Royal Holloway University of London

Lecture 14 – Support for Procedures

Karl R. Wilcox
Karl@cs.rhul.ac.uk

Objectives

- **In this lecture we will cover**
  - **Review of recent lectures**
  - **Procedure / sub-routine calls**
  - **Hardware support (or lack thereof)**

(Diagrams from Patterson & Hennessy)

Royal Holloway University of London

Review of Recent Lectures

- **We constructed a simple datapath to implement the MIPS instruction set**
- **We revised this datapath to provide a 5 stage pipeline (with hazard avoidance / resolution)**
  - **And other processor architectures**
- **We then looked at additional hardware to support:**
  - **Exceptions / interrupts**
  - **Instruction and data caches**
  - **Virtual memory**

Why Do We Have Procedures?

- **We tend to take procedure / sub-routine calls "for granted"**
  - **What is actually needed for such a call?**
  - **And why?**

- **Why?**
  - **Think about writing a program in a language in which the only flow control statement is a (possibly conditional) goto**
  - **How could you re-use parts of your code?**
  - **How do you know where you "came from"?**

Procedure Requirements

- We need a means of jumping to a specific location
  - The section of code we wish to re-use
- We need a way of passing information to the procedure
  - Argument passing
- We may need a way to return information
  - Return values
- We need a way of returning back where we came from
  - Return statement

Royal Holloway University of London

MIPS Procedure Calling

- **Jumping to a procedure and returning**
  - **JAL Instruction (Jump and Link)**
    - **Puts new address into the program counter**
    - **Puts current address (+4) into a register ($ra)**
  - **JR Instruction (Jump Register)**
    - **Puts an address from a register ($ra) into the program counter**
- **Examples**
  - `jal  10000    # call procedure at 10000`
  - `jr   $ra      # return to callee`

Why Not Have a "RET" Instruction?

- **There is no need**
  - **It would not be any quicker**
- **The register number must be in the instruction**
  - **$ra is just one of the registers**
- **If you really want one, create a "pseudo-instruction" in the assembler**
  - `ret → jr $ra`

Argument Passing and Value Returning

- **Argument Registers**
  - **$a0 - $a3 used to pass arguments to the procedure**
- **Return values**
  - **$v0 - $v1 used to return values to the caller**
- **These are Compiler / Assembler *conventions***
  - **There is no specific hardware support here**
  - **(Nor is there any needed)**

More Than Four Arguments?

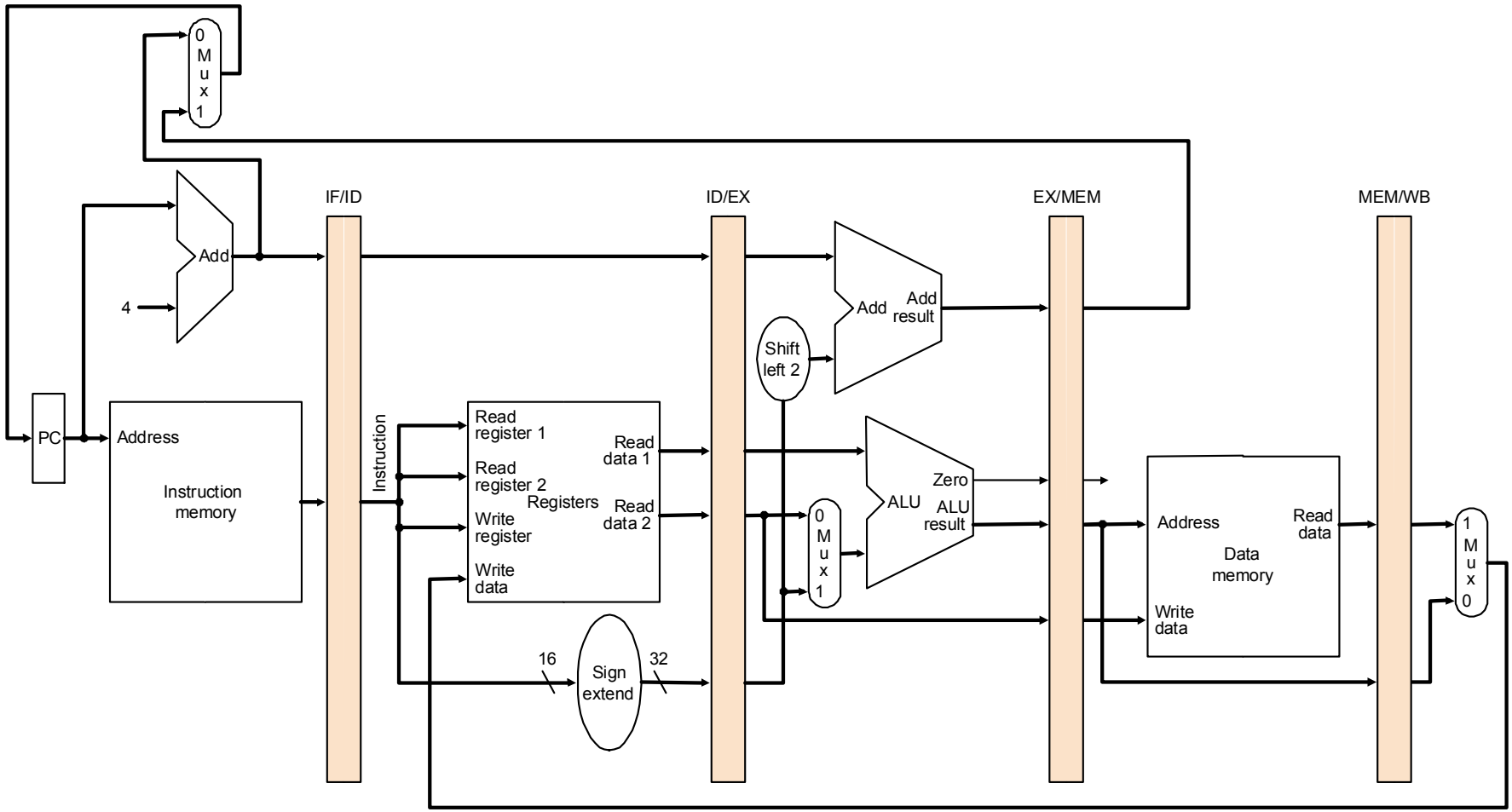
- **If there are more than four arguments, they are placed on "the stack"**
  - **This is a last-in-first-out data structure**
  - **Notional operations are "push" and "pop"**
- **Where is the stack?**
  - **The top is pointed to by the "stack pointer" register ($sp)**
  - **By *convention* the stack grows "downwards" in address**

MIPS Stack Implementation

- $sp is just another register
- To store one or more values (push):
  - Copy them to locations starting at $sp
  - Adjust $sp to point to the new "top" of the stack
- To remove one or more values (pop):
  - Copy them from locations below $sp
  - Adjust $sp to point the new "top" of the stack

Why Not Have "PSH" and "POP"?

- **It would appear useful to have a single instruction that combine the actions necessary for a push**
- **And another that combines the actions for a pop**
- **But:**
  - **How does this fit into the datapath?**
  - **How can it be implemented in hardware?**
  - **How does it fit in with pipelining?**


0
M
u
x
1
Add
4
PC
Address
Instruction memory
IF/ID
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ID/EX
Shift left 2
Add
Add result
0
M
u
x
1
ALU
Zero
ALU result
EX/MEM
Address
Data memory
Write data
Read data
MEM/WB
1
M
u
x
0

Push and Pop Pseudo-Instructions

- **If we really need push and pop, implement these as "pseudo-instructions" in the assembler**

```
– psh $t0                  # push $t0 onto stack
    ▪ becomes
– SW $t0, 0($sp)           # store value on stack
– sub $sp, $sp, 4          # stacks grow down

– pop $t0                  # pop $t0 from stack
    ▪ becomes
– lw $t0, 0($sp)           # load value from stack
– add $sp, $sp, 4          # stacks shrink upwards
```

What About Other Registers?

- We have, by convention, uses for $a0 - $a3 and $v0 - $v1
- Is the procedure free to use the other registers?
  - The calling code may already be using them…
- Again, we use a ***convention***
  - The procedure may use $t0 to $t9 as required
  - If the procedure wishes to use $s0 to $s7:
    - It may do so, but it must restore their previous values prior to returning to the caller
    - I.e. it must save their value somewhere
  - Where stored? – On the stack

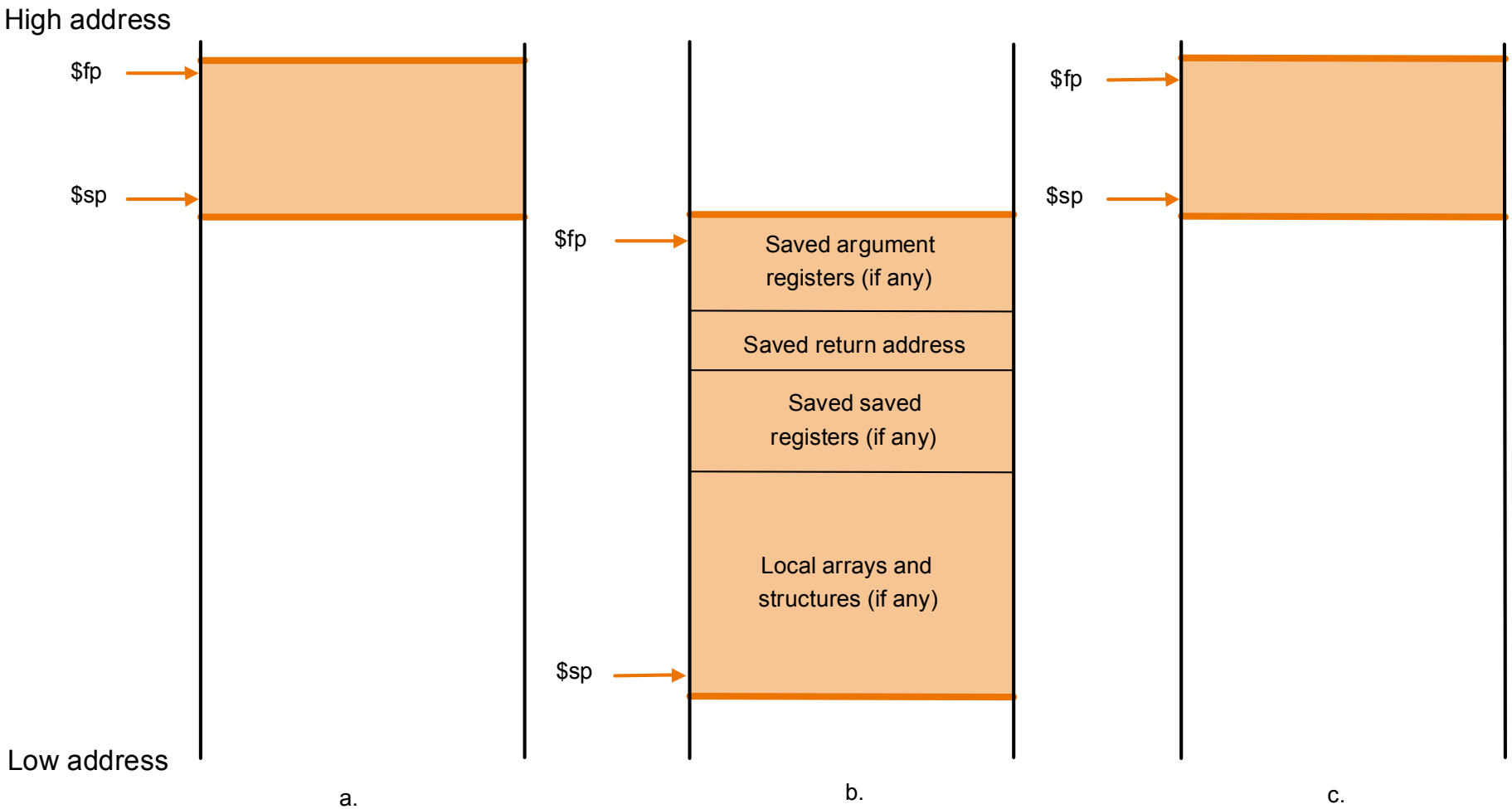
What About Nested Procedures?

- **Nested procedures must abide by the same conventions**
  - **But in addition, they must also store the return address ($ra)**
  - **And the argument registers**
  - **Where stored? – On the stack**
- **This also works for recursive procedures**
  - **They must abide by the same conventions**

What If It Needs More Storage?

- **What if the procedures require more storage than is available in the registers?**
  - **It will need to use main memory**
- **But where?**
  - **Also on the stack**
- **How does the procedure know where its "local" data is on the stack?**
  - **It is relative to another register, the "frame pointer" register ($fp)**
  - **(It can also be found relative to the stack pointer, but this requires more arithmetic)**

The Stack - Putting It All Together



Royal Holloway University of London

MIPS Register Conventions

Name	Number	Usage	Preserved?
$zero	0	constant 0	n.a.
$at	1	[reserved for assembler]	n.a.
$v0 - $v1	2 – 3	return values	no
$a0 - $a3	4 – 7	arguments	no
$t0 - $t7	8 – 15	temporaries	no
$s0 -	16 – 23	saved	yes

Summary

- **We have looked at support for procedure calls**
  - **The JAL instruction is the only additional instruction required to support procedures**
  - **Everything else is based on particular register usage**
    - **By convention**
  - **But all registers are general purpose**
    - **You can (in assembler) use them however you like**

Next Lecture

- **Revision**
  - **Revision topics**
  - **Past papers**
  - **Anything I might have missed**