Karlsruhe Institute of Technology

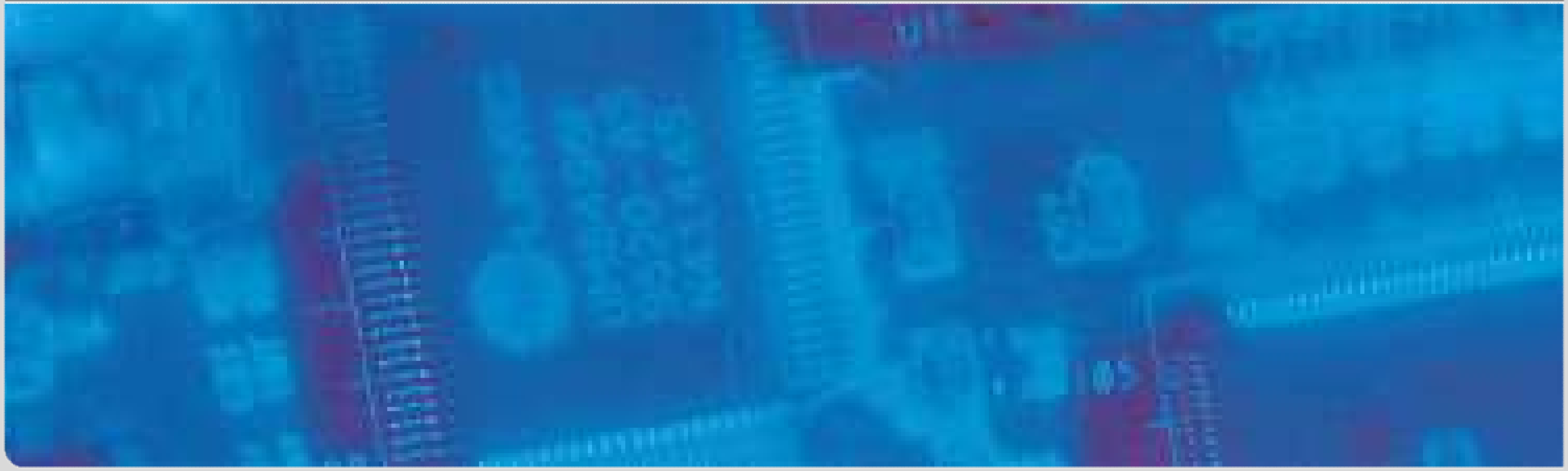
Lower Power Design

Lecture 7: Low Power Software and Compiler

Anuj Pathania on behalf of Prof. Dr. Jörg Henkel

Summer Semester 2017

CES – Chair for Embedded Systems



ces.itec.kit.edu

Organizational Issues

- Slides available for download -
  - http://cesweb.itec.kit.edu/teaching/LPD/s17_slides/
  - Username: student
  - Password: CES-Student
- Homework
  - Read a relevant scientific paper.
  - Discussion next class.
- Oral Exam
  - Make appointment with KIT CES secretary 6-8 weeks in advance.
  - Exam will be in English (or German if told in advance).
  - More information: http://ces.itec.kit.edu/972.php

Lectures

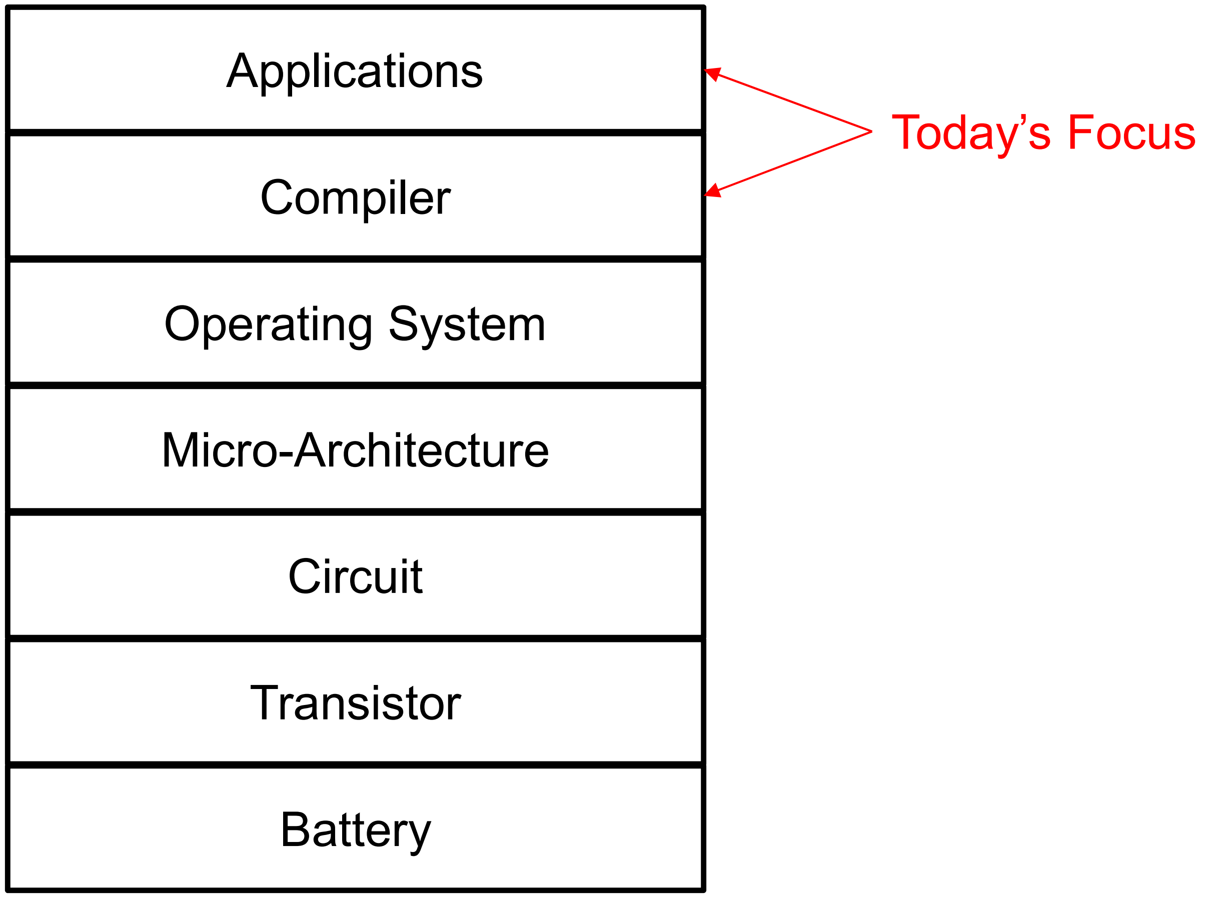
- 27.04.2017 – ~~Lecture 0: Introduction~~
- 04.05.2017 – ~~Lecture 1: Energy Sources~~
- 11.05.2017 – ~~Lecture 2: Battery Modelling Part 1~~
- 18.05.2017 – ~~Lecture 3: Battery Modelling Part 2~~
- 25.05.2017 – Ascension Day (Holiday)
- 01.06.2017 – ~~Hardware Power Optimization and Estimation 1~~
- 08.06.2017 – ~~Hardware Power Optimization and Estimation 2~~
- 15.06.2017 – Corpus Christi (Holiday)
- 22.06.2017 – ~~Hardware Power Optimization and Estimation 3~~
- 29.06.2017 – Low Power Software and Compiler
- 06.07.2017 – Thermal Management
- 13.07.2017 – Aging-Aware Design
- 20.07.2017 – Temperature-Aware Design
- 27.07.2017 – Infrared Camera based Thermal Measurement

Overview for Today

- Software Power Analysis/Estimation
- Software Power Estimation Models
- Optimizing Software for Low Power Through Compilation Phase
  - Instruction Scheduling
  - Compiler Driven DVS

Abstraction Layers

- Different levels of abstractions in embedded systems.

