ESE532: System-on-a-Chip Architecture

Day 22: November 15, 2017

Real Time

Today

Real Time
- Demands
- Challenges
  - Algorithms
  - Architecture
- Approaches

Message

- Real-Time applications demand different discipline from best-effort tasks
- Look more like synchronous circuits
- Can sequentialize, like processor
  - But must avoid/rethink typical general-purpose processor common-case optimizations

Real-Time Tasks

- What applications demand real-time computing tasks?

Real-Time Tasks

- Human consumed media:
  - video, audio, games, UI, graphics
- Control
  - Anti-lock brakes, cruise-control, auto-pilot, UAV, self-driving car, industrial automation
- Stock trading
- Network traffic handling
- Crypto (avoid information leak)

Real-Time Guarantees

- What guarantees might we want for real-time tasks?

Real-Time Guarantees

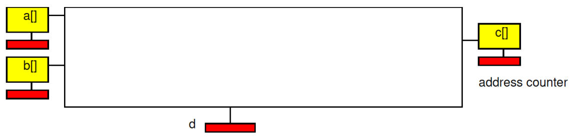
- Attention/processing within fixed interval
  - Sample new value every XX ms
  - Produce new frame every 30 ms
  - Both: schedule to act and complete action
- Bounded response time
  - Respond to keypress within 20 ms
  - Detect object within 100 ms
  - Return search results within 200 ms

Synchronous Circuit Model

- A simple synchronous circuit is a good "model" for real-time task
  - Run at fixed clock rate
  - Take input every cycle
  - Produce output every cycle
  - Complete computation between input and output
  - Designed to run at fixed-frequency
    - Critical path meets frequency requirement

Preclass 1

- How implement spatial pipeline?



```
float a[MAX], b[MAX], c[MAX], d;
// stuff to load/define a, b, d ... maybe an outer loop
for (i=0;i<MAX;i++) c[i]=(a[i]+b[i])*d;
```

Historically

- Real-Time concerns grew up in EE
  - Because an analog circuit was the only way could meet frequency demands
  - …later a dedicated digital circuit…
- Where worried about
  - Signal processing, video, control, …

Technological Change

- Why not be satisfied with this answer today?
  - For real-time task need dedicated synchronous circuit?

Performance Scaling

- As circuit speeds increased
  - Can meet real-time performance demands with heavy sequentialization
- Circuit and processor clocks
  - from MHz to GHz
- Many real-time task rates unchanged
  - 44KHz audio, 33 frames/second video
- Even 100MHz processor
  - Can implement audio in a small fraction of its computational throughput capacity

HW/SW Co-Design

- Computer Engineers – know can implement anything as hardware or software
- Want freedom to move between hardware and software to meet requirements
  - Performance, costs, energy

Real-Time Challenge

- Meet real-time demands / guarantees
  - Economically using programmable architectures
- Sequentialize and share resources with deterministic, guaranteed timing

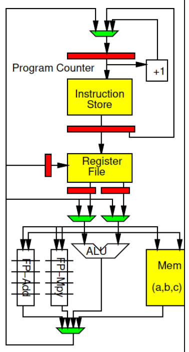
Preclass 2

- Time for loop iteration case (a)?

```
top: bzero r4, exit
        ld f1, r1 //  f1<-*r1 (read a[i])
        ld f2, r2 //  f2<-*r2 (read b[i])
        fpadd f3,f1,f2 // f3<-f1+f2 (floating-point)
        fpmul f5,f3,f4 // f5<-f3*f4 (floating-point)
        st f5, r3 // *r3<-f5 (write c[i])
        subi r4,#1,r4 // r4<-r4-1 (decrement i)
        addi r1,#4,r1 // r1<-r1+4 (update a ptr)
        addi r2,#4,r2 // r2<-r2+4 (update b ptr)
        addi r3,#4,r3  // r3<-r1+4 (update c ptr)
        b top
exit:
```

Preclass 2 Processor

- With data hazard stalls, bypassing



Preclass 2

- Time for loop iteration case (a)?

```
top: bzero r4, exit
        ld f1, r1 //  f1<-*r1 (read a[i])
        ld f2, r2 //  f2<-*r2 (read b[i])
        fpadd f3,f1,f2 // f3<-f1+f2 (floating-point)
        fpmul f5,f3,f4 // f5<-f3*f4 (floating-point)
        st f5, r3 // *r3<-f5 (write c[i])
        subi r4,#1,r4 // r4<-r4-1 (decrement i)
        addi r1,#4,r1 // r1<-r1+4 (update a ptr)
        addi r2,#4,r2 // r2<-r2+4 (update b ptr)
        addi r3,#4,r3  // r3<-r1+4 (update c ptr)
        b top
exit:
```

