Tutorial 9

- Code Generation

Code Generation

- Idea: Convert WLP4 to MIPS
  - ↳ Assemble MIPS to machine code to run our program!
- Convention: WLP4 programs will:
  - ↳ Use $1 & $2 for input (wains parameters)
  - ↳ Use $3 for ouput (wain's return value)
  - Done via mips.twoints or mips.array
- General Code Gen for Variables:

eg: Generate code for the following:

```
int wain (int a, int b) {
    int c = 42;
    return a;
}
```

↳ Store all WLP4 vars on the stack $30

→ All vars must be declared 1st in our wain & procedures. Easy to identify

→ But what if in our MIPS code another procedure modifies the stack? How do I track my current procedures vars? Need a frame pointer

↳ Frame pointer ($29): A dedicated register always fixed to a fixed MEM address within $30 while executing a procedure.

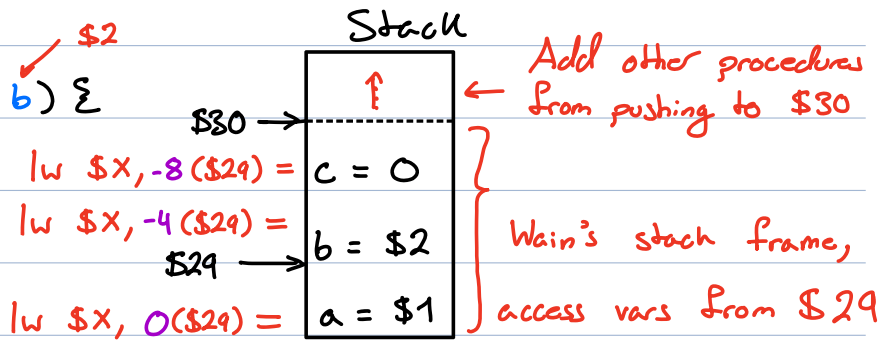
→ We can use fixed offsets w.r.t. the unchanging $29 → access vars in the current stack frame!

→ Note: Often we set $29 = the 1st value in the frame of a procedure

From our eg:

```
int wain (int a, int b) {
    int c = 42;
    return a;
}
```

($1 = a, $2 = b)



Code gen:

```
; Prologue - Push all vars onto the stack
lis $4
.word 4
sub $29, $30, $4    ; Set $29 = 1st var on stack a
sw $1, 0($29)       ; Push a = $1 onto the stack
sub $30, $30, $4    ; Update $30 = $30 - 4
sw $2, -4($29)      ; Push b = $2 onto the stack
sub $30, $30, $4    ; Update $30 = $30 - 4
lis $3
.word 42
sw $3, -8($29)      ; Push c = 42 onto the stack
sub $30, $30, $4    ; Update $30 = $30 - 4
```

```
; Begin Code
lw $3, 0($29)  ; a's offset to $29 is 0

; Epilogue - Restore stack & return
add $30, $30, $4   } ; Programatically we reset $30
add $30, $30, $4   } ; to represent "popping off" the
add $30, $30, $4   } ; wain/procedure
jr $31
```

Code Gen for expressions

eg: Generate the code for the following WLP4 program:

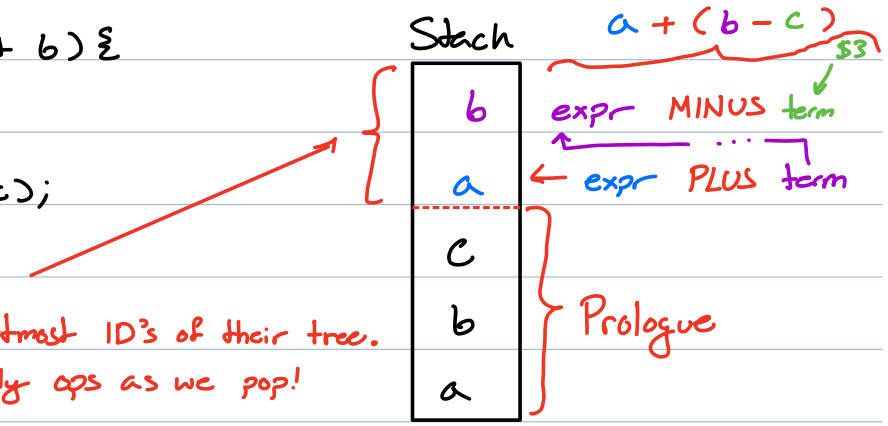
```
int wain(int a, int b){
    int c = 3;
    return a + (b - c);
}
```

How do we enforce precedence? ( )

How do we store intermediate steps?