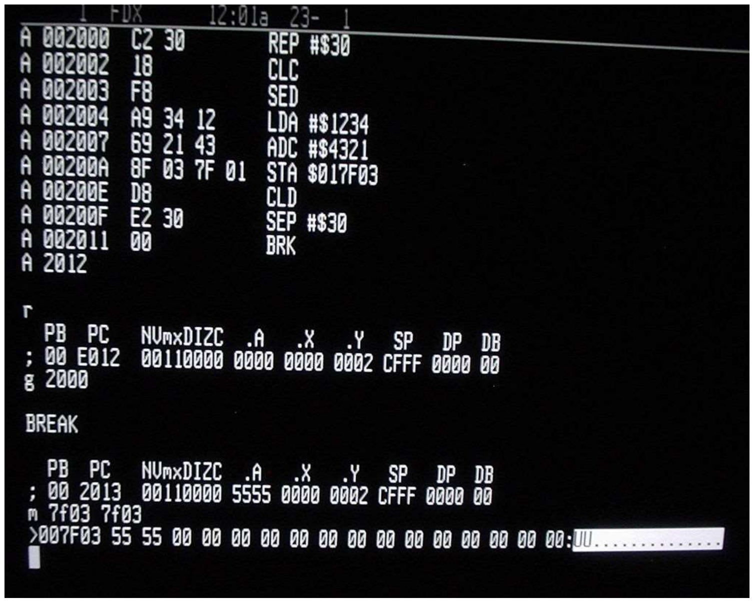


Overview for Today

- Software Power Analysis/Measurement
- Software Power Estimation Models
- Optimizing Software for Low Power Through Compilation Phase
    - Instruction Scheduling
    - Compiler Driven DVS

Software Power Analysis/Measurement

- Different application consume different power on same processors.
  - Why?
- Same application consume different power on different processors.
  - Why?
- Answer lies in machine code and how power-efficiently it is executed.



```
    1  FDX       12:01a  23-  1
A 002000  C2 30        REP #$30
A 002002  18           CLC
A 002003  F8           SED
A 002004  A9 34 12     LDA #$1234
A 002007  69 21 43     ADC #$4321
A 00200A  8F 03 7F 01  STA $017F03
A 00200E  D8           CLD
A 00200F  E2 30        SEP #$30
A 002011  00           BRK
A 2012

r
  PB  PC    NVmxDIZC  .A   .X   .Y   SP   DP   DB
; 00 E012  00110000 0000 0000 0002 CFFF 0000 00
g 2000

BREAK

  PB  PC    NVmxDIZC  .A   .X   .Y   SP   DP   DB
; 00 2013  00110000 5555 0000 0002 CFFF 0000 00
m 7f03 7f03
>007F03 55 55 00 00 00 00 00 00 00 00 00 00 00 00 00 00:UU..............
```

Source: Wikipedia

Very High Level Power Equations for Processor

- $P = I \times V_{cc}$
  - P: Average Power Consumed
  - I: Average Current Consumed
  - V_{cc}: Supply Voltage
- $E = P \times T$
  - T: Execution Time
- $T = N \times \tau$
  - N: Number of Clock Cycles
  - τ: Clock Period (1 ns for processor running at 1 GHz)

Source: [Tiwari 1994]

Energy Model (Base Cost)

- Most processors are pipelined.

1: Instruction Fetch	2: Decode-1	3: Decode-2	4:Execution	5:Register Write-Back

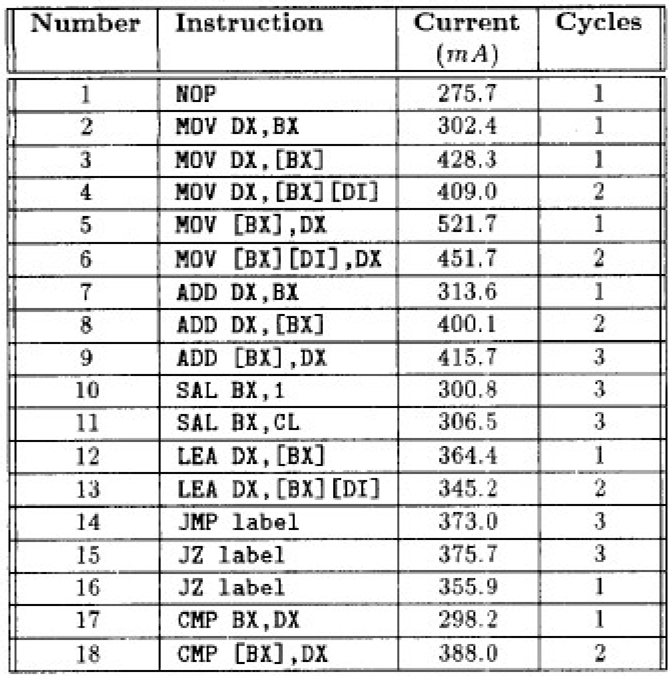
Internal pipelining in the 486DX2

- Energy consumed = f (Instruction Sequence)
- $E_{cycle} = E1_{I1} + E2_{I2} + E3_{I3} + E4_{I4} + E5_{I5}$
  - E_{cycle}: Total energy consumed by processor in a cycle.
  - Ex_{Iy} = Energy consumed by Instrction y in pipeline Stage x.
- $E_{I1} = E1_{I1} + E2_{I1} + E3_{I1} + E4_{I1} + E5_{I1}$
- Average current for execution of instruction I_1 is $E1_{I1}/(V_{cc} \times \tau)$

Source: [Tiwari 1994]

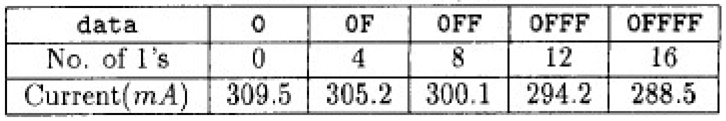
Energy Model (Base Cost) 2

SUBSET OF THE BASE COST TABLE FOR THE 486DX2[1]



Number	Instruction	Current (mA)	Cycles
1	NOP	275.7	1
2	MOV DX,BX	302.4	1
3	MOV DX,[BX]	428.3	1
4	MOV DX,[BX][DI]	409.0	2
5	MOV [BX],DX	521.7	1
6	MOV [BX][DI],DX	451.7	2
7	ADD DX,BX	313.6	1
8	ADD DX,[BX]	400.1	2
9	ADD [BX],DX	415.7	3
10	SAL BX,1	300.8	3
11	SAL BX,CL	306.5	3
12	LEA DX,[BX]	364.4	1
13	LEA DX,[BX][DI]	345.2	2
14	JMP label	373.0	3
15	JZ label	375.7	3
16	JZ label	355.9	1
17	CMP BX,DX	298.2	1
18	CMP [BX],DX	388.0	2

BASE COSTS OF MOV BX, DATA



data	0	0F	0FF	0FFF	0FFFF
No. of 1's	0	4	8	12	16
Current(mA)	309.5	305.2	300.1	294.2	288.5

Source: [Tiwari 1994]

Energy Model (Inter-Instruction Effect)

- Switching activity in a circuit function of present input and previous state. Switching activity is positively correlated with power consumed.
- Base cost is obtained by repeating same instruction over and over.
  - Switching activity is much higher than when sequence of instructions repeated.
- Power consumption of sequence of instruction can differ
  - XOR instruction base cost 319.2 mA
  - ADD instruction base cost 316.4 mA
  - XOR BX, 1; ADD AX, DX
    - Expected cost (Base cost for ADD): 316.4 mA
    - Real cost (Observed): 323.2 mA
    - Difference of 6.8 mA
  - Difference is called **Circuit State Overhead.**
- Modeling through sequence of instructions better than base costs.
  - Longer the chain of sequence, more profiling is needed.
  - Sequence of 2 is probably accurate enough.

Source: [Tiwari 1994]

Energy Model (Stalls and Cache Misses)

- Stalls can cause program execution to take longer.
  - Sequence of 120 *Mov DX, [BX]*
    - should take 120 cycles (1 per instruction instance).
    - actually takes 164 cycles in practices due prefetch buffer stalls.
  - Type of stalls and their reasons, beyond scope of this lecture.
  - Empirically determined penalty cost.
    - Eg: 250 mA for prefetch buffer stalls.
- Cache misses can also prolong program execution.
  - Fixed cost penalty of 216 mA.
  - Cache miss rate can be obtained by profiling the application through hardware counters.

