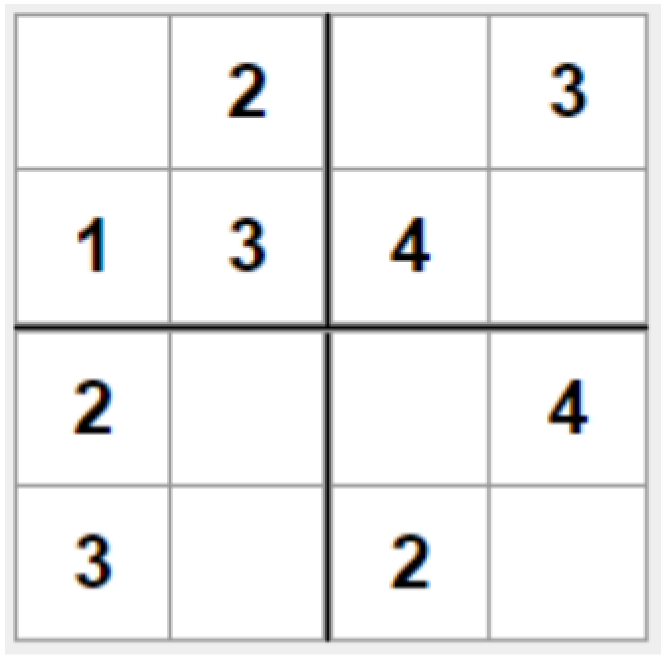
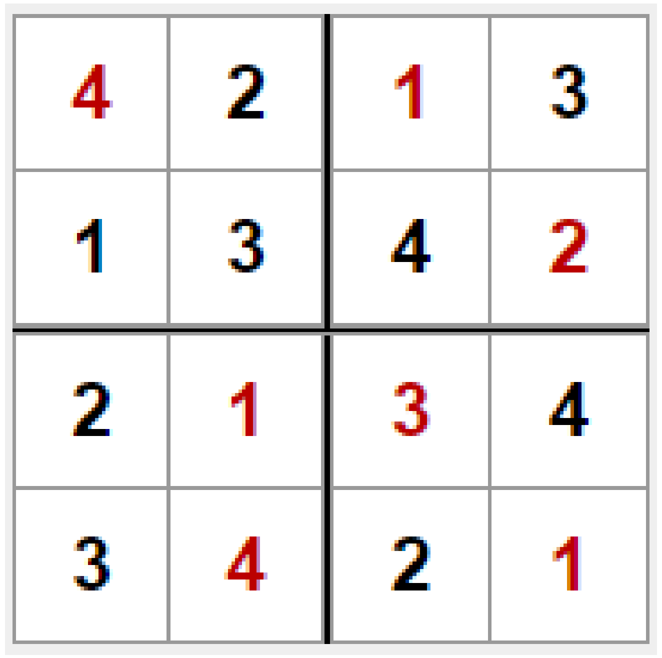
Lab 1
Mini-Sudoku

Minisudoku

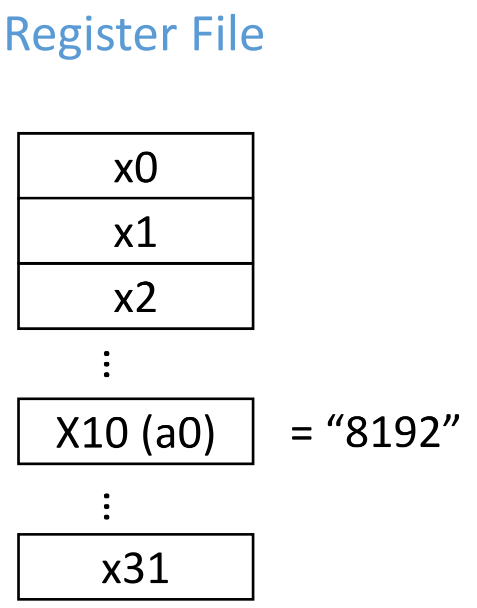


	2		3
1	3	4	
2			4
3		2	



4	2	1	3
1	3	4	2
2	1	3	4
3	4	2	1

At the beginning of function call



Memory

8192: 0 4 3 0 0 0 4 2 0 2 0 0 3 0 0 0

“lb t0 0(a0)” will load “0” to t0

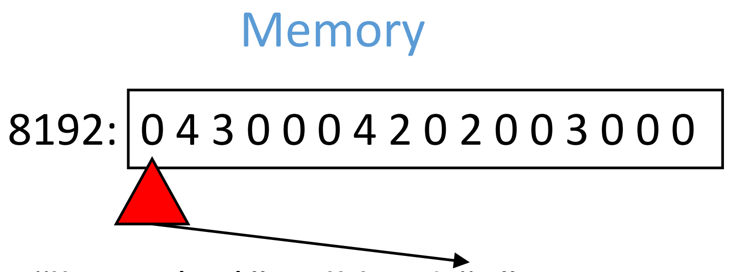
Our Goal:

