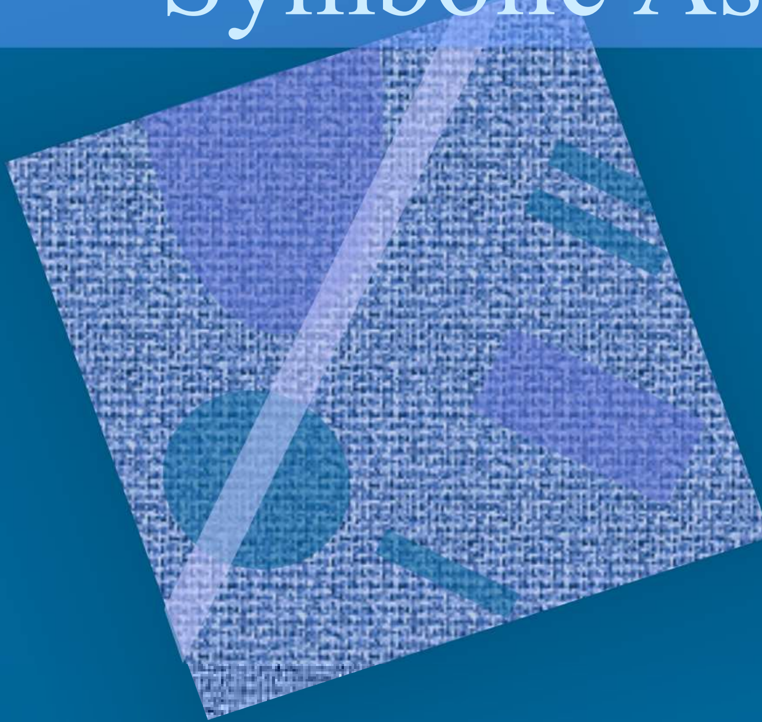
Lecture 4

Subroutine Implementation with Symbolic Assembly Language



Subroutine types,

Parameters

Activation record (AR)

Activation record stack

Recursion

Copyright Teemu Kerola 2020

Subroutine Types

- High level language concepts
  - Subroutine, procedure
    - Parameters (input and output parameters)
  - Function
    - Parameters, return value
  - Method
    - Parameters, possibly return value
- Assembly language concept
  - Subroutine
    - Parameters, possible return value(s)

Copyright Teemu Kerola 2020

Parameters and Return Value

- Formal parameters
  - Defined in subroutine at programming time
  - Certain order and type
  - Return value(s)
    - Handled very much the same way as parameters

```
Print (int x, y)
void Print (int x, y)

Comp(int x): int
int Comp(int x)
```

- Actual parameters and return value
  - Actual parameters are inserted instead of formal parameters at subroutine call time at execution time
  - Return value is obtained at subroutine return time and it is used like any other value

```
Print (5, apu+1);
x = Comp( y+234);
```

Parameter Types

- Call by value parameter
  - Copy of actual parameter value is passed
  - Copied value can be read
  - Copied value can be modified, but original parameter value is safe
- Call by reference parameter
  - Address of actual parameter is passed (it must have address!)
  - Parameter value and address can be read, value can be written
  - Subroutine can modify data in calling routine! Output parameter?
- Call by name parameter *(in interpreted macros or scripts)*
  - Name (string) of actual parameter is passed
  - Name (string) is mapped to value at call time
  - Semantics is determined only at call time

swap(i, j) -- macro, call by name parameters

```
tmp = i;
i = j;
j = tmp;
```

```
swap(x, y);
```

→

```
tmp = x;
x = y;
y = tmp;
```

```
swap(k, T[k]);
```

→

```
tmp = k;
k = T[k];
T[k] = tmp;
```

Discuss

Components of Subroutine Implementation

- Return address
  - Address of the next instruction after the call
- Parameter passing
- Return value passing
- Local variables and other data structures
- Register allocation
  - Calling routine wants to retain values of all registers that it is using
    - Main program, other subroutine, same subroutine, …
  - Subroutine must at the beginning save the values of all (working) registers that it is using, and at the end recover their original values