Source: [Tiwari 1994]

Energy Model (Combined)

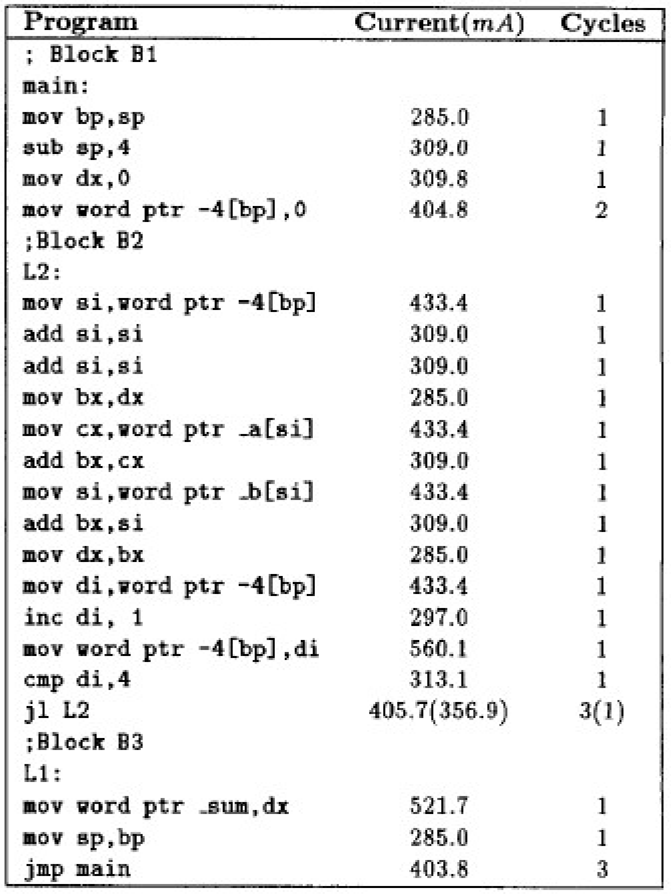
- Program energy cost = Σ_I ($\text{Base}_I \cdot N_I$) + $\Sigma_{I,J}$ ($\text{Ovhd}_{I,J} \cdot N_{I,J}$) + $N_{CM} \cdot \text{Penalty}_{CM}$+ $N_{Stall} \cdot \text{Penalty}_{Stall}$
    - N_I : Number of times Instruction I is executed.
    - Base_I : Base energy cost of I (ignores stalls, cache misses).
    - N_I : Number of times Instruction I,J is executed in sequence.
    - $\text{Ovhd}_{I,J}$: Circuit state overhead when I,J are adjacent.
    - N_{CM} : Number of cache misses.
    - Penalty_{CM} : Cache miss penalty.
    - N_{stall} : Number of stalls.
    - Penalty_{Stall} : Stall penalty.

Source: [Tiwari 1994]

Estimation Example

- 3 Basic Building Blocks.
  - A contiguous section of code with exactly one entry and exit point.
- Building block cost
  - Summation of all ins. cost (current * cycles)
  - B1:1713 mA; B2:4709.8 mA; B3: 2017.9 mA
- Execution: B1 (1x); B2 (4x) and B3 (1x)
- $\text{BaseCost}_{\text{PROGRAM}} = \Sigma\ \text{BaseCost}_{\text{BLOCKi}} * \text{Instances}_{\text{BLOCKi}}$
- Est. Base Current = $\text{BaseCost}_{\text{PROGRAM}}$ / 72 = 369.0mA
  - Program is estimated to execute for 72 cycles.
- Final estimation = 369 + 15 = 384.0 mA
  - Circuit state overhead = 15 mA
- Actual Measured current = 385 mA
  - Very Accurate !!!

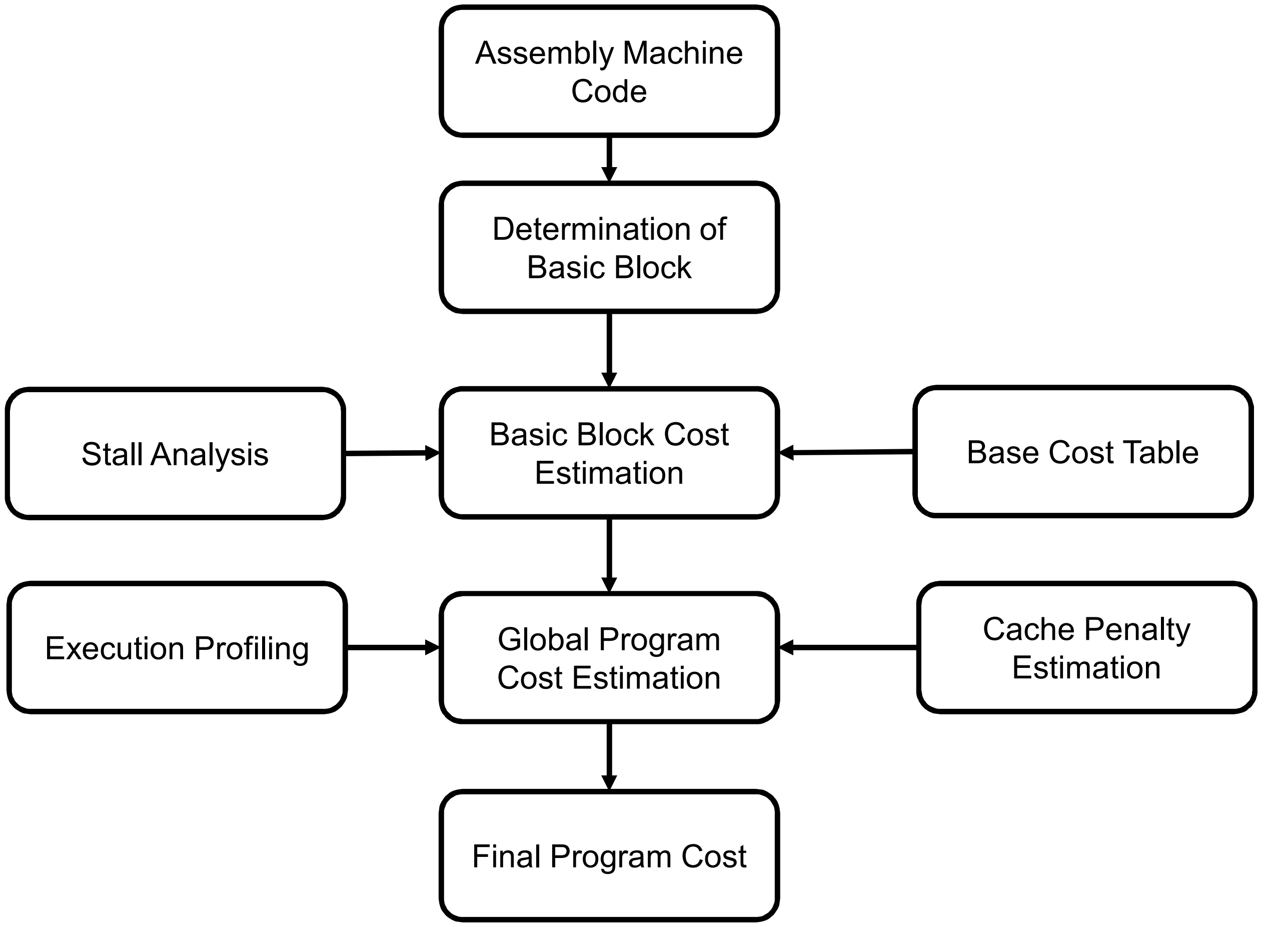
ILLUSTRATION OF THE ESTIMATION PROCESS



Program	Current(mA)	Cycles
; Block B1		
main:		
mov bp,sp	285.0	1
sub sp,4	309.0	1
mov dx,0	309.8	1
mov word ptr -4[bp],0	404.8	2
;Block B2		
L2:		
mov si,word ptr -4[bp]	433.4	1
add si,si	309.0	1
add si,si	309.0	1
mov bx,dx	285.0	1
mov cx,word ptr _a[si]	433.4	1
add bx,cx	309.0	1
mov si,word ptr _b[si]	433.4	1
add bx,si	309.0	1
mov dx,bx	285.0	1
mov di,word ptr -4[bp]	433.4	1
inc di, 1	297.0	1
mov word ptr -4[bp],di	560.1	1
cmp di,4	313.1	1
jl L2	405.7(356.9)	3(1)
;Block B3		
L1:		
mov word ptr _sum,dx	521.7	1
mov sp,bp	285.0	1
jmp main	403.8	3

Source: [Tiwari 1994]

Overall Estimation Flow



Source: [Tiwari 1994]

Software Optimizations

- Example: Heapsort
- Compiler Generated Code vs Hand Generated Code
  - 9% Current Reduction
  - 24% Running Time Reduction
  - 40.6% Energy Reduction
- Register vs Memory Access
  - Power-efficient to use register, if possible, to avoid memory accesses.
  - Hand generated code relies more on registers than memory read/write.

