Code generation

Activation Record

```
void A() {
    int a;
    short b[4];
    double c;

    B();
    C();
}
```

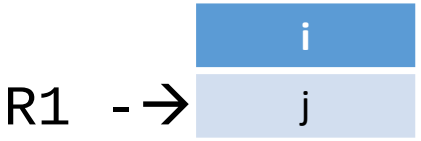
```
void B() {
   int X;
   char *y;
   char *z[2];

   C();
}
```

```
void C() {
  double m[3];
  int n;
  ..
}
```

- 20 bytes in the activation record for A, 16 for B and 28 for C
- When A is called, stack pointer is decremented by 20 bytes to point to the base address of A
  - A calls B, stack pointer decremented by 16 bytes to point to the base address of B
  - B calls C, 28 more
  - C returns, stack pointer just shifts up (no content destroyed)
  - B returns, stack pointer shifts up
  - A calls C, the activation record for C just overlays the same place as occupied by B earlier

Assembly code generation by compilers




```
int i;                          M[R1+4] = 10;
int j;
i = 10;
j = i + 7;                      R2=M[R1+4]; R3=R2+7; M[R1]=R3;
j++;                            R2=M[R1];R2=R2+1; M[R1]=R2
```

- Base address of activation record in register R1
- All operations deal with 4 byte quantities

Assembly code generation by compilers

```
int i;
short s1, s2;
i = 200;            M[R1+4] = 200;
s1 = i;             M[R1+2] = M[R1+4]; -- not allowed
                    R2 = M[R1+4]; M[R1+2] = R2 -- will overwrite i
                    R2 = M[R1+4]; M[R1+2] = .2 R2 – 2 bytes copied
s2 = s1 + 1;        R2 = M[R1+2]; -- will copy 4 bytes
                    R2 = .2 M[R1+2]; R3 = R2 + 1; M[R1] = .2 R3
```

- All instructions have an implicit .4 unless otherwise mentioned

Assembly code generation by compilers

```
int array[4];
int i;
for (i = 0; i < 4; i++) {
   array[i] = 0;
}
i--;
```

```
M[R1] = 0;
R2 = M[R1];
BGE R2, 4, PC + 40;
R3 = M[R1];
R4 = R3 * 4;
R5 = R1 + 4;
R6 = R4 + R5;
M[R6] = 0;
R2 = M[R1];
R2 = R2 + 1;
M[R1] = R2;
JMP PC – 40;
R2 = M[R1];
R2 = R2 – 1;
M[R1] = R2;
```

Pointers and casts

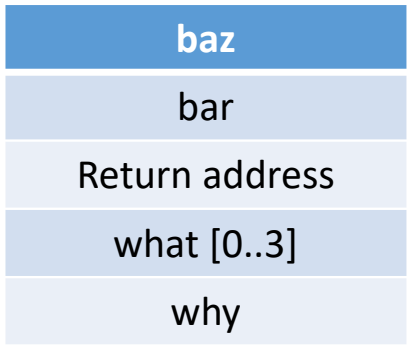
```
struct fraction {
  int num;
  int denom;
};
struct fraction pi;

pi.num = 22;              M[R1] = 22;
pi.denom = 7;             M[R1+4] = 7;

((struct fraction *) &pi.denom)-> denom = 451;  M[R1+8] = 451;
```

Code generation with activation records

```
void foo (int bar, int * baz) {
    char what[4];
    short *why;
    ....
}
```



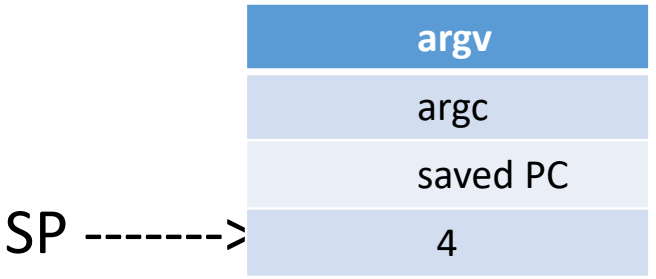
- local variables and parameters are all packed closely in memory
- 20 byte activation record that makes up any call for foo
- Parameters are laid down high to low address
- Local variables stacked in the order they appear
- Parameters appear in the order in the prototype
- Return address specifies where to return

Code generation with activation records

```
Int main (int argc, char **argv) {
    int i = 4;
    foo (i, &i);
    return 0;
}
```

- Very first thing a C function does is make space for its local variables

```
SP = SP – 4;
M[SP] = 4;
```



Code generation with activation records

```
SP = SP – 4;
M[SP] = 4;
SP = SP – 8;
R1 = M[SP+8];
R2 = SP + 8;
M[SP] = R1;      // for i
M[SP+4] = R2; // for &I
CALL <foo>      // Jump to the first assembly code inside foo
SP = SP+8       //  jump back when done – this address is laid down at
                    Saved PC2 automatically by the caller instruction
```

X:

argv
argc
Saved PC1
4
X
4
Saved PC2

- When foo is done executing, it has information where to jump back

Code generation with activation records

```
void foo (int bar, int * baz) {
    char what[4];
    short *why;
    why = (short *)(what + 2);
    *why = 50;
}
```

- foo has to make space for accommodating the variables what and why

Code generation with activation records

```
foo:  SP = SP – 8;
      R1 = SP+6;
      M[SP] = R1;
      R1=M[SP];
      M[R1] = .2 50 // 2 byte figure
      SP = SP + 8  // Deallocate, return to main
      RET          //  Brings SP up by 4 more bytes, populates the PC
                        with whatever is written in Saved PC2
```

argv
argc
Saved PC1
4
X
4
Saved PC2
what
why

Code generation with activation records

```
SP = SP + 8;   // back to main, deallocates space for parameters
RV = 0;
```

- RV is the return value register
  - 4 byte register dedicated to communicate return values between callee and caller functions
  - Whenever a function returns, the caller knows where to look to get the return value
  - Did not have a return value for the foo function

A general activation record