);  
FOR \(i := 1\) TO \(N\) DO  
  \(B[i] := f(A[i])\); |

- \(B\) is needed afterward
- \(C\) is not needed afterward – reorder so that \(B\) can overwrite \(C\)
- Only memory space requirement is reduced.
Reduced Intermediate Value Storage

Before

FOR j := 1 to M DO
    FOR i := 1 TO N DO
        A[i][j] := g(A[i][j - 1], D);
    END;
FOR i := 1 to N DO
    OUT[i] := A[i][M];

After

FOR i := 1 TO N DO BEGIN
    FOR j := 1 TO M DO
        A[i][j] := g(A[i][j - 1], D);
    END;
    OUT[i] := A[i][M];
END;

- Reduce storage for intermediate values A[i][M]
  - Reduced from N locations to 1

Dual Memory Load Example

Evaluate \((x \times y) + z\)
Maximize Parallel Loads of Multiple Memory Words

- Note that dual word loads reduce energy by 47%.
- Methods to reduce the power:
  1. Maximize dual loads with memory allocation.
  2. Combine memory accesses with other operations.

Case (a): All variables in the same block → no dual load; only overlap a load and a multiply
Case (b): x and z may loaded in parallel. → Dual load; but no other operation could be overlapped
Case (c): Dual load + Parallel execution (Load and Multiply)

<table>
<thead>
<tr>
<th>RAM A</th>
<th>RAM B</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD B ← x</td>
<td>DLOAD B ← x, A ← z</td>
</tr>
<tr>
<td>LOAD C ← y</td>
<td>LOAD C ← y</td>
</tr>
<tr>
<td>LOAD A ← z;</td>
<td>MULT B ← B, C</td>
</tr>
<tr>
<td>MULT B ← B, C</td>
<td>ADD A ← A, B</td>
</tr>
<tr>
<td>ADD A ← A, B</td>
<td></td>
</tr>
</tbody>
</table>

Energy (pJ) 10.57 9.32 8.85
Power (μW) 105.7 93.2 118.0

Minimize Memory Bandwidth

Use the following methodologies for the power reduction.

- Registers allocation to minimize external memory references
- Cache blocking (An optimization technique that modifies a program so that it works on blocks of data that fit in cache, and deals completely with each block before going on to another.): An example is the transformation on an array computation so that blocks of array elements only have to be read in the cache once.
- Loop unrolling (A programming or compiler strategy whereby instructions that are executed within a loop are copied one or more times to reduce (or eliminate) the number of times the loop is executed. It is a compiler optimization technique in which the body of a loop is replicated several times, and the increment of the loop variable is adjusted to match the unrolling (for example, instead of using array element a(i) in one statement, the loop has four statements using elements a(i), a(i+1), a(i+2), and a(i+3), and then increments i by 4). This reduces the loop overhead by the amount of unrolling. This technique can reduce the overhead associated with looping, but it increases instruction memory usage.): Fix array computations so that blocks of array only read once (as was used in previous examples)
Minimize Memory Bandwidth

- **Register blocking** – Eliminate redundant register loads
- **Recurrence detection & optimization**: Use registers for values carried *over from one recursion level* to next
- **Compact multiple memory** references into 1 reference
- Cache locking (cache locking techniques, which load the cache contents with some values and lock it in order to ensure that the contents will remain unchanged) is the ability to prevent some or all of a microprocessor’s instruction or data cache from being overwritten.

It is a feature that can force a block of code to run entirely from cache to avoid external memory accesses. If a processor supports it, therefore,
  - Lock data into cache.
  - Prevents memory reads/writes from going to main memory.

Instruction Selection/Ordering

- **Instruction packing**
  - Single instruction does **both ALU operation** and **memory data transfer** (as was used in previous examples).
  - Much of the instruction overhead **not duplicated** when operations run in parallel.
  - Concurrent execution of integer & floating point Ops.

Easier to do in VLIW (Very Long Instruction Word) and superscalar architectures.

- Reorder instructions to **minimize circuit state effects** (compiler task).
  - Most significant for DSP units.
  - Accumulator spilling and mode switching may be problematic.
Operand Swapping/Ordering

- **Swapping** – minimizes switching of functional unit inputs by selecting the proper place for each operand.
  
  Example: $x + 7$ and $y + 7$ – keep 7 on same adder input.