Compiler Generated Code

```
push ebx
push esi
push edi
push ebp
mov ebp,esp
sub esp,24
mov edi,dword ptr 014H[ebp]
mov esi,1
mov ecx,esi
mov esi,edi
sar esi,cl
lea esi,1[esi]
mov dword ptr -20[ebp],esi
mov dword ptr -8[ebp],edi
L3:
mov edi,dword ptr -20[ebp]
cmp edi,1
jle L7
mov edi,dword ptr -20[ebp]
sub edi,1
mov dword ptr -20[ebp],edi
lea edi,[edi*4]
mov esi,dword ptr 018H[ebp]
add edi,esi
mov edi,dword ptr [edi]
mov dword ptr -12[ebp],edi
jmp L8
L7:
mov edi,dword ptr 018H[ebp]
mov esi,dword ptr -8[ebp]
lea esi,[esi*4]
add esi,edi
mov ebx,dword ptr [esi]
mov dword ptr -12[ebp],ebx
mov edi,dword ptr 4[edi]
mov dword ptr [esi],edi
mov edi,dword ptr -8[ebp]
sub edi,1
mov dword ptr -8[ebp],edi
```

Energy Efficient Code

```
push ebp
mov edi,dword ptr 08H[esp]
mov esi,edi
sar esi,1
inc esi
mov ebp,esi
mov ecx,edi
L3:
cmp ebp,1
jle L7
dec ebp
mov esi,dword ptr 0cH[esp]
mov edi,dword ptr[edi*4][esi]
mov ebx,edi
jmp L8
L7:
mov edi,dword ptr 0cH[esp]
mov esi,dword ptr 4[edi]
mov ebx,dword ptr [ecx*4][edi]
mov dword ptr [ecx*4][edi],esi
dec ecx
cmp ecx,1
jne L8
mov dword ptr 4[edi],ebx
jmp L2
```

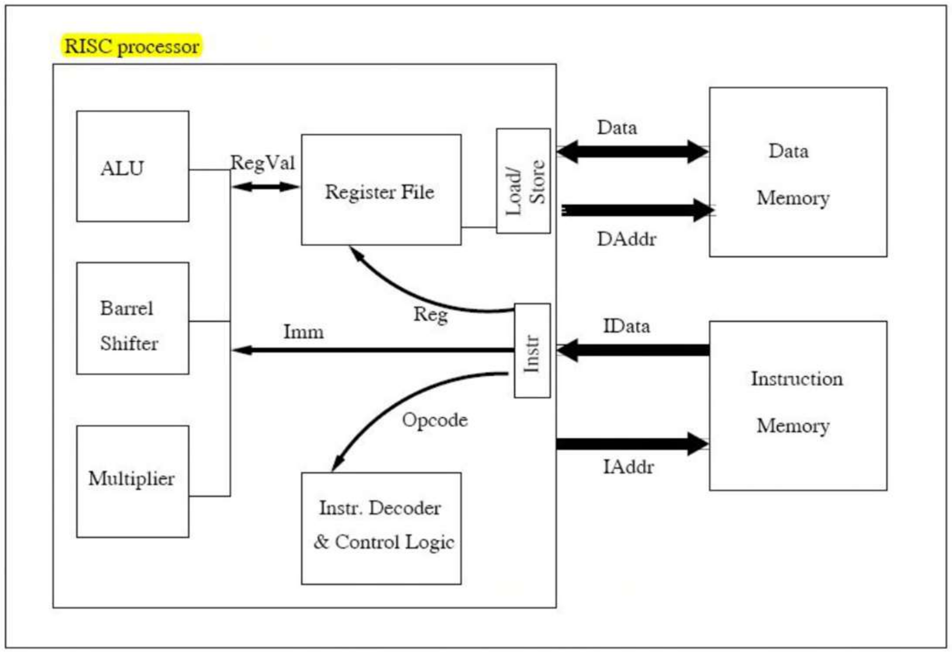
Source: [Tiwari 1994]

Overview for Today

- Software Power Analysis/Measurement
- Software Power Estimation Models
- Optimizing Software for Low Power Through Compilation Phase
    - Instruction Scheduling
    - Compiler Driven DVS

A More Detailed Instruction Level Power Model

- Modelling of new parameters
  - Memory accesses cost (through Memory Bus).
  - Cost of activating and deactivating of functional units.
  - Separation of instruction and data induced costs.



Source: [Steinke 2001]

Energy Model

- $E_{total} = E_{cpu\_instr} + E_{cpu\_data} + E_{mem\_instr} + E_{mem\_data}$
  - E_{total}: Total energy cost of the program
  - E_{cpu_instr}: Instructions dependent costs inside CPU.
  - E_{cpu_data}: Data dependent costs inside CPU.
  - E_{mem_instr}: Instruction dependent cost in the instruction memory.
  - E_{mem_data}: Data dependent cost in the data memory.
- Some more notations:
  - w(x): Number of bits in word x with state “1”.
  - h(x,y): Number of bus lines with different state in x and y (Hamming Distance).
  - BaseCPU(x): Cost caused within CPU due to instruction x.
  - BaseMem(x): Cost caused within memory due to instruction x.
  - FUChange (x,y): Cost of activating and deactivating a function unit if first x and then y is executed.
  - α, β: Linear regression parameters.

Source: [Steinke 2001]

Energy Model (E_{cpu_instr})

- Special notations:
  - *m*: sequence of m instructions; *s*: number of immediate values in the instruction word; *t*: number of registers in the instruction word.
  - *Reg*: Register number; *RegVal*: Value in the register.
  - *IAddr*: Instruction address.

$$
\begin{aligned}
E_{cpu\_instr} = \sum_{i=1}^{m} \Big(& \\
& BaseCPU(Opcode_i) + \\
& \sum_{j=1}^{s}(\alpha_1 * w(Imm_{i,j}) + \beta_1 * h(Imm_{i-1,j}, Imm_{i,j})) + \\
& \sum_{k=1}^{t}(\alpha_2 * w(Reg_{i,k}) + \beta_2 * h(Reg_{i-1,k}, Reg_{i,k})) + \\
& \sum_{k=1}^{t}(\alpha_3 * w(RegVal_{i,k}) + \beta_3 * h(RegVal_{i-1,k}, RegVal_{i,k})) + \\
& \alpha_4 * w(IAddr_i) + \beta_4 * h(IAddr_{i-1}, IAddr_i) + \\
& FUChange(Instr_{i-1}, Instr_i)\Big)
\end{aligned}
$$

- Similar modelling for memory E_{mem_instr} [self-study].

Source: [Steinke 2001]

Energy Model (E_{cpu_data})

- Special notations:
  - *n*: number of data accesses.
  - *DAddr*: Data address.
  - *Data*: Value of data itself.
  - *dir*: Direction of access – read or write.

$$E_{cpu\_data} = \sum_{i=1}^{n} \Big(\alpha_5 * w(DAddr_i) + \beta_5 * h(DAddr_{i-1}, DAddr_i) + \alpha_{6,dir} * w(Data_i) + \beta_{6,dir} * h(Data_{i-1}, Data_i) \Big)$$

- Similar modelling for memory E_{mem_data} [self-study].

Source: [Steinke 2001]

Results

- 1.7% Inaccuracy for *ARM7 TDMI* Processor.

Table 1. *parameters of ARM7TDMI energy model*

parameter	energy (pJ)		parameter	energy (pJ)	
	Read	Write		Read	Write
α_4, α_5	n.a.	48.0	β_4, β_5	n.a.	219.9
α_6	11.0	26.4	β_6	-5.5	224.1
α_7, α_9	n.a.	-19.2	β_7, β_9	n.a.	138.9
α_8	-115.3	n.a.	β_8	57.7	n.a.
α_{10}	-115.3	-60.4	β_{10}	57.7	22.8

Table 2. *overhead for activating or deactivating functional units*

instruction i	instruction $i+1$	overhead for activating/ deactivating (mA)
ALU	Load/Store	2.2
Multiplier	Load/Store	2.5
BarrelShifter	ALU	3.3
Register File	ALU	3.8
Register File	Multiplier	2.1

Source: [Steinke 2001]

Overview for Today

- Software Power Analysis/Measurement
- Software Power Estimation Models
- Optimizing Software for Low Power Through Compilation Phase
    - Instruction Scheduling
    - Compiler Driven DVS

Low-Power Compilers

- Use instruction-level energy costs to guide code generation.
- Minimize memory accesses by utilizing registers effectively.
- Processor-specific optimizations. (Not Covered In Lecture)
    - Dual memory loads, instruction packing.
- Optimize instruction scheduling to reduce activity in specific parts of the system.
    - internal instruction-bus, processor-memory bus, instruction register and register decoder.

