cse30 discussion 9 - final review

Ibrahim Awwal

July 27, 2015

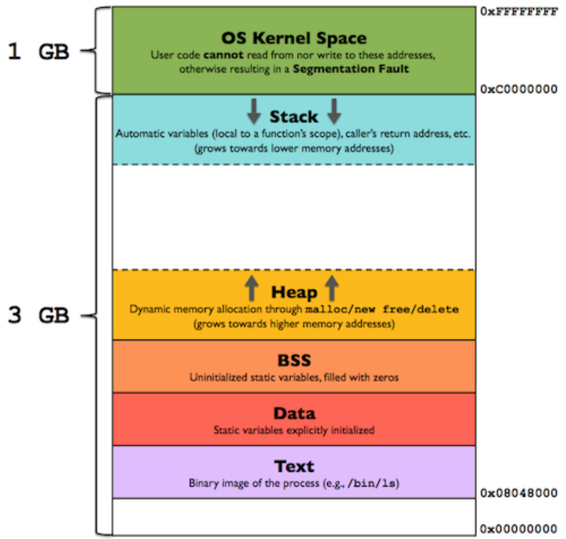
pointers and memory

pointer arithmetic

```
int arr[4] = {256, -1, 3, 0xAABBCCDD}
int *ptr = arr;
*(ptr+1);
*(((char *) ptr) + 1);
```

- `ptr+1` increments by 1*sizeof(datatype)
- Casting a pointer changes the increment
- Subtracting pointers within same array makes sense, adding does not - why?

heap/stack



1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

memory alignment

- Processors read multiple bytes from memory at once
- Variable address must be aligned on a multiple of the size
- Struct fields should be ordered to minimize padding
- Array of Structs vs Struct of Arrays

structs in assembly

- Compute offsets into struct to read individual fields
- Be aware of alignment/padding when calculating offsets

```
struct foo{
    char c;
    short s;
    int x;
}

@ r0 is address of struct
ldrb r1, [r0, #0]   @ c
ldrh r2, [r0, #2]   @ s
ldr r3, [r0, #4]    @ x
```

multi dimensional vs multi level arrays

- Multi dimensional: `int X[3][4];`
- Multi level:
  - `int *X[3]; X[0] = (int *) malloc(4*sizeof(int));`
- Instructions needed to access an element?
- When would we use each type?

endianness

- Little Endian: LSB stored first
- Big Endian: MSB stored first
- x86, ARM are both Little Endian (though ARM can be configured Big Endian)
- Network byte order is Big Endian

Example: int x = 0x12345678; &x=0x8000

Address	0x8000	0x8001	0x8002	0x8003
Little	0x78	0x56	0x34	0x12
Big	0x12	0x34	0x56	0x78

assembly programming

bit manipulations

- Clearing bits: BIC, Shifts, AND
- Setting bits: OR
- Masking

conditional execution

```
CMP r0, #0
BEQ null
LDR r0, [r0]
null: MOV r0, #-1
```

function call conventions

1. Save temporary registers onto the stack (`push`)
2. Put arguments in r0-r3 (additional arguments on stack)
3. Branch and link to target address (`bl <funcname>`)
4. Callee puts return value in r0-r1
5. Callee branches back to link register (`bx lr`)

- Caller vs Callee save registers?
- ARM Architecture Procedure Call Standard (AAPCS)

nested and recursive function calls

- Set up a function call as usual
- `bl <funcname>`
- After function returns, which instruction is executed next?

```
int some_func(int arg){
...
return some_func(arg-1);
}
```

- What does this recursive call correspond to?
- What about `return some_func(arg-1) + some_func(arg-2);`

stack

- Basic operations: `push`, `pop`
- `sp` points to top of stack (last entry pushed)
- Stack grows downward
- Each function call has its own “stack frame”

miscellaneous gotchas

- Not saving required registers onto the stack
- Using volatile registers between function calls
- Incorrect recursive calls (bl without returning)
- Conditional branches not accounting for fall through
- Not indexing by data type size
- Forgetting to increment loop counter

pipelining

pipeline intro

- Idea: Split execution of an instruction into multiple stages
- Multiple instructions can then execute in parallel, also can shorten time for each stage
- Increases Instructions Per Cycle (IPC) or Instruction Level Parallelism (ILP)
- Less logic per stage => shorter clock period needed
- What is the drawback?
- What is the limitation of this technique?