Preclass 2

- Time for loop iteration case (b)?

```
top: bzero r4, exit
        ld f1, r1 //  f1<-*r1 (read a[i])
        ld f2, r2 //  f2<-*r2 (read b[i])
        fpadd f6,f1,f2 // f6<-f1+f2 (floating-point)  ** different **
        fpmul f5,f3,f4 // f5<-f3*f4 (floating-point)
        subi r4,#1,r4 // r4<-r4-1 (decrement i)
        addi r1,#4,r1 // r1<-r1+4 (update a ptr)
        addi r2,#4,r2 // r2<-r2+4 (update b ptr)
        addi r3,#4,r3  // r3<-r1+4 (update c ptr)
        mv f3, f6 //f3 <- f6                          ** new **
        st f5, r3 // *r3<-f5 (write c[i])             ** different place **
        b top
exit:
```

Data-dependent hazard

- Stalls instruction pipeline
  - Only when data needed before computed

Observe

- Instructions on "General Purpose" processors take variable number of cycles

Preclass 3

- How many cycles?

```
sum=0;
for (i=0;i<32;i++) {
    sum+=(0-(b%2)) & a;
    b=b>>1;
    a=a<<1;
    }
```

Preclass 3

- How many cycles?

```
sum=0;
for (;b!=0;b=b>>1)  {
     if (b%2==1)
        sum+=a;
     a=a<<1;
}
```

Observe

- Data-dependent branching, looping
  - Means variable time for operations

Two Challenges

1. Architecture – Hardware have variable (data-dependent) delay
   - Esp. for General-Purpose processors
     - Instructions take different number of cycles
2. Algorithm – computational specification have variable (data-dependent) operations
   - Different number of instructions

$$Time = \sum_i Cycles(i)$$

Algorithm

- What programming constructs are data-dependent (variable delay)?

Programming Constructs

- Conditionals: if/then/else
- Loops without compile-time determined bounds
  - While with termination expressions
  - For with data-dependent bounds
- Recursion
- Hash tables with variable-sized buckets
- Memoization
- Interrupts
  - I/O events, time-slice

Programming Constructs

- Dynamic Dataflow
  - Variable rates
  - Switch/select operators

…like Hardware

- Many problematic constructs similar to C/Programming-Language constructs need to avoid for hardware
  - Dynamic allocation (malloc)
  - Recursive functions
  - Loops without determined bounds
  - Mux-conversion/predications for if/then/else

Architecture

- What processor constructs are variable delay?

Processor Variable Delay

- Data hazards
- Caches
- Data-dependent branching / branch delays
- Speculative issue
  - Out-of-Order, branch prediction
- Dynamic arbitration for shared resources
  - Bus, I/O, Crossbar output, memory, …

Cache Predictable?

- Is an element in or out of cache?
  - Accessed before?
  - Had an address conflict?
  - Depend on access pattern
- If shared
  - Did someone else write it?
  - Depends on everything else sharing

Hardware Architecture

- Some “optimizations” can cause variable delay even in dedicated hardware data path
  - Caches
  - Common-case optimizations
  - Pipeline stalls

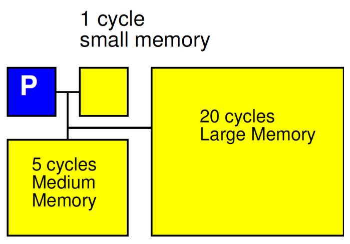
What can we do to make architecture more deterministic?

- Explicitly managed memory
- Fixed-delay pipelines
  - Scheduled
  - Multi-threaded
- Deadlines
- Offline-scheduled resource sharing

Explicitly Managed Memory

- Make memory hierarchy visible
  - Use Scratchpad memories instead of caches
- Explicitly move data between memories
  - E.g. DMA into OCM, movement into local memory
- Already do for Register File in Processor
  - Load/store between memory and RF slot
  - …but don’t do for memory hierarchy

Explicitly Managed Memory

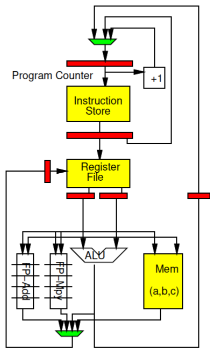


Fixed Delays (1)

- Drop dynamic data hazards, branch speculation
- Data becomes available after a predictable time
- Branches take effect at a fixed time
  - Likely delayed
- Schedule to delays to get correct data

Fixed Delay Example

- Branches occur
  - 1 cycle later (uncond)
  - 3 cycles later
- Non-FP data
  - Available on 2nd instr
- FP data
  - Available on 6th instr