Copyright Teemu Kerola 2020

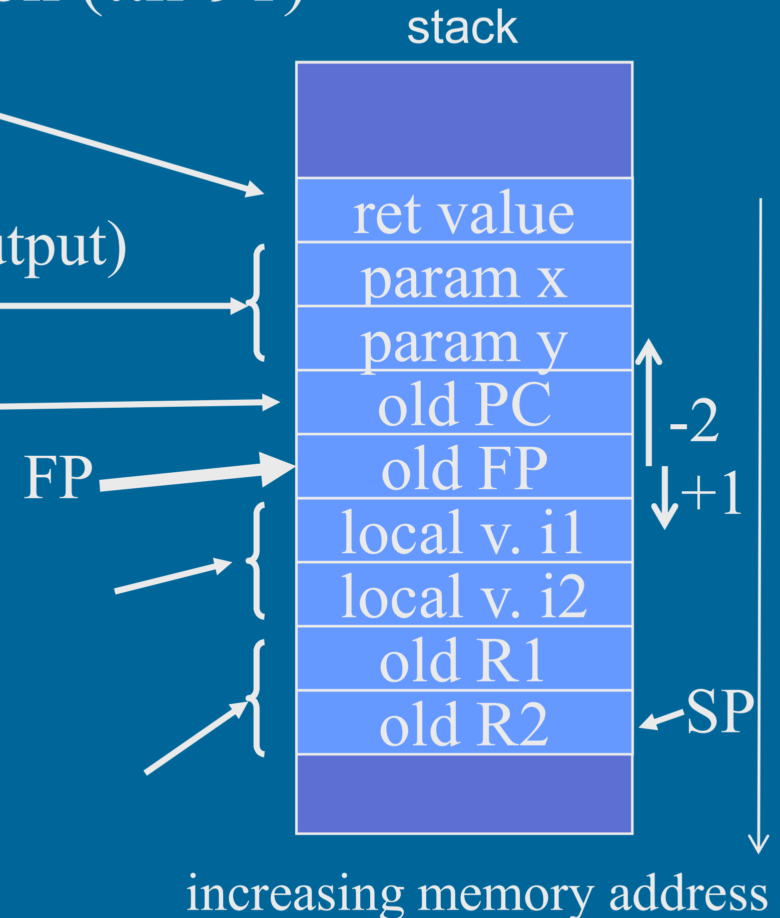
Activation Record (AR) (ttk-91)

(activation frame)

(aktivaatiotietue)

int funcA (int x,y);

- Subroutine implementation (ttk-91)
  - Function return value (return values)
  - Values of all (input and output) parameters
  - Return address
  - Calling routine (old) AR
  - Local variables and other data structures
  - Original values of saved (working) registers



Copyright Teemu Kerola 2020

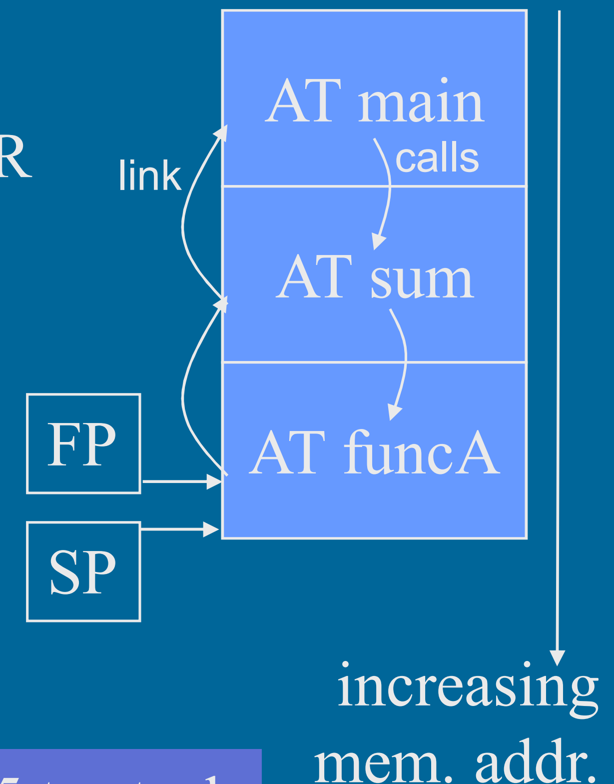
AR Stack in Memory (ttk-91)

aktivaatio-tietuepino

- ARs are reserved and dismantled dynamically (at execution time) from stack (in memory)
  - SP (i.e., R6) points to top of stack
- Activation record stack
  - FP (i.e., R7) points to current AR in <u>specific place</u> (in ttk-91: location of old FP)
- AR (in stack) is built and dismantled with instructions:
  - PUSH, POP, PUSHR, POPR
  - CALL, EXIT (SVC, IRET)

