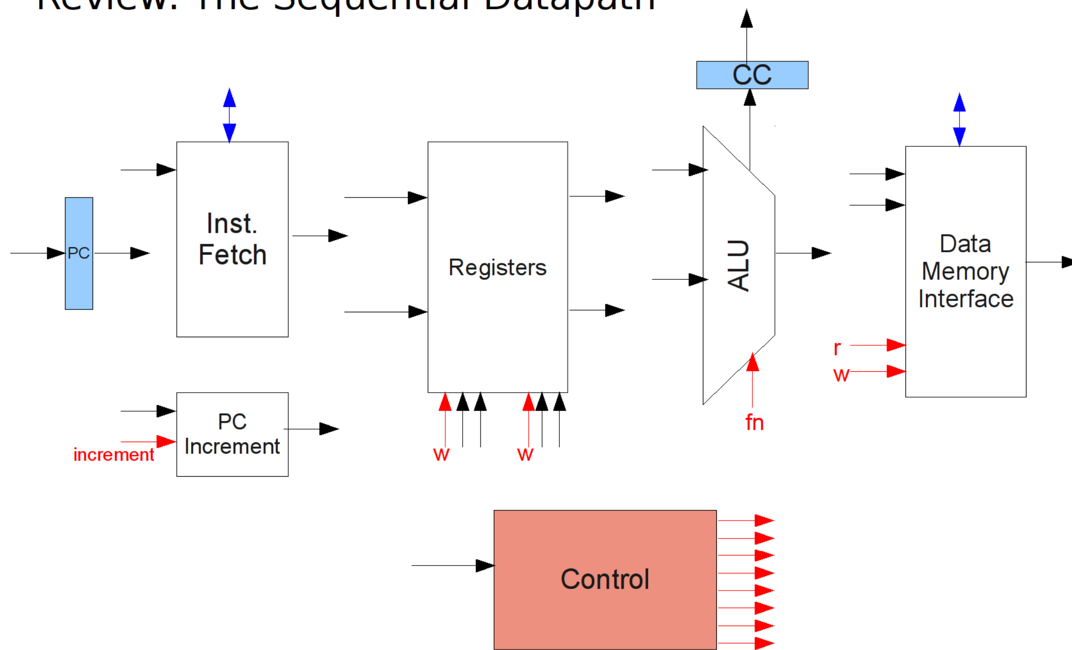
The Hardware/Software Interface

CSE351 Winter 2011

Module 8: Going Faster – Pipelining and Parallel Execution

Review: The Sequential Datapath



How Can We Go Faster? (Part 1)

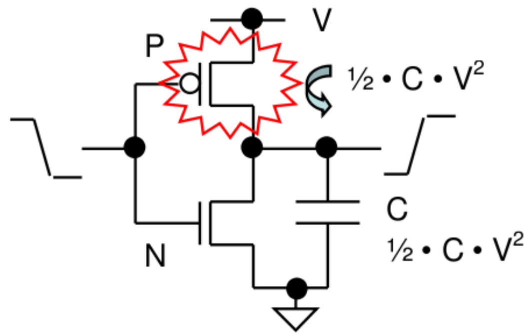
- **Increase the clock rate**
  - Moore's Law (1965)
    - Number of transistors will double about every 2 years
    - Smaller transistors → faster switching times
    - More transistors → more complicated functionality
  - Wirth's Law
    - "Software is getting slower more rapidly than hardware becomes faster."
      - Office 2007 slower on a 2007 machine than Office 2000 on a 2000 machine.
- **Put multiple copies of the datapath on a single chip**
  - "Multi-core processors"
  - Each core runs a different program
    - A single program may not run faster, but if you have a lot of them to run you get them all done faster
    - So why go to multi-core?