Traditional Instruction Scheduling

- Traditional instruction scheduling strategies (increase performance) :
  - avoid pipeline stalls.
  - improve resource (register file, etc.) usage.
  - increase instruction level parallelism (ILP).
- Traditional steps for instruction scheduling
  - Partition program into regions or basic blocks.
  - Build a **Control Dependency Graph** (CDG) and **Data Dependency Graph** (DDG).
  - Schedule instructions within resource constraints.
- Traditional instruction scheduling minimizes Pipelines Stall (PS) derived cost function.

$$PS = \sum D(I_j, I_{j+1}), j = 0, \ldots, n-1$$

  - D (I_j , I_{j+1}): Number of pipeline stalls between instruction I_j and I_{j+1}.

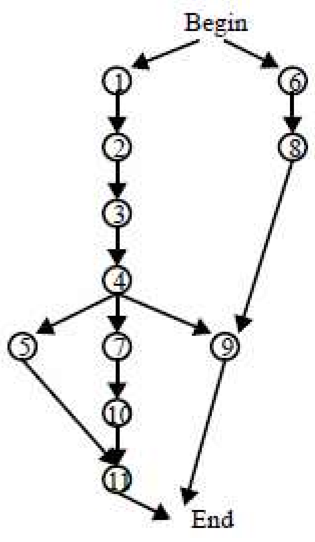
Source: [Despain 1994]

Cold Scheduling Idea

- Cold scheduling minimizes Bit Switches (PS) derived cost function.

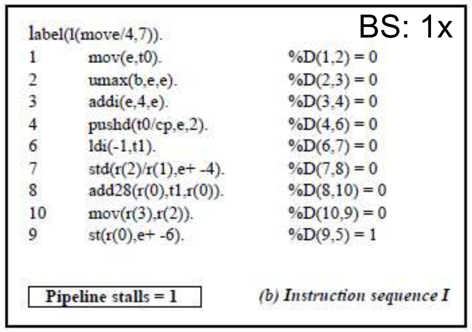
$$BS=\sum S(I_j, I_{j+1}), j=0,\dots,n-1$$

  - S (I_j , I_{j+1}): Number of bit switches between instruction I_j and I_{j+1}.



(a) data dependencies graph

Best Performance

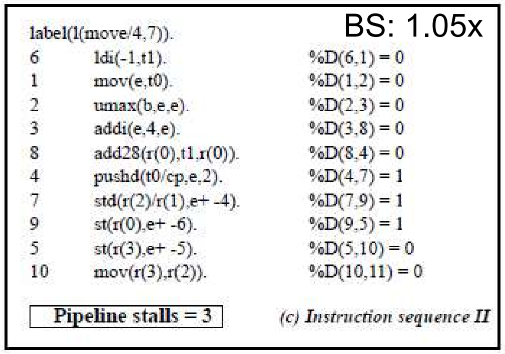


BS: 1x

```
label(l(move/4,7)).
1       mov(e,t0).                 %D(1,2) = 0
2       umax(b,e,e).               %D(2,3) = 0
3       addi(e,4,e).               %D(3,4) = 0
4       pushd(t0/cp,e,2).          %D(4,6) = 0
6       ldi(-1,t1).                %D(6,7) = 0
7       std(r(2)/r(1),e+ -4).      %D(7,8) = 0
8       add28(r(0),t1,r(0)).       %D(8,10) = 0
10      mov(r(3),r(2)).            %D(10,9) = 0
9       st(r(0),e+ -6).            %D(9,5) = 1
```

Pipeline stalls = 1   ***(b) Instruction sequence I***

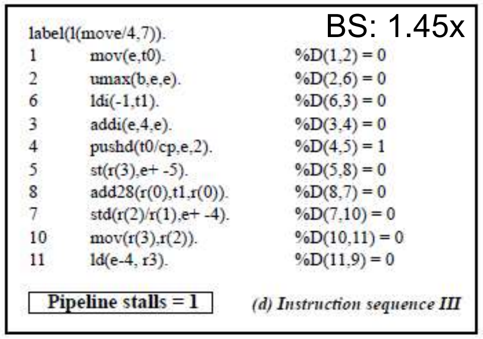
Best Power



BS: 1.05x

```
label(l(move/4,7)).
6       ldi(-1,t1).                %D(6,1) = 0
1       mov(e,t0).                 %D(1,2) = 0
2       umax(b,e,e).               %D(2,3) = 0
3       addi(e,4,e).               %D(3,8) = 0
8       add28(r(0),t1,r(0)).       %D(8,4) = 0
4       pushd(t0/cp,e,2).          %D(4,7) = 1
7       std(r(2)/r(1),e+ -4).      %D(7,9) = 1
9       st(r(0),e+ -6).            %D(9,5) = 1
5       st(r(3),e+ -5).            %D(5,10) = 0
10      mov(r(3),r(2)).            %D(10,11) = 0
```

Pipeline stalls = 3   ***(c) Instruction sequence II***



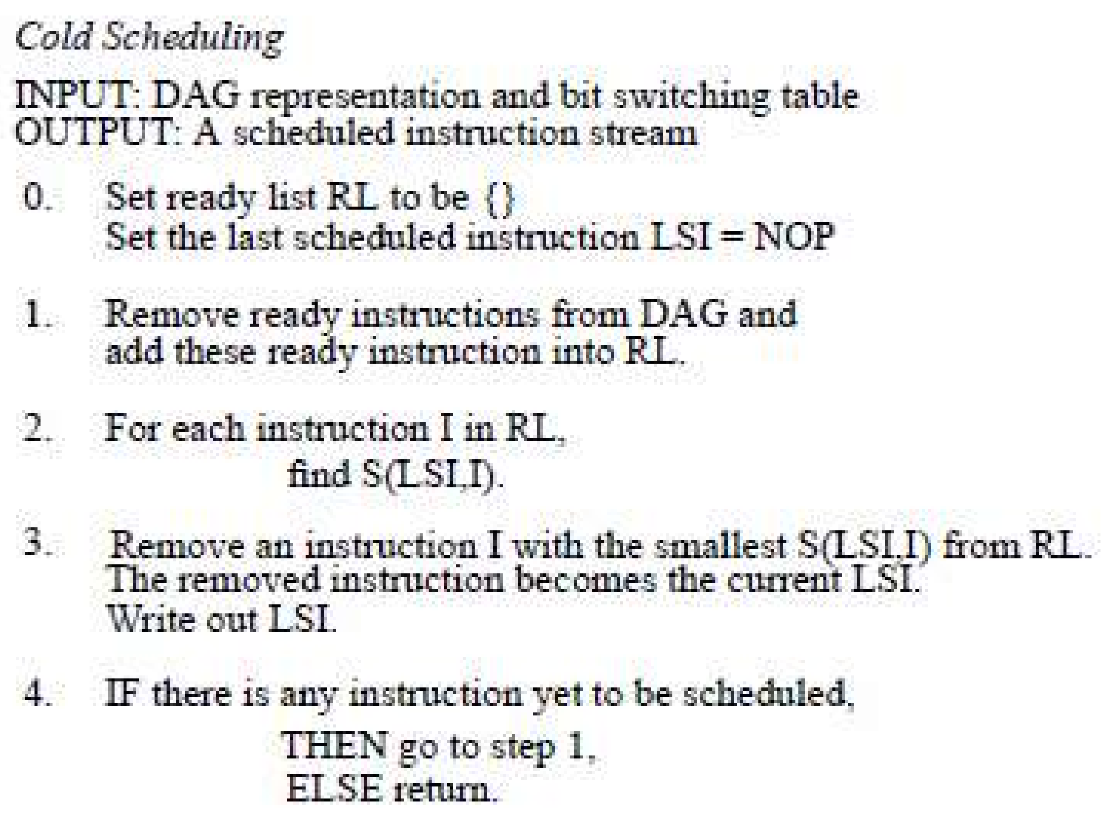
BS: 1.45x

```
label(l(move/4,7)).
1       mov(e,t0).                 %D(1,2) = 0
2       umax(b,e,e).               %D(2,6) = 0
6       ldi(-1,t1).                %D(6,3) = 0
3       addi(e,4,e).               %D(3,4) = 0
4       pushd(t0/cp,e,2).          %D(4,5) = 1
5       st(r(3),e+ -5).            %D(5,8) = 0
8       add28(r(0),t1,r(0)).       %D(8,7) = 0
7       std(r(2)/r(1),e+ -4).      %D(7,10) = 0
10      mov(r(3),r(2)).            %D(10,11) = 0
11      ld(e-4, r3).               %D(11,9) = 0
```