Push value to stack

Push all R0-R5 to stack



Copyright Teemu Kerola 2020

Subroutine Call Implementation

- Implementation split to various units

Calling routine
- Reserve space from stack for return value
- Put parameters (values/addresses) to stack
- Save old PC and FP, set new PC and FP ← CALL instr

Called routine
- Allocate space for local variables (prolog)
- Save old values for used registers to stack (prolog)
- (subroutine implementation – actual work)
- Recover saved register values (epilog)
- Deallocate space for local variables (epilog)
- Recover old PC and FP (epilog) ← EXIT instr
- Deallocate parameter space from stack (epilog) ← EXIT instr

Calling routine
- Take return value from stack

Subroutine Example (13)

```
int fA (int x, y)
{
        int z = 5;

        z = x * z + y;
        return (z);
}

…
T = fA (200, R);
```

subroutine use:

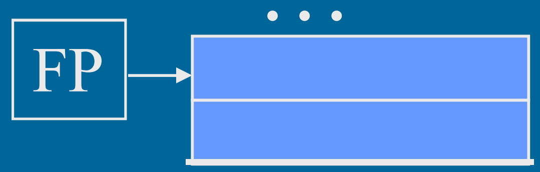
```
R       DC  24
…
PUSH  SP,=0 ; space for ret val
PUSH SP, =200
PUSH  SP, R
CALL  SP, fA
POP      SP, R1
STORE R1, T
…
```

from mem to mem!!

save PC, FP
set PC,
call & return
recover FP, PC
dealloc par space

2nd operand is always reg

current FP



Subroutine example

(no animation)

```
int fA (int x, y)
{
        int z = 5;

        z = x * z + y;
        return (z);
}

…

T = fA (200, R);
```

Subroutine use:

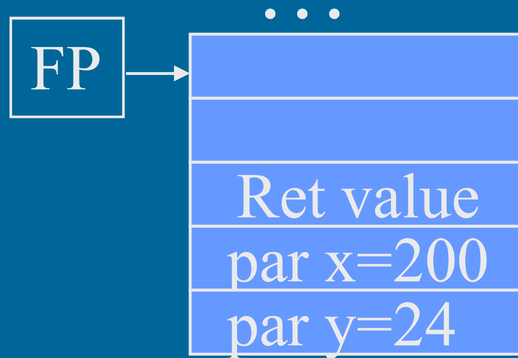
```
R       DC  24
…
PUSH  SP,=0 ; ret. value space
PUSH SP, =200
PUSH  SP, R

CALL  SP, fA

POP     SP, R1
STORE R1, T
…
```

From mem to mem!!

save PC, FP
set PC,
call & return
recov FP, PC

2nd oper is always register

current FP



Copyright Teemu Kerola 2020

Subroutine Example(12)

```
int fA (int x, y)
{
        int z = 5;

        z = x * z + y;
        return (z);
}
…
T = fA (200, R);
```

all references done relative to FP

ret value

Subroutine implementation:

```
retfA   EQU  -4 # return value
parX    EQU  -3 # params
parY    EQU  -2
locZ    EQU  1  # local vars

fA      PUSH  SP, =0 ; alloc Z
        PUSH  SP, R1 ; save R1

        LOAD  R1,=5; init Z
        STORE R1, locZ (FP)

        LOAD R1, parX (FP)
        MUL   R1, locZ (FP)
        ADD   R1, parY (FP)
        STORE R1, locZ (FP)
        STORE R1, retfA (FP)
        POP    SP, R1; recover R1
        SUB    SP, =1 ; free Z
        EXIT   SP, =2 ; 2 param.
```

prolog

epilog

Copyright Teemu Kerola 2020

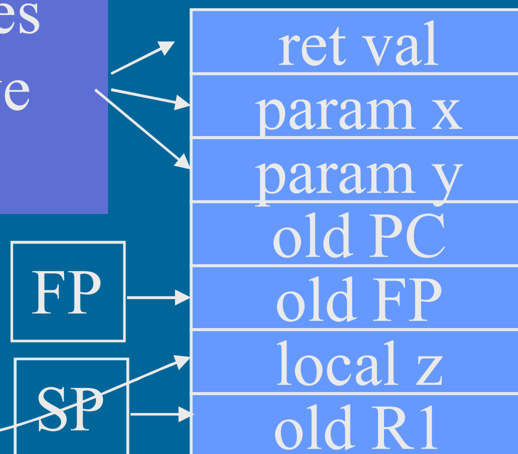
Discuss

(no animation)