Preclass 4a

- Where code not work?

```
top: bzero r4, exit
        ld f1, r1 //  f1<-*r1 (read a[i])
        ld f2, r2 //  f2<-*r2 (read b[i])
        fpadd f6,f1,f2 // f6<-f1+f2 (floating-point)
        fpmul f5,f3,f4 // f5<-f3*f4 (floating-point)
        subi r4,#1,r4 // r4<-r4-1 (decrement i)
        addi r1,#4,r1 // r1<-r1+4 (update a ptr)
        addi r2,#4,r2 // r2<-r2+4 (update b ptr)
        addi r3,#4,r3  // r3<-r1+4 (update c ptr)
        mv f3, f6 //f3 <- f6
        st f5, r3 // *r3<-f5 (write c[i])
        b top
exit:
```

Preclass 4a

- Where code not work?

```
top: bzero r4, exit
        ld f1, r1 //  f1<-*r1 (read a[i])
        ld f2, r2 //  f2<-*r2 (read b[i])
        fpadd f6,f1,f2 // f6<-f1+f2 (floating-point)
        fpmul f5,f3,f4 // f5<-f3*f4 (floating-point)
        subi r4,#1,r4 // r4<-r4-1 (decrement i)
        addi r1,#4,r1 // r1<-r1+4 (update a ptr)
        addi r2,#4,r2 // r2<-r2+4 (update b ptr)
        addi r3,#4,r3  // r3<-r1+4 (update c ptr)
        mv f3, f6 //f3 <- f6
        st f5, r3 // *r3<-f5 (write c[i])
        b top
exit:
```

Preclass 4b

- How fix?

```
top: bzero r4, exit
        ld f1, r1 //  f1<-*r1 (read a[i])
        ld f2, r2 //  f2<-*r2 (read b[i])
        fpadd f6,f1,f2 // f6<-f1+f2 (floating-point)
        fpmul f5,f3,f4 // f5<-f3*f4 (floating-point)
        subi r4,#1,r4 // r4<-r4-1 (decrement i)
        addi r1,#4,r1 // r1<-r1+4 (update a ptr)
        addi r2,#4,r2 // r2<-r2+4 (update b ptr)
        addi r3,#4,r3  // r3<-r1+4 (update c ptr)
        mv f3, f6 //f3 <- f6
        st f5, r3 // *r3<-f5 (write c[i])
        b top
exit:
```

Preclass 4b: Quick Fix

```
top: bzero r4, exit
       ld f1, r1 //  f1<-*r1 (read a[i])
       ld f2, r2 //  f2<-*r2 (read b[i])
       noop // let f2 load
       fpadd f6,f1,f2 // f6<-f1+f2 (floating-point)
       fpmul f5,f3,f4 // f5<-f3*f4 (floating-point)
       subi r4,#1,r4 // r4<-r4-1 (decrement i)
       addi r1,#4,r1 // r1<-r1+4 (update a ptr)
       addi r2,#4,r2 // r2<-r2+4 (update b ptr)
       addi r3,#4,r3  // r3<-r1+4 (update c ptr)
       mv f3, f6 //f3 <- f6
       b top
       st f5, r3 // *r3<-f5 (write c[i])        ** accommodate branch delay
exit:
```

Preclass 4b: Avoid noop

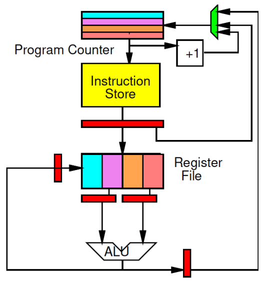
```
start: bzero r4, exit
top:   ld f1, r1 //  f1<-*r1 (read a[i])
       ld f2, r2 //  f2<-*r2 (read b[i])
       st f5, r3 // *r3<-f5 (write c[i])        ** move this back to open slot
       fpadd f6,f1,f2 // f6<-f1+f2 (floating-point)
       fpmul f5,f3,f4 // f5<-f3*f4 (floating-point)
       subi r4,#1,r4 // r4<-r4-1 (decrement i)
       addi r1,#4,r1 // r1<-r1+4 (update a ptr)
       addi r2,#4,r2 // r2<-r2+4 (update b ptr)
       addi r3,#4,r3  // r3<-r1+4 (update c ptr)
       mv f3, f6 //f3 <- f6
       b top
       bzero r4, exit                           ** in branch delay slot
exit:
```

Fixed-Delay (2)

- Drop dynamic data hazards, branch speculation
- Pipeline processor
- But only feed one instruction per thread through processor at time
  - Each instruction completes before next issues (no dependencies)
- Use pipeline to issue from multiple threads
  - For throughput, not latency

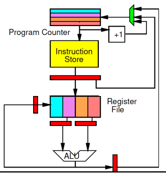
Multithreaded Pipeline

- Only one instruction per thread in pipeline
- C-slow (Day 7)
  - looks like PIPEDEPTH slower processors
- No interlock/bypass
  - Smaller control
  - Faster cycle?