Pipeline stalls = 1   ***(d) Instruction sequence III***

Source: [Despain 1994]

Cold Scheduling Algorithm

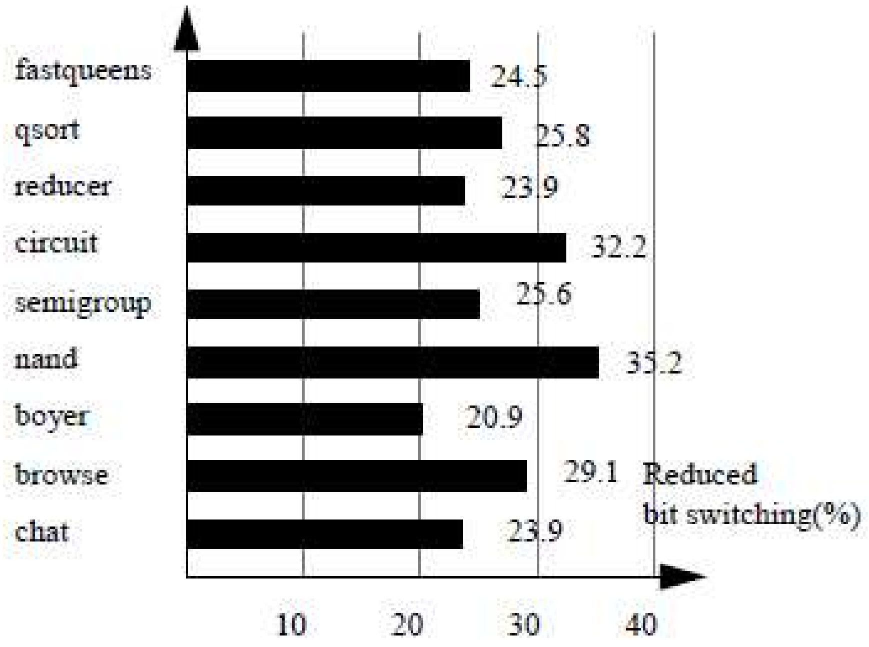


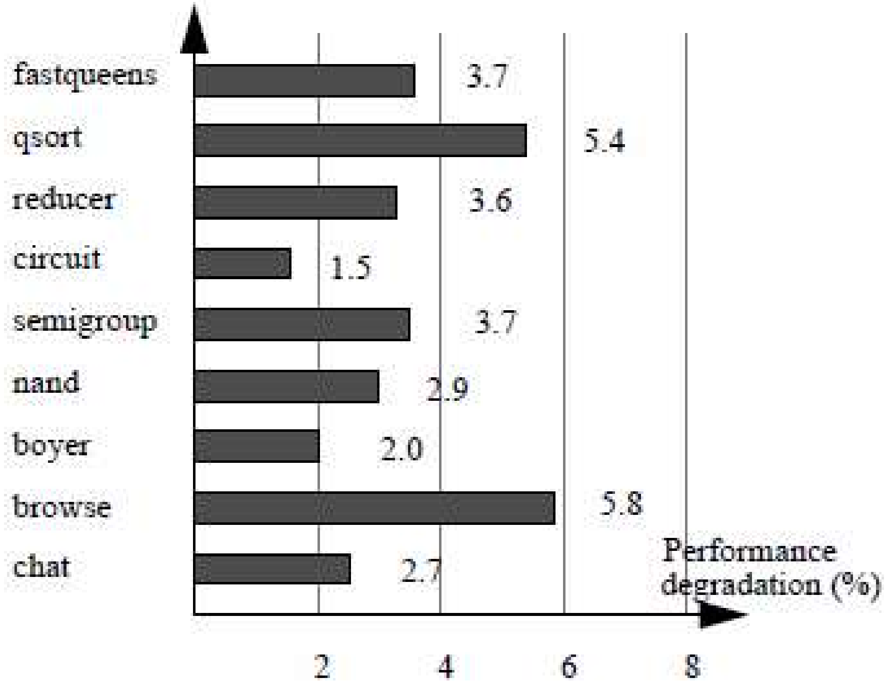
Cold Scheduling

```
INPUT: DAG representation and bit switching table
OUTPUT: A scheduled instruction stream

0.  Set ready list RL to be {}
    Set the last scheduled instruction LSI = NOP

1.  Remove ready instructions from DAG and
    add these ready instruction into RL.

2.  For each instruction I in RL,
            find S(LSI,I).

3.  Remove an instruction I with the smallest S(LSI,I) from RL.
    The removed instruction becomes the current LSI.
    Write out LSI.

4.  IF there is any instruction yet to be scheduled,
            THEN go to step 1,
            ELSE return.
```

Source: [Despain 1994]

Cold Scheduling Results

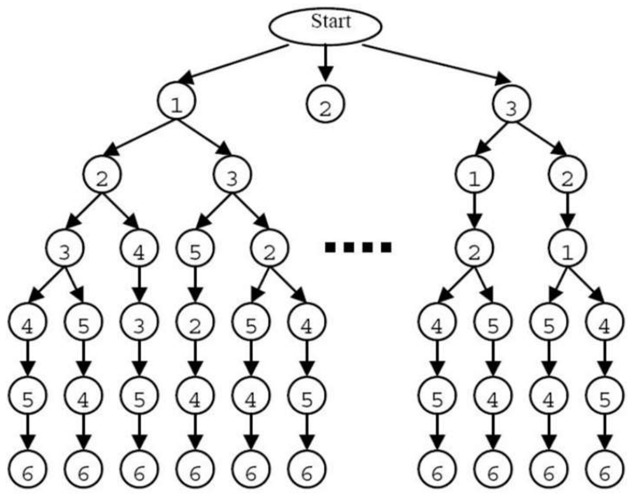


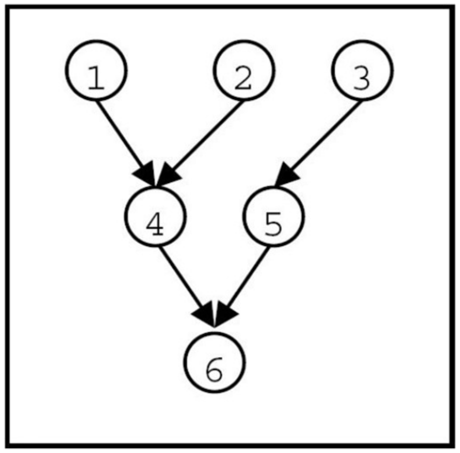


Source: [Despain 1994]

Complexity and Design Space of Ins. Sequencing

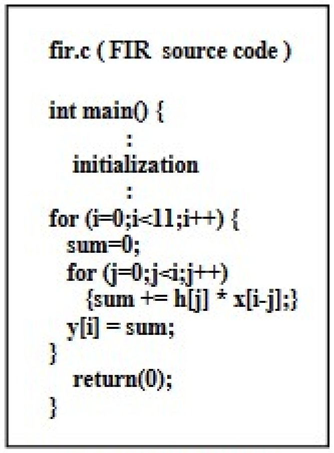
- For n unique instructions, (n-1)!/2 possible sequences.
  - 11 instructions in medium size basic block = 16 million unique combinations.
  - Each combination have a different power consumption.
  - Brute force very difficult.
  - Problem is similar to Travelling Salesman Problem (TSP) : NP-Hard
- In practice, design space smaller.
  - Due to precedence/dependencies not all sequences are valid.