Moore's Law

Fusion: Triple product $nT\tau$ doubles every 1.8 years
Accelerators: Energy doubles every 3 years
Moore s Law: Transistor number doubles every 2 years
Performance
Year
1000
100
10
1
0.1
0.01
0.001
1970
1980
1990
2000
T3, ST, TFR, PLT, Alcator A, PDX, JET, DIII, Alcator C, TFTR, JET, JT60U, JT60U, JT60U, JT60U, JT60U, JT60U, JT60U, JET, TFTR, DIIID
4004, 8080, 8086, 80286, 80386, 80486, Pentium, Pentium II, Pentium III, Pentium 4
SppS, Tevatron, LHC
JG03.35-17c

http://iter.rma.ac.be/en/community/Worldwide/index.php

Dynamic Power Dissipation



CMOS Inverter

Dynamic power ~ $C\ V^2\ f$

Process	Product	Frequency	Performance	Power (watts)	Voltage (volts)
130 nm	Pentium 4 (Northwood)	3.4 GHz	1342 SpecInt2K	89.0	1.525
130 nm	Pentium M (Banias)	1.0 GHz	673 SpecInt2K	7.0	1.004 ULV
90 nm	Pentium 4 (Prescott)	3.6 GHz	1734 SpecInt2K	103	1.47
90 nm	Pentium M (Dothan)	2.0 GHz	1429 Specint2K	21	1.32
65 nm	Pentium 4 (Cedarmill)	3.6 GHz	1764 SpecInt2K	86	1.33
65 nm	Core Duo (Yonah)	2.167 GHz	1721 SpecInt2K	31	1.3

Table 2: Performance and Power of Intel Microprocessors, 130 nm to 65 nm

Product	Normalized Performance	Normalized Power	EPI on 65 nm at 1.33 volts (nJ)
i486	1.0	1.0	10
Pentium	2.0	2.7	14
Pentium Pro	3.6	9	24
Pentium 4 (Willamette)	6.0	23	38
Pentium 4 (Cedarmill)	7.9	38	48
Pentium M (Dothan)	5.4	7	15
Core Duo (Yonah)	7.7	8	11

Table 3: EPI of Intel Microprocessors

http://www.intel.com/pressroom/kits/core2duo/pdf/epi-trends-final2.pdf

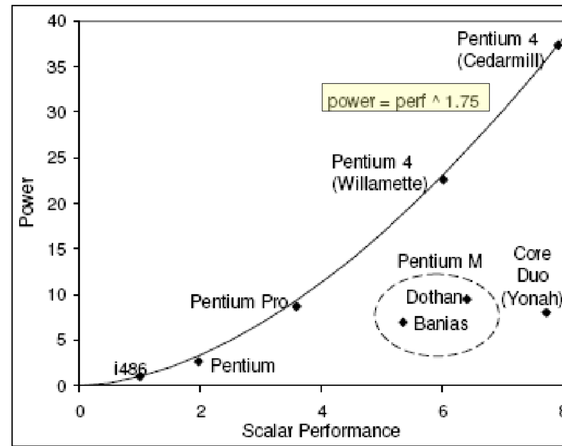


Figure 2: Normalized Power versus Normalized Scalar Performance for Multiple Generations of Intel Microprocessors