- **Ordering most significant if commutative operands not treated symmetrically by hardware.**

  Example: *Booth multiplier*
  
  - 2\textsuperscript{nd} operand bit pattern determines # additions and subtractions (called *recoding weight*).
  - Put operand with lowest recoding weight on 2\textsuperscript{nd} operand.
  - Save (10-30%) of the power.

Power Management

- **Software can often control processor power-down modes:**
  
  - User interfaces – activity comes in bursts
  
    ➔ When system idle time exceeds threshold, likely to continue to be idle ➔ Start shutdown.

  - During the time waiting to exceed the threshold, power is consumed ➔ Minimize the threshold.

  - Predictive techniques use the previous history to determine the threshold time for each user.
Power Management

- Example: SPARClite
  - Power-down register masks or enables the clock for:
    - SDRAM, DMA module, FPU, floating-point queues
- Example: Hitachi SH3
  - Standby mode: CPU core stopped, peripheral controller, bus controller, memory refresh continue
  - Sleep mode: Everything but real-time clock stops
- Example: Intel 486SL
  - System Management Mode (SMM): Transparent to operating system and user applications. Software for the SMM, enters an asynchronous interrupt.
  - Can enable, disable, & switch between fast and slow clocks for CPU and ISA bus

More Examples

- Example: PowerPC 603 and 604
  - Dynamic power management (DPM) – removes clock from execution units (saves 8-16%)
  - Static power management (SPM) –
    - Doze – shuts off most function units, keeps bus snooping (it is an activation) enabled to maintain data cache coherence.
    - Nap – shuts off bus snooping and sets wakeup timer, keeps phase-locked loop running to allow quick clock restart.
    - Sleep mode – also shuts off phase-locked loop.

Note: Software control of power management better than pure hardware control.
Automated Low-Power Code Generation

- It is a high-level approach (compiler task):
- Examples of those available to describe DSP algorithms for low power applications using parallelism:
  - Graphical language [Lauwerreins90].
  - Textual language [Hilfinger93].
    ➔ It does not obscure the natural parallelism and data flow.
  - HYPER_LP [Chandrakasan95] – uncovers parallelism and minimizes critical paths.
    ➔ Allows data path supply voltage to be reduced.
  - MASAI – reorganizes loops to minimize memory transfers and size.

Note: DSP Compiler technology: must deal with small register set, irregular data paths, fully using parallel resources ➔ Using low-power techniques more difficult.

A Proposed Cold Scheduling Algorithm

It is an instruction scheduling algorithm that reduces the bus switching activity related to the change in the state when execution moves from one instruction type to another.

It is a list scheduler that prioritizes the selection of each instruction based on the power cost (bus switching activity) of placing that instruction next into the schedule.

1. Allocate registers
2. Pre-assemble: Calculate target addresses, index symbol table, transform instructions to binary
3. Schedule instructions with cold scheduling
4. Post-assemble: Complete assembly

➔ Reduced switching activity 20-30%, performance loss of 2-4%
Another Proposed Scheduling Algorithm

- It had a similar approach as the previous one (instruction scheduling for minimum bus switching activity), but used an as soon as possible packing of instructions plus optimizing memory bank assignments and cold scheduling.
  1. Allocate registers and select instructions.
  2. Build a data flow graph (DFG) for each basic block.
  3. Optimize memory bank assignments by simulated annealing.
  4. Perform as soon as possible (ASAP) packing of instructions.
  5. Perform list scheduling of instructions (similar to cold scheduling).

- Saved 26 to 73% energy compared with no instruction packing and no memory bank assignment optimization.

Instruction Set Co-Design

Some of other efforts at instruction set optimization:
- PEAS-I system.
- Takes (inputs)
  - HDL for CPU and C compiler, assembler, and simulator for CPU
  - Design constraints (chip area, power), hardware module database, sample application program, program data set
- Optimizes the instruction sets (output)
  - Here, different instructions sets for the CPU are tried to find out the best set which leads to the minimum power consumption.
→ Continued to the next
Instruction Set Co-Design

- Optimizes instruction set and implementations for an application program and given data set. Change the HDL for the CPU (output):
  - Starts with core instructions needed for any C program
  - Augments new instructions for C operators (new instructions not already included as a single instruction)
  - Evaluate hardware, micro-program, and software implementations.
  - Estimates power and area of each new instruction implementation.
  - Accounts for pipeline hazards.
  - Optimal selection of implementation methods for each instruction is defined as an integer program which is solved using branch-and-bound search.
  - The power and area contribution of each instruction implementation is estimated and added together to include in the area/power constraints for that specific instruction set implementation.
  - The number of execution cycles of each instruction is multiplied by the frequency of the occurrence in the sample application and added together to form the objective function for the optimization.