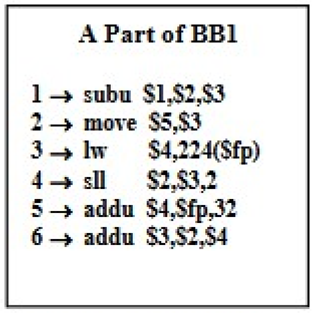
Source: [Choi 2001]

Getting Problem Description



```
fir.c ( FIR  source code )

int main() {
          :
   initialization
          :
for (i=0;i<11;i++) {
  sum=0;
  for (j=0;j<i;j++)
    {sum += h[j] * x[i-j];}
  y[i] = sum;
}
   return(0);
}
```

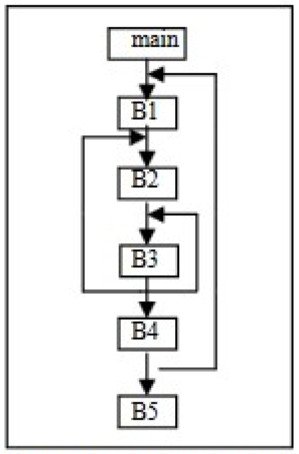
(a) Source Code



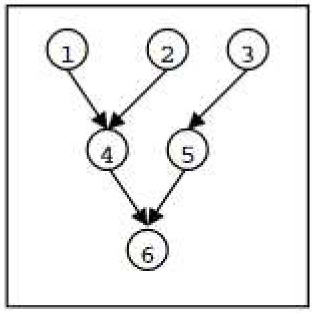
A Part of BB1

```
1 → subu  $1,$2,$3
2 → move  $5,$3
3 → lw    $4,224($fp)
4 → sll   $2,$3,2
5 → addu  $4,$fp,32
6 → addu  $3,$2,$4
```

(b) Assembly Code



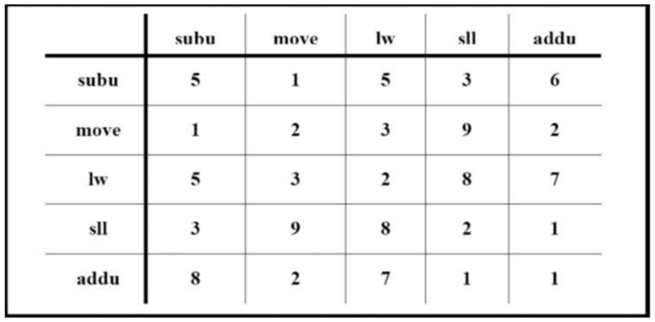
(c) Control Flow Graph



(d) CDG

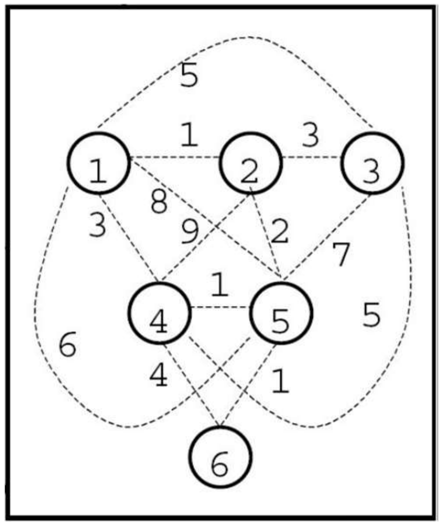
Source: [Choi 2001]

Instruction Scheduling Algorithm

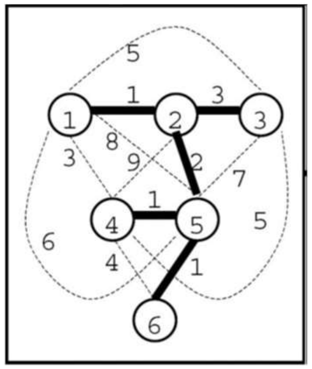


	subu	move	lw	sll	addu
subu	5	1	5	3	6
move	1	2	3	9	2
lw	5	3	2	8	7
sll	3	9	8	2	1
addu	8	2	7	1	1

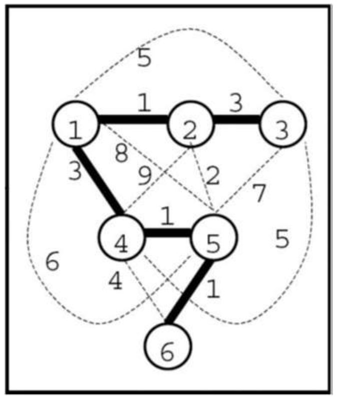
Power Dissipation Table: When an instruction in leftmost column is followed by instruction in top row, then the given power consumption applies.



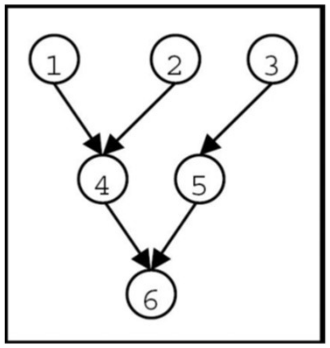
Weighted Strongly Connected Graph (SCG): Contains all edges of CDG plus: edges between any two nodes where precedence is not important (like 1<->2, 1<->3, 2<->3 etc.). This may be one or two edges subject to whether costs are different. Each weight gives power cost for repeated execution of the two connected instructions.



Minimum Spanning Tree (MST): Tree with all edges connected with minimum weight (Prims Algorithm).



Hamiltonian Path: Minimum weight path with each node only traversed once (Simulated Annealing)



Control Dependency Graph: Gives dependency constraints Ex 1: before '4', '2','1' needs to execute; Ex 2: '1', '2', '3' can be executed in any order.

Source: [Choi 2001]

Overview for Today

- Software Power Analysis/Measurement
- Software Power Estimation Models
- Optimizing Software for Low Power Through Compilation Phase
    - Instruction Scheduling
    - Compiler Driven DVS

DVS (Dynamic Voltage Scaling)

- One of the most **effective** method for power savings due to quadratic relationship between power and supply voltage.
- DVS comes at the **cost** of performance degradation. *Idea*: deploy DVS such that it does not incur a penalty.
- An effective DVS strategy:
  - determine intelligently when to adjust the voltage setting (i.e. find the best **‘scaling points’**).
  - Where to adjust to i.e. which voltage setting to choose (i.e. ‘**scaling factors**’).
- Overhead:
  - Switching to and from new voltage setting costs time and energy (=> may reduce or eliminate potential savings).
  - Hundreds of micro-seconds i.e. tens of thousands of instructions (i.e. not even cache misses may be used to perform the transition).
- Considered here **intra-task DVS**: i.e. scaling points may be in the middle of the task execution (in contrast to inter-task DVS).

Source: [Kremer 2003]

Compiler-Directed DVS

- Problem Statement:
  - Given a **program P**, find a **region R** and **a frequency f** such that, if region R is executed at frequency f and the rest of the program P − R is executed at the peak frequency **f**$_{max}$, the total execution time plus **the switching overhead T**$_{trans}$**·2·N(R)** is increased no more than **r percent** of the original execution time **T(P, fmax**), while the total energy usage is minimized. Region R should be at least as large as fraction **ρ** of the entire execution.

$$\min_{R,f} \; P_f \cdot T(R,f) + P_{f_{max}} \cdot T(P-R, f_{max}) + P_{trans} \cdot 2 \cdot N(R)$$

Under constraint

$$T(R,f) + T(P-R, f_{max}) + T_{trans} \cdot 2 \cdot N(R) \leq (1+r) \cdot T(P, f_{max})$$

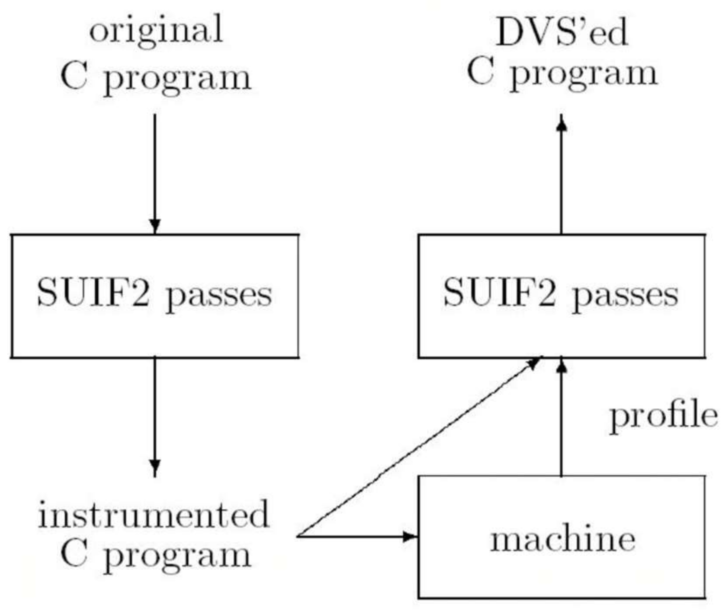
$$T(R, f_{max}) / T(P, f_{max}) \geq \rho$$

  - *T(R, f):* total execution time of region R running at frequency f.
  - *N(R):* Number of times region R is executed.
  - *P*$_f$*:* power consumption of the system at frequency f.
  - *Ttrans, Ptrans:* single switching overhead in terms of performance, power, respectively.