5 stage pipeline

1. Instruction fetch
2. Instruction Decode/Register read
3. Execution
4. Data Memory
5. Register Write

hazards

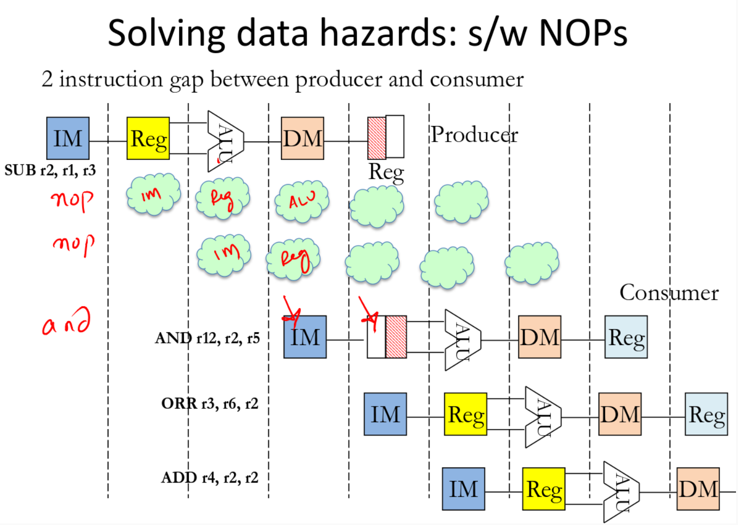
- Not every set of instructions can be parallelized
- If one instruction uses result of previous instruction as operand, we need to wait for this result (data hazard)
- Register write happens in first half of cycle (stage 5)
- Register read happens in second half of cycle (stage 2)
- If we have a conditional branch, we can't continue pipelining instructions until we know whether the branch is taken (control hazard)

examples of data hazards

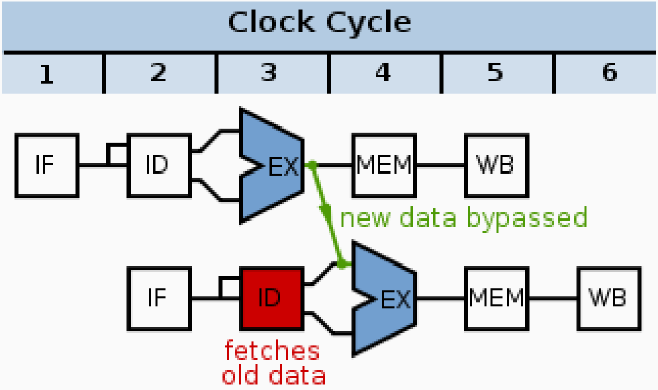
```
ADD r2, r0, r1
MUL r2, r2, r3

LDR  r4, [r0]
AND r5, r4, r1
```

handling data hazards

- Insert NOPs during compilation
- Stall pipeline (in hardware) and wait for result
- Reorder instructions (before assembling) to eliminate stalls
- Forward result from previous stages

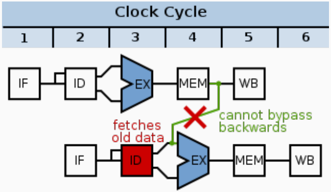

Solving data hazards: s/w NOPs
2 instruction gap between producer and consumer
IM
Reg
ALU
DM
Reg
Producer
SUB r2, r1, r3
nop
im
Reg
ALU
nop
im
Reg
and
Consumer
AND r12, r2, r5
IM
ALU
DM
Reg
ORR r3, r6, r2
IM
Reg
ALU
DM
Reg
ADD r4, r2, r2
IM
Reg
ALU
DM

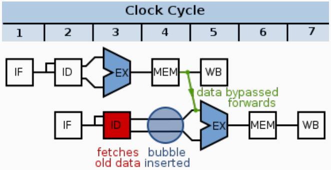
example: forwarding



examples: data memory

```
LDR  r4, [r0]
AND r5, r4, r1
```





bonus architecture techniques

- Caches: Main memory can take 100s of cycles to access, so put commonly accessed data closer to CPU
- Branch Prediction: Predict whether a branch will be taken, where it will go
- Out of Order Execution: Reorder instructions to eliminate pipeline stalls
- Superscalar: Use multiple functional units to issue and process multiple instructions at once

practice final questions