Instruction Set Design

- Huang and Despain system:
  - Optimizes instruction set for sample application, too
  - Group micro-operations (MOPS) together to form higher-level instructions.
  - Merge MOPS together as a result of scheduling.
  - MOPS that are scheduled to the same clock cycle are combined.
  - Constrain with instruction bitwidth, instruction set size, and hardware resources.
  - Solved by Simulated Annealing with objective of minimizing execution cycles & instruction set size.
Reconfigurable Computing

- Some of hardware interconnect and logic is modified at the run time for a specific applications.
- Can closely optimize to wide variety of applications.
- Can reconfigure at gate level or at architectural level.
- Usually implemented using FPGAs
- Gives software designer chance to tailor software and processor to fit each other
  - Even after hardware design is fixed

Architectural and Circuit Level Decisions

Memory System Considerations

<table>
<thead>
<tr>
<th>Feature</th>
<th>Low-Power Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total size</td>
<td>Code compaction, algorithm transformations</td>
</tr>
<tr>
<td></td>
<td>⇒ Smaller/ lower C ⇒ faster</td>
</tr>
<tr>
<td>Bank partitioning</td>
<td>Needed if parallel loads are to be used. Determined by application size and data structure</td>
</tr>
<tr>
<td>Wide data bus</td>
<td>Needed for parallel loads</td>
</tr>
<tr>
<td>Proximity to CPU</td>
<td>Reduces C ⇒ makes memory use less costly</td>
</tr>
<tr>
<td>Cache size</td>
<td>Minimize cache misses for application Use techniques such as cache blocking to maintain high spatial and temporal locality</td>
</tr>
<tr>
<td>Cache protocol</td>
<td>Optimize for application</td>
</tr>
<tr>
<td>Cache locking</td>
<td>If significant portions of an application can run entirely from cache, this can be used to prevent any external memory accesses while those portion of the code are executing.</td>
</tr>
</tbody>
</table>
## Architectural and Circuit Level Decisions

### Architectural Considerations

<table>
<thead>
<tr>
<th>Processor class</th>
<th>Application dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP/RISC/CISC</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel Processing VLIW, Superscalar, SIMD, MIMD</th>
<th>Does parallel processing greatly improve performance so that reduced $V_{DD}$ and slower clock can be used?</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Bus architecture</th>
<th>Separating address, data, instruction, I/O busses make it easier to optimize instructions, addressing, and data to minimize bus activity</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Register file size</th>
<th>Eases register allocation, reduces memory accesses, too many increases power</th>
</tr>
</thead>
</table>

RISC: Reduced Instruction Set Computers  
CISC: Complex Instruction Set Computers  
SIMD and MIMD are used for parallel processing:  
SIMD: Single Instruction Multiple Data  
MIMD: Multiple Instruction Multiple Data  
VLIW: Very Long Instruction Word

### Power Management Considerations

<table>
<thead>
<tr>
<th>Software vs. hardware control</th>
<th>Software control lets power management be tailored to application</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Clock/voltage removal</th>
<th>Shutdown modes needed to benefit from minimized execution times</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Clock/voltage reduction</th>
<th>Useful if there is schedule slack for a software task (power reduced during slack)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Guarded evaluation</th>
<th>Latch functional unit inputs to avoid meaningless calculation when not in use</th>
</tr>
</thead>
</table>
Summary

• Estimation of software contribution to power dissipation was discussed.
• Discussed minimizing software power dissipation through:
  – Selecting instruction sequences to minimize switching in CPU and data path.
  – Choosing the best algorithm that is suited to hardware resources.
  – Minimizing memory size and expensive memory accesses.
    • Algorithm transformations
    • Efficient data mapping onto memory
    • Optimally using of memory bandwidth, registers & cache.
  – Optimally using available parallelism for application.
  – Using hardware power management support