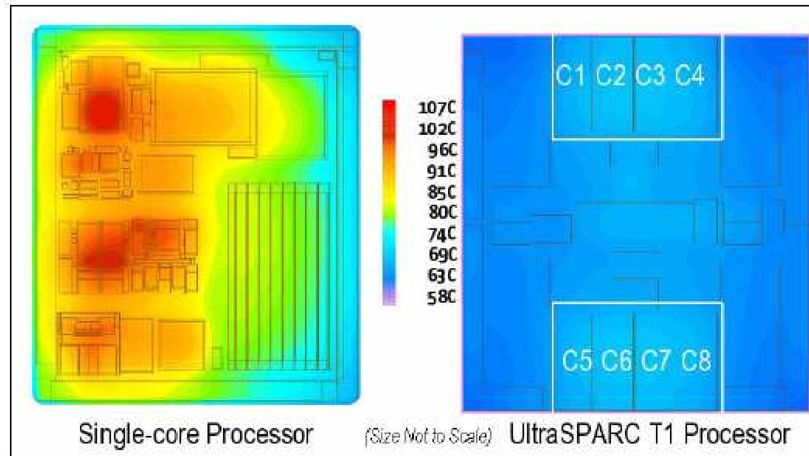


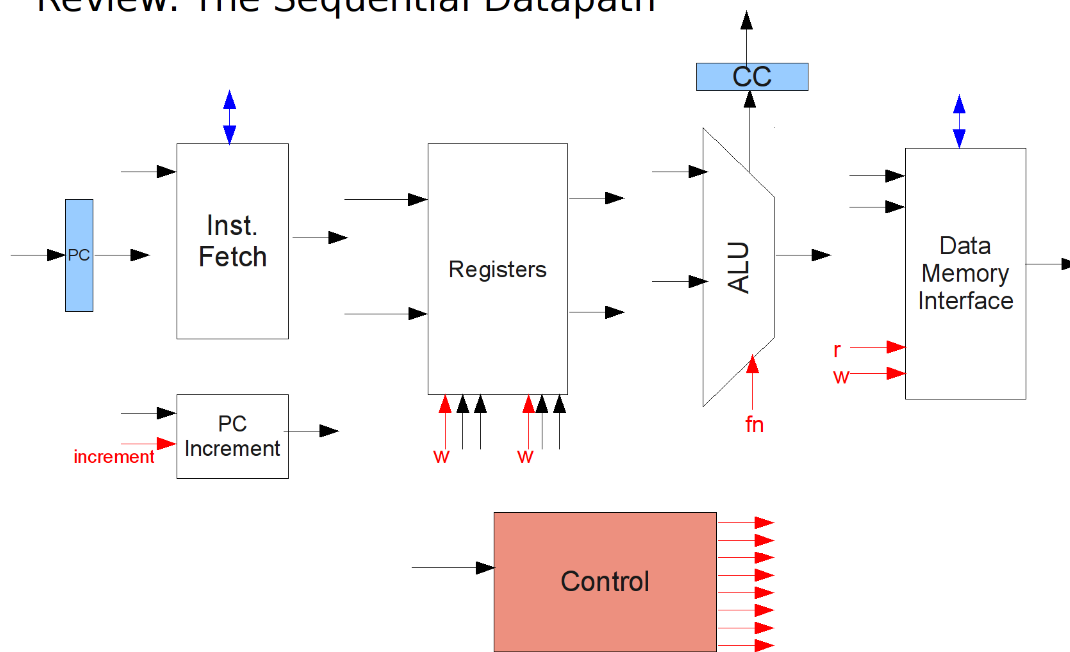
FIGURE 2 Power Density: Single-Core Processor vs. UltraSPARC T1 Processor

http://www.sun.com/processors/whitepapers/UST1_pwrsav_v1.0.pdf

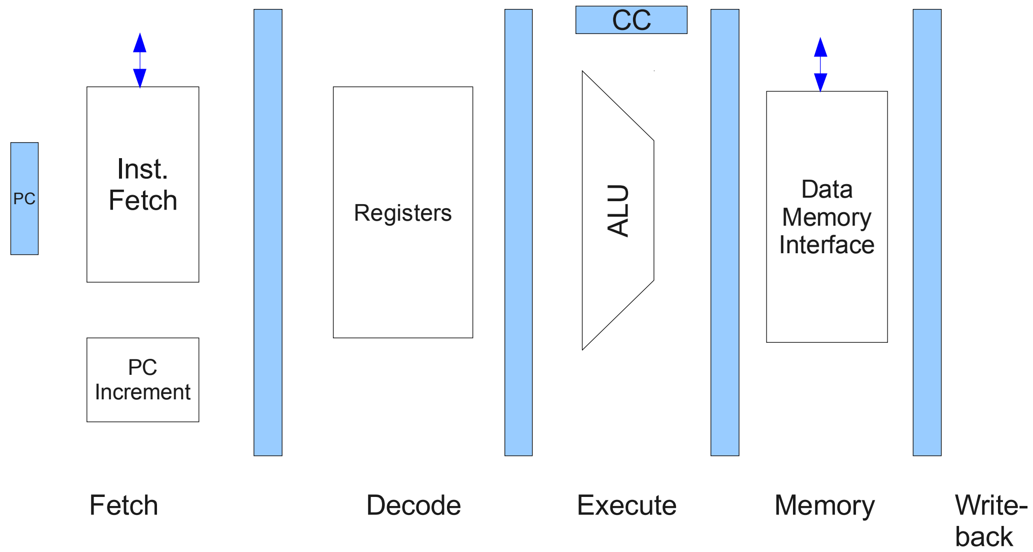
How Can We Go Faster? (Part 2)