Subroutine Example

```
int fA (int x, y)
{
        int z = 5;

        z = x * z + y;
        return (z);
}

…

T = fA (200, R);
```

all references done relative to FP



subroutine imlementation:

```
retfA   EQU  -4
parX    EQU  -3
parY    EQU  -2
locZ    EQU   1

fA      PUSH  SP, =0 ; alloc Z
        PUSH  SP, R1  ; save R1

        LOAD  R1,=5; init Z
        STORE R1, locZ (FP)

        LOAD R1, parX (FP)
        MUL   R1, locZ (FP)
        ADD   R1, parY (FP)
        STORE R1, locZ (FP)
        STORE R1, retfA (FP)
        POP    SP, R1; recover R1
        SUB    SP, =1 ; free Z
        EXIT   SP, =2 ; 2 param.
```

prolog

epilog

Copyright Teemu Kerola 2020

Call by Ref Params

Use T = fB(R, S, T)

```
R        DC  24
S        DC 56
T        DC 77
pT       DC  0     ; pointer
…
LOAD   R1, =T    ; initialize pT
STORE  R1, pT
…
PUSH  SP,=0 ; space for ret val
PUSH SP, R  ; call by value
PUSH  SP, =S ; call by reference
PUSH  SP, pT; call by reference
CALL  SP, fB
POP       SP, R1
STORE    R1, T
…
```

Address of variable S
Value of pointer variable pT is address of variable T

Y and Z are call by reference:

```
retfB    EQU  -5  ; return value
parX     EQU  -4  ; call by value
vparY EQU  -3  ; call by reference
vparZ EQU  -2  ; call by reference

fB       PUSH  SP, R1 ; save R1

         LOAD R1, parX (FP)
         MUL   R1, @vparY (FP)
         ADD   R1, @vparZ (FP)
         STORE R1, retfB (FP)
         POP     SP, R1; recover R1
         EXIT   SP, =3 ; 3 param.
```

Procedural OS Service Call

- Same way (e.g.) as subroutine call
  - Instead of CALL instruction use SVC (SuperVisor Call)
- Space for return value?
- Parameters in stack or in registers? (each OS service can be different)
- SVC SP, =5
  - Service routine number as operand
  - Saves PC, FP and SR?
  - Sets PC, FP and SR
- Return with IRET (e.g.)
  - Recover PC, SR and SR
- Remove return value (OK, error) from stack

fOK = ReadBlock (fp, 64)

```
...
PUSH  SP, =0 ;ret val
PUSH  SP, =FileBuffer
PUSH  SP, CharCnt
PUSH  SP, FilePtr

SVC    SP, =ReadBlock

POP      SP, R1
JNZER  R1, FileTrouble
…
```


Copyright Teemu Kerola 2020

Subroutine Call Implementation

- Implementation split to various units

Calling routine
- – Reserve space from stack for return value
- – Put parameters (values/addresses) to stack

CALL instr
- – Save old PC and FP, set new PC and FP

Called routine
- – Allocate space for local variables
- – Save old values for used registers to stack

prolog

- – (subroutine implementation – actual work)
- – Recover saved register values
- – Deallocate space for local variables

EXIT instr
- – Recover old PC and FP
- – Deallocate parameter space from stack

epilog

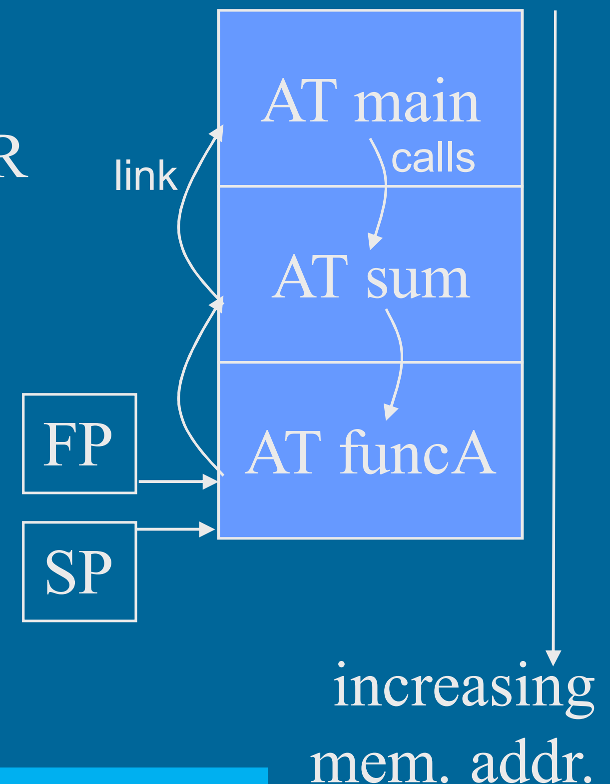
Calling routine
- – Take return value from stack

Avoid? Make faster?