Source: [Kremer 2003]

Compiler-Directed DVS 2

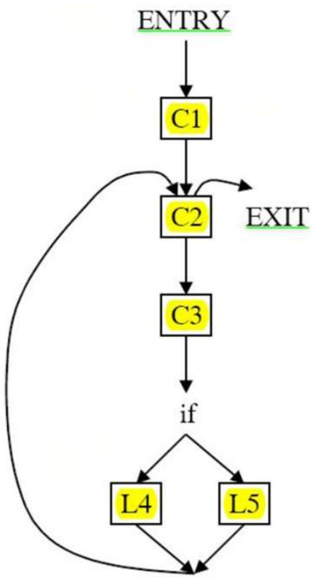
- Steps of compiler-directed DVS:
  1. **Instrumenting**: the input program at selected program locations.
  2. **Profiling**: the instrumented code is executed, filling a subset of entries in tables T(R, f) and N(R).
  3. Rest of table entries are derived (using call graphs etc.); based on **inter-procedural analysis** -> analysis is faster than profiling. [Not covered in Lecture]
  4. The **minimization** problem is solved by enumerating all possible regions and frequencies.
  5. The corresponding DVS system calls are **inserted** at the boundaries of the selected region.



Source: [Kremer 2003]

Compiler-Directed DVS Example

- Assumptions
  - One two CPU frequencies f_{max} and f_{min}.
  - *C*: Call sites; *L*:Loop Sites.
  - First all $N(R_i)$, $T(R_i, f)$ are profiled for the basic regions.
  - **Combined regions**: one entry point, one exit point => all top level statements are executed same # of times.
    - Example for combined regions: if(L4, L5), seq(C2, C3).
    - not allowed: seq(C1, C2), seq(C3, L4), etc.
  - The profile-driven approach gives results that are **not portable** but it captures properties that may not be captured using a compile-time prediction model.

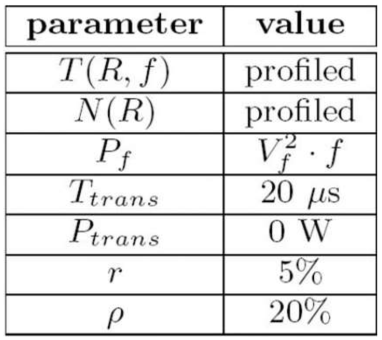


R	$N(R)$	$T(R,f)$	
		f_{max}	f_{min}
C1	1	0	0
C2	10	10	12
C3	10	0	0
L4	8	8	12
L5	2	2	4

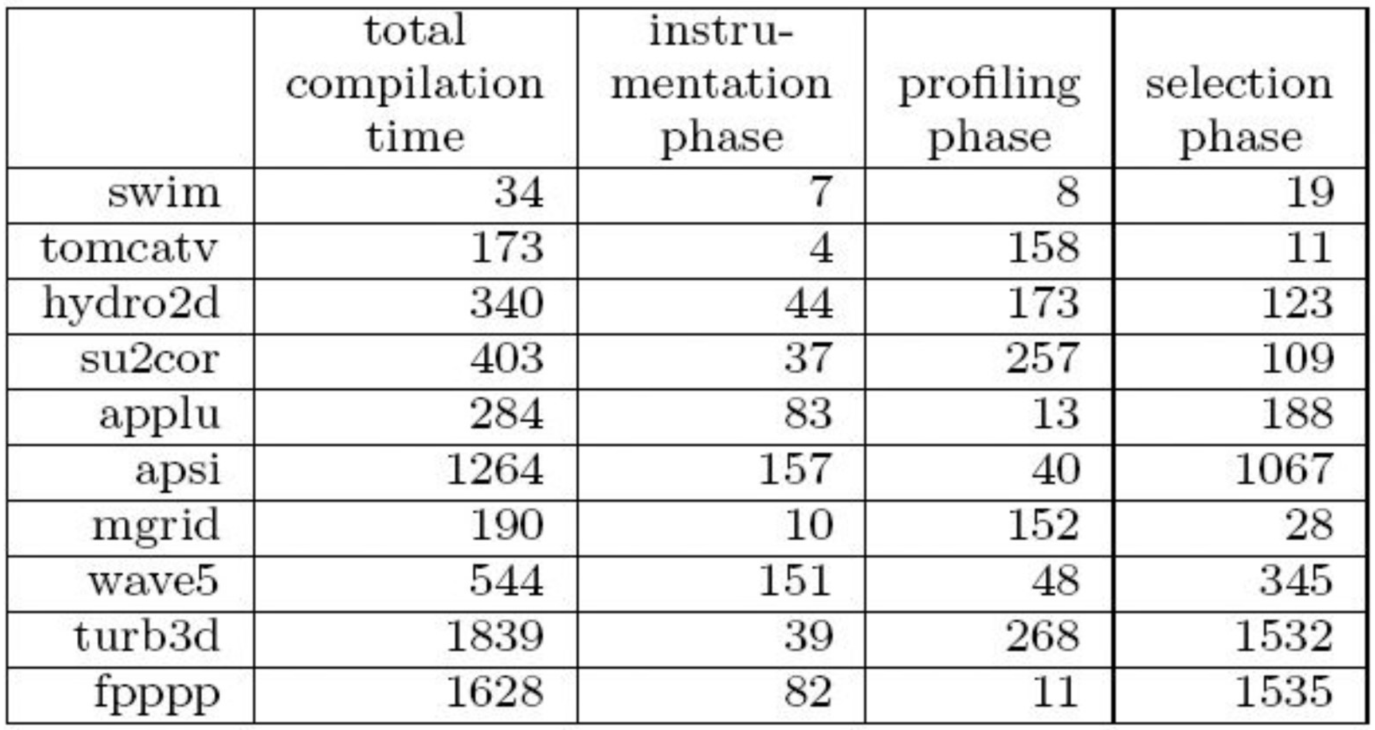
Source: [Kremer 2003]

Results

- Power savings: 0%-28%
- Performance penalty: 0%-4.7%



parameter	value
$T(R, f)$	profiled
$N(R)$	profiled
P_f	$V_f^2 \cdot f$
T_{trans}	20 μs
P_{trans}	0 W
r	5%
ρ	20%



	total compilation time	instru-mentation phase	profiling phase	selection phase
swim	34	7	8	19
tomcatv	173	4	158	11
hydro2d	340	44	173	123
su2cor	403	37	257	109
applu	284	83	13	188
apsi	1264	157	40	1067
mgrid	190	10	152	28
wave5	544	151	48	345
turb3d	1839	39	268	1532
fpppp	1628	82	11	1535

Source: [Kremer 2003]

Conclusion

- Software power estimation is possible and necessary.
  - It represents a high level of abstraction and therefore it is faster than estimating power consumption of the underlying hardware circuitry.
- Compiler may include optimization for low power
  - Instruction scheduling.
  - Intra-procedural DVS.
- Optimizing for low power and high performance are two distinct tasks (most of the times)!

Source

- Homework >> Tiwari, V., Malik, S., & Wolfe, A. (1994). Power analysis of embedded software: a first step towards software power minimization. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2(4), 437-445.
- Steinke, S., Knauer, M., Wehmeyer, L., & Marwedel, P. (2001, September). An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proc. of PATMOS*.
- Hsu, C. H., & Kremer, U. (2003, June). The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In ACM SIGPLAN Notices (Vol. 38, No. 5, pp. 38-48). ACM.
- Su, C. L., Tsui, C. Y., & Despain, A. M. (1994, February). Low power architecture design and compilation techniques for high-performance processors. In *Compcon Spring'94, Digest of Papers.* (pp. 489-498). IEEE.
- Choi, K. W., & Chatterjee, A. (2001, September). Efficient instruction-level optimization methodology for low-power embedded systems. In *Proceedings of the 14th international symposium on Systems synthesis* (pp. 147-152). ACM.