Multithreaded Pipeline

- Can run multiple threads
- Non-real-time threads can share
- Timing of threads not impact each other
- Non-real-time threads take variable time
  - Not interfere with real-time thread slots



Deadline Instruction

- Set a hardware counter for thread
- Demand counter reach 0 before allowed to continue
- Orderly way to tolerate variable instructions in algorithm
- Model: fixed rate of attention
  - Stall if get there early
  - Similar to flip-flop on a logic path
    - Wait for clock edge to change value
- Model: fixed-time

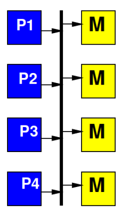
Offline Schedule Resource Sharing

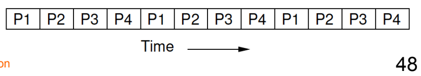
- Don't arbitrate
- Decide up-front when each shared resource can be used by each thread or processor
  - Simple fixed schedule
  - Detailed Schedule
- What
  - Memory bank, bus, I/O, network link, …

Time-Multiplexed Bus

Fixed by hardware master

- 4 masters share a bus
- Each master gets to make a request on the bus every 4th cycle
  - If doesn't use it, goes idle



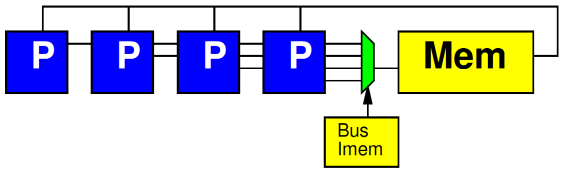


Time-Multiplexed Bus

- Regular schedule
- Fixed bus slot schedule of length N > masters
  - (probably a multiple)
- Assign owner for each slot
  - Can assign more slots to one
- E.g. N=8, for 4 masters
  - Schedule (1 2 1 3 1 2 1 4)

Fully Scheduled

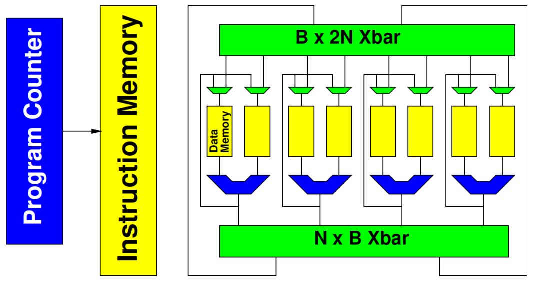
- At extreme, fully schedule which tasks gets resource on each cycle



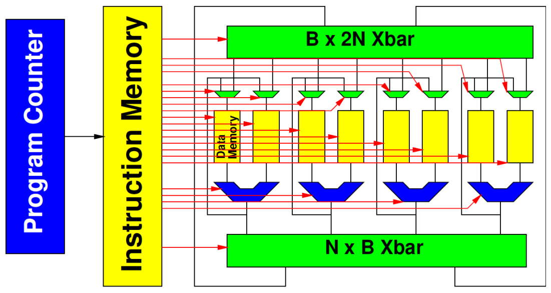
Fully Scheduled

- At extreme, fully schedule which tasks gets resource on each cycle
- Sensible if all master's sharing resource are also fully scheduled, running in lock-step
- Think of instruction field for bus

Fully Scheduled (before instr)



Fully Scheduled



SoC Opportunity

- Can choose which resources are shared
- Can dedicate resources to tasks
- Isolate real-time tasks/portions of tasks from best-effort
  - Separate hardware/processors
  - Separate memories, network

Different Goals

Real-Time
- Willing to recompile to new hardware
- Want time on hardware predictable
- Willing to schedule for delays in particular hardware

General Purpose/Best Effort
- ISA fixed
- Want to run same assembly on different implementations
- Tolerate different delays for different hardware
- Run faster on newer, larger implementations

WCET

- WCET – Worst-Case Execution Time
- Analysis when working with algorithms and architectures with data-dependent delay
  - Need to meet real time
  - Calculate the worst-case runtime of a task
    - Like calculating the critical path (but harder)
    - Worst-case delay of instructions
    - Worst-case path through code
    - Worst-case # loop iterations

Big Ideas:

- Real-Time applications demand different discipline from best-effort tasks
- Look more like synchronous circuits and hardware discipline
- Can sequentialize, like processor
  - But must avoid/rethink typical processor common-case optimizations
  - Offline calculate static schedule for computation and sharing
    - Instead of dynamic arbitration, interlocks

Admin

- Function+Energy milestone due Friday
- P4 (area, 1Gb/s) milestone
  - Out now
  - Due Friday 12/1
  - (nothing due the Friday of Thanksgiving)
- Next week
  - Meet on Monday, but not Wednesday
    - Because Wednesday is a logical Friday…