Copyright Teemu Kerola 2020

AR Stack in Memory (ttk-91)

- ARs are reserved and dismantled dynamically (at execution time) from stack (in memory)
  - SP (i.e., R6) points to top of stack
- Activation record stack
  - FP (i.e., R7) points to current AR in specific place (in ttk-91: location of old FP)
- AR (in stack) is built and dismantled with instructions:
  - PUSH, POP, PUSHR, POPR
  - CALL, EXIT (SVC, IRET)



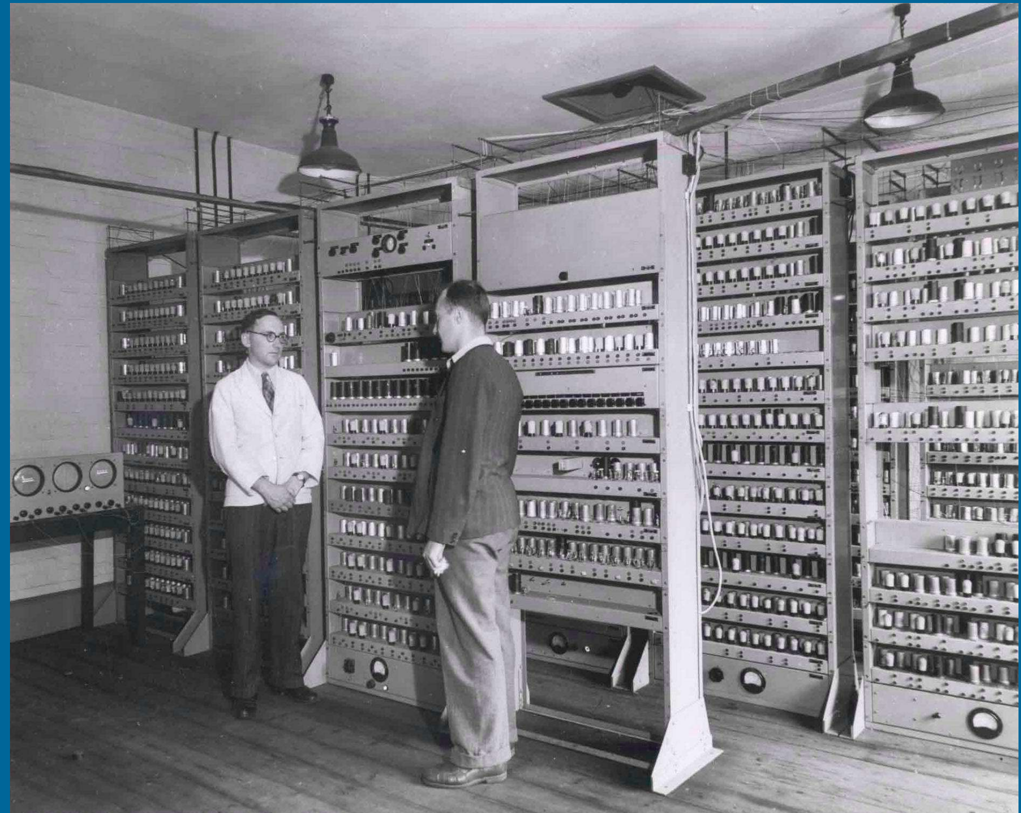
Reference sum in funcA???

Copyright Teemu Kerola 2020

-- END --

M. Wilkes:
EDSAC (1949)

- Register (6) with vacuum tubes
- Instr and data memory,
  32 mercury delay lines, each
  32 words á 18b
- 2's complement int
- Int mult 5.4 ms
- Fractions -1 < x < +1
- No DIV or CALL instr
- Aver speed 650 IPS (instr per sec)
- 1st "stored program" computer (fast instr mem)
- 3000 vacuum tubes, power cons. 12 kW, space 5x4 m
  - Vacuum tubes lasted longer, if they were never turned off



http://www.cl.cam.ac.uk/Relics/archive_photos.html

Copyright Teemu Kerola 2020