- **Execute more than one instruction at a time**
  - Think about a single datapath / core
  - Keep the clock frequency / voltage the same, but execute (say) 5 instructions at a time
    - The instruction completion rate with be (up to) 5 times higher than if we execute only one at a time
    - How to do this? Pipelining...

Review: The Sequential Datapath

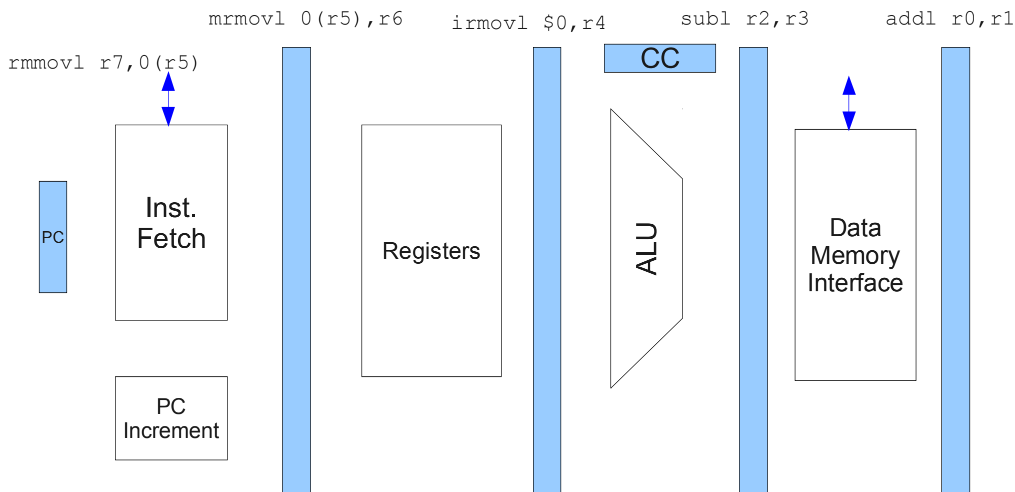


The Pipelined Datapath



Instructions in flight

```
addl    r0, r1
subl    r2, r3
irmovl  $0, r4
mrmovl  0(r5), r6
rmmovl  r7, 0(r5)
```



Notes About Pipelining

- **One way to think about this is that the single cycle datapath leaves most resources "unused" most of the time**
- **To think about performance, think about the rate at which instructions complete**
  - The time between fetching any individual instruction and completing it is no less than it was in the single cycle datapath
- **The big question: Does the pipelined datapath get the right results?**
  - A generalization: When will simultaneous execution of instructions get the same result as their sequential execution?

Parallel Execution

This is okay

```
addl  r0, r1
addl  r0, r2
addl  r3, r4
```

→ | addl r0, r1 | addl r0, r2 | addl r3, r4 |

This isn't okay

```
addl  r0, r1
addl  r1, r2
addl  r3, r4
```

→ | addl r0, r1 | addl r1, r2 | addl r3, r4 |

Dependences restrict opportunities for parallel execution

Dependences

- **Read-after-write (RAW)**
  - addl r4, r5
    addl r5, r6
    - Also known as a "flow dependence"
    - Also known as a "true dependence"
- **Write-after-read (WAR)**
  - addl r4, r5
    addl r3, r4
    - Also known as an "anti-dependence"
- **Write-after-write (WAW)**
  - irmovl $0, r5
    irmovl $8, r5
    - Also known as an "output dependence"
- **WAR and WAW are "false dependences"**
  - The dependences have to do with names, not values
  - They can be eliminated by "re-writing the code"

Dependences and Pipelining

- **WAW (write-after-write)**
  - Not a problem because all register writes happen (only) in the write stage
    - Instructions flow "in order" through the write stage
- **WAR (write-after-read)**
  - Not a problem because:
    - register reads happen in decode stage, which precedes the write stage
    - instructions flow "in order" through those stages
      - so an earlier read has definitely happened before a later write can possibly occur

Dependences and Pipelining

- **RAW (read-after-write)**
  - Problem...
    - `addl r2, r3`
      `subl r3, r4`
    - The `addl` will write register 3 when it reaches stage 5
    - The `subl` will read register 3 when it reaches stage 2
      - When it reaches stage 2, the `addl` is only at stage 3, and hasn't yet written its result!
    - This particular instance of RAW can be fixed using forwarding

Forwarding

Forwarding Doesn't Always Work

- ```
  mrmovl   0(r3), r4
  addl     r4, r5
  ```
- **The value in memory (to be written to r4) isn't available until too late**
  - It's fetched during the same cycle that the addl needs to use the ALU to do the add
- **In cases like this we have to stall the pipeline**
  - The unresolvable dependence is recognized during the decode stage
  - The fetch and decode stages are stalled (frozen)
  - A NOP instruction (a "bubble") is inserted into the pipeline behind the mrmovl
  - Separating the two instructions by a NOP now allows forwarding to work
    - The addl fetches its operand from the pipe register that follows the Memory stage

Another Problem: Jump instructions

- **Every instruction implicitly reads the PC**
- **Jumps (aka "branches") write the PC**
- **There is therefore a RAW dependence between a jump and the instruction(s) that follow**
- **For conditional jumps, it may not be known until late in the pipeline whether the jump should take place or not**
  - E.g., after the Execute stage
- **What should the pipeline do while waiting to resolve the conditional branch?**
  ```

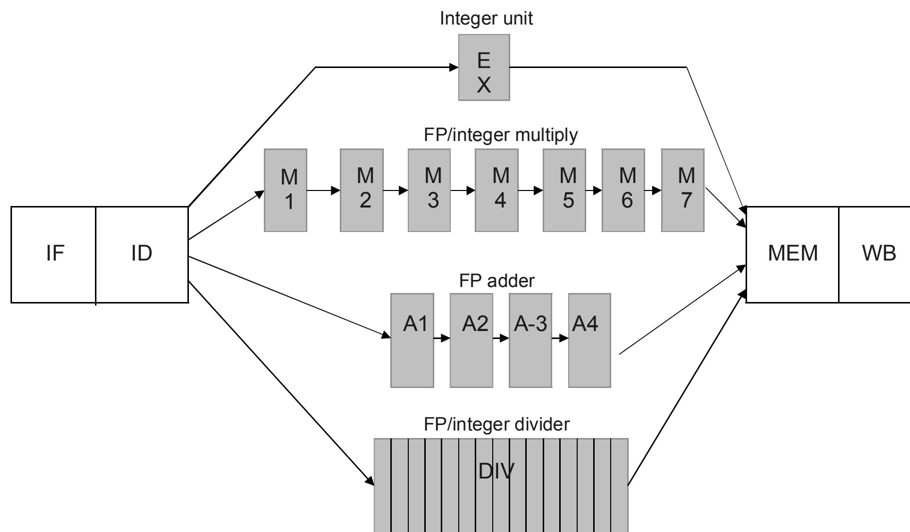
Branch Prediction

- **Branch prediction is the general notion of guessing what the next PC should be (after a jump has been fetched)**
  - Option 1: Assume jump will not be taken
    - fetch instructions sequentially, like always
    - if the jump ends up being taken, convert the pipe registers behind the branch to represent NOPs, rather than the instructions that have been fetched
      - Note that this is okay so long as those instructions haven't yet written any registers or memory...
        - Why is this okay?
- **Mis-predictions waste cycles**
  - The mis-prediction penalty increases with the depth of the pipeline

Branch Prediction (cont.)

- **Option 2: Maintain a branch prediction table (in hardware)**
- **For example (the simplest thing possible)**
  - Keep a hash table that maps the current value of the PC to the next instruction that should be fetched (the next PC)
  - When a branch is taken, insert the PC of the branch instruction and the target address of the branch into the table
    - E.g., the branch at PC = 0x00041804 transfers control to 0x00042208
  - For unconditional branches, you'll miss predict the first time you encounter the branch, but predict correctly after that
  - For conditionals, well... You can imagine missing the first time for a (taken) branch at the bottom of a loop, then then hitting until the loop terminates
- **There are MANY branch prediction schemes, of many levels of complexity**
- **(You can start to see where all the transistors Moore's Law has given us have been going...)**

Beyond Simple Pipelining



Beyond Pipelines

- We have two basic choices:
  - Only one instruction may be in EX stage, no matter how long it takes it to get through there, or...
  - Let's cram instructions into EX as fast as we can
- Which should we do?
  - Reminder: We're trying to go fast...
- Putting multiple functional units in parallel is both a problem and an opportunity
- The Opportunity:
  - *Hey, this is great! Why don't I just stuff a bunch of ALUs, some memory interfaces, some float units, etc. in there?*
    - More hardware → higher performance?
  - *In fact, why don't I issue more than one instruction per cycle?!!!*
    - "multi-issue" → NOT part of today's material, but not far from it

Multiple Functional Units

- In order execution leads to under-utilization of hardware

- Parallel execution → out of order execution / completion
  - Time per stage is not a constant
    - » Structural hazards are possible
      - FP divide takes many cycles, and is not pipelined
      - May need to write more than one register in a cycle

- Out of order execution
  - RAW dependences may be longer

Eliminating False Dependences

- **False (WAR and WAW) dependences restrict the progress of successive instructions**
  - It can be difficult to find enough dispatchable instructions to keep the functional units busy

- **False dependences are dependences on names, not values**
- **They can be eliminated by by "using more names"**

Renaming: Using Java as an Example

```
String name = getName( id0 );
String printStr = id0 + ": " + name;
name = getName( id1 );
printStr = id1 + ": " + name;
```

```
String name = getName( id0 );

String  printStr = id0 + ": " + name;

name = getName( id1 );

printStr = id1 + ": " + name;
```


Rewritten Code

```
String name0 = getName( id0 );
String printStr0 = id0 + ": " + name0;
String name1 = getName( id1 );
String printStr1 = id1 + ": " + name1;
```

```
String name0 = getName( id0 );            String name1 = getName( id1 );

String printStr0 = id0 + ": " + name0;   String printStr1 = id1 + ": " + name1;
```

- Can't get rid of flow dependences by renaming
- Renaming costs memory
- Loops tend to produce false dependences
  - "Loop unrolling" does what that sounds like
  - Unrolled loops can benefit from renaming

Renaming in the Processor: Register Renaming

- **The ISA says there are 8 registers**
  - Programs are forced to reuse them
- **The hardware includes, say, 64 registers**
- **As instructions are fetched, the hardware detects false dependences and rewrites the dependent instruction to use some otherwise unused register**
  - ```
    addl   r3, r4   →   addl   r3, r4
    irmovl $0, r3       irmovl $0, r17
    addl   r5, r3       addl   r5, r17
    addl   r3, r6       addl   r17,r6
    ```
- **This requires a somewhat complicated record keeping scheme in the hardware**
  - More transistors used...

Final Observations

- **The ISA is a logical specification**
- **There are direct implementations, but...**
- **There's no reason the implementation has to correspond directly to the simple logical view provided by the ISA**
  - Any implementation that gets results equal to what the ISA promises is correct
- **“Multi-core” is visible to programs**
  - Moreover, programs have to be (re)written to take advantage of them
- **Clever implementations of a fixed ISA are not visible to programs**
  - Old programs still work
  - Programs compiled a while ago probably run faster on newer implementations, but...
  - Compilers may be aware of some aspects of implementation, and adjust the code they generate accordingly
    - E.g., the compiler may understand the branch prediction algorithm, and try to generate code that tends to minimize the mis-prediction rate
    ```