8192: 2 4 3 1 1 3 4 2 4 2 1 3 3 1 2 4

Recursive Algorithm For Sudoku

```
boolean solve(int index) {
        if ( index >= 16 ) return true;
        if ( set[index] > 0 ) return solve(index+1);
        else {
                for ( n = 1 to 4 ) {
                        set[index] = n;
                        if ( check(index) and solve(index+1) ) return true;
                }
                set[index] = 0; // returns the value to 0 to mark it as empty
                return false; // no solution
        }
}
```

Recursive Algorithm For Sudoku



```
"lb t0 0(a0)" will load "0" to t0
```

Current value is "0" (blank).
Must try out different values for this slot

```
addi t0 zero 1
sb t0 0(a0)
```

Let's try "1"

```
# store ra and a0 in stack
jal ra check
# restore from stack, move a0 to, say, t0
# if return value is 1, try another number for "sb" ("1"?)
            # if all four numbers have been tried, set a0 to "1" and ret

# if return value is 0, store a0, etc in stack, add 1 to a0, call "solve" recursively
```

Call "check" (we need to implement this)
Returns 0 in a0 if sudoku restrictions met, 1 otherwise

Recursive Algorithm For Sudoku

- “solve” returns true if
  - Index is >= 16 (must keep track of index in addition to address in a0
- Returns false if
  - “Check” returns false
  - Recursive “solve” returns false

- Easy input:
  - a0 stores address of the first element in array
  - a1 stores index

“Check” Implementation

- Easy input:
  - a0 stores address of first element in array,
  - a1 stores index

```
Let row = index/4;
Let col = index%4;
Let blk = ((index/8)*8)+(col/2)*2; // index of one of the 4 sub-grids
```

RV32i doesn’t have “MUL”/”DIV” must use shift!

```
for ( c = 0 to 3 ) {
        rowval = set[row*4+c];
        colval = set[c*4+col];
        blockval = set[blk+(c/2)*4+(c%2)];
        // check if rowval, colval, blockval are unique
}
```

“Check” Implementation

- Checking for uniqueness
  - Allocate 12 bytes in stack (addi sp -12) and set all to zero
  - For each nonzero rowval, check if sp+rowval is zero. Collision if not
  - For each nonzero colval, check if sp+4+rowval is zero. Collision if not
  - For each nonzero blockval, check if sp+8+rowval is zero. Collision if not
  - De-allocate stack and return

Implementation tips

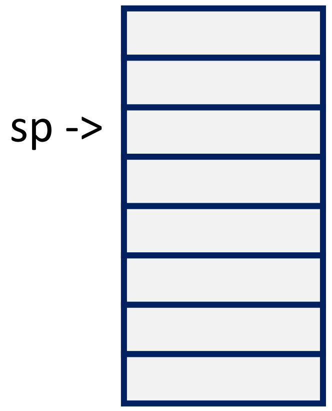
- Break everything into functions!
  - “solve” calls “trysolve” four times, for each candidates
  - “trysolve” calls “check” and “solve” at most once each
  - “check” calls “getrowval” “getcolval” “getblockval” four times each

- As long as calling conventions are maintained
  - Use stack to store and restore a and t registers
  - Each function should not be too difficult

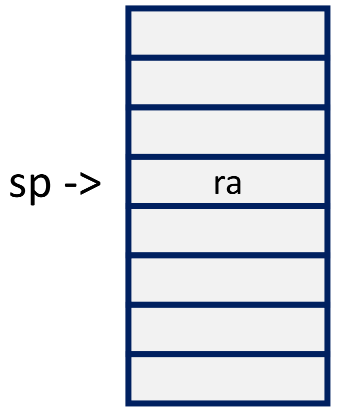
Suggested implementation order

- Implement “check”
  - Modify inputs at sudoku.s to test check
- Modify input to have only one zero
  - Simply loop index from 0 to 15 to discover the zero
  - Try four values, run check each time. Return if true
- Use original input, implement recursive function
  - Get index as argument, try setting that index 1 to 4, check each time
  - If false, return false. If true, recursively call “solve” with idx+1 and return that

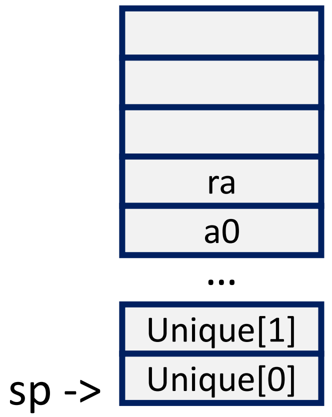
Stack modifications during execution



“solve” called

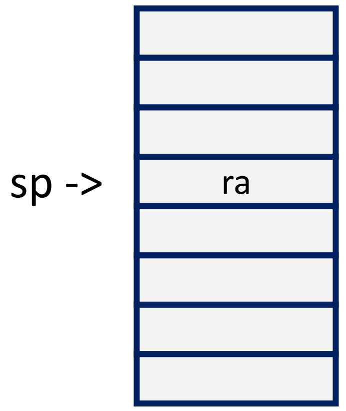


Store ra



“check” called
Register “ra” now has new value
Register “a0” now had new value

Caller (“solve”) stores a0 in stack
Callee (“check”) does not store ra again
because check does not call other functions



Return from “check”
a0 restored from stack
ra will be restored before
returning from “solve”