- Problem: How do we store intermediate steps of any expression?
  - ↳ Use temp registers? Limited & not scalable
- Idea: Store steps of an expression in the stack!
  - ↳ recursively push non-terminals (push from $3)
  - ↳ pop at terminals & calculate expressions (pop to $5)
    - inside-out (according to precedence in our parse tree!)
    - → ✱ our grammar enforces precedence in our expressions, so all we need to do is traverse our tree with push & pop!

```
int wain (int a, int b) {
    int c = 3;
    return a + (b - c);
}
```



Code Gen:

```
; Prologue — Push wain's vars onto $30
lis $4
.word 4
sub $29, $30, $4 ← ; $29 = $30-4
sw $1, -4($30)   } ; Push a = $1
sub $30, $30, $4 }
sw $2, -4($30)   } ; Push b = $2
sub $30, $30, $4 }
lis $5           } ; Push c=3 (use $5 by convention)
.word 3          }
sw $5, -4($30)   }
sub $30, $30, $4 }

; Begin Code
lw $3, 0($29)    ; load a into $3
sw $3, -4($30)   } ; Push a onto $30
sub $30, $30, $4 }
lw $3, -4($29)   ; load b into $3
```

```
sw $3, -4($30)   } Push b onto $30
sub $30, $30, $4 }

lis $3, -8($29)  ; $3 = c
add $30, $30, $4 } ; Pop b into $5 ($5 = b)
lw $5, -4($30)   }
sub $3, $5, $3   ; $3 = b - c
add $30, $30, $4 } ; Pop a into $5 ($5 = a)
lw $5, -4($30)   }
add $3, $5, $3   ; $3 = a + (b - c)
```

```
; Epilogue - Unwind Stack
add $30, $30, $4 }
add $30, $30, $4 } ; 3 vars = wain's stack frame
add $30, $30, $4 }
jr $31
```

Adding New Expressions

eg: C & C++ have pre & post increment operators ++i & i++. If we added the following rules to WLP4:

factor → PLUS PLUS lvalue (++i)

factor → lvalue PLUS PLUS (i++)

Assume these are only applied to INT types.

Write pseudocode to generate the correct MIPS output for each rule.

- ++i code gen
  - ↳ Increases the value of i by 1, then returns i
  - ↳ Note: this is a factor rule, not an lvalue. We cannot nest (++++i is invalid)

```
void genCode (tree t) {
    ...
    if (t.rule = "factor -> PLUS PLUS lvalue") {
        // gen code to put lvalue in $3
        genCode (t.children[2]); // $3 = lvalue
        // Recall: lvalue is a MEM address, load the actual value
        lw $5, 0($3); // $5 = RHS of lvalue
        // add one to lvalue
        lis $11
        .word 1
        add $5, $5, $11
        // save new value in MEM
        sw $5, 0($3)
        // Return new value
        add $3, $5, $0
    }
}
```

- i++ code gen
  - L> return i, increment i by 1
  - L> need to remember the old value to return in $3

```
void genCode (tree t) {
    ...
    if (t.rule = "factor → lvalue PLUS PLUS") {
        genCode (t.children[0]); // $3 = lvalue
      * lw $5, 0($3);            // $5 = RHS of lvalue
        add $6, $5, $0;          // $6 = $5, copies the lvalue
        lis $11
        .word 1
        add $5, $5, $11          // add one to lvalue
        sw $5, 0($3)             // save new value in MEM
        add $3, $6, $0           // Return old value
    }
}
```

* Note: for 'lw $5, 0($3)' we assume that the full address in $3 is not offset. Otherwise, we would need to do something like: 'add/sub $3, $3, $29' to reorient